

Intro Lab Session 4

Introduction to Blackboard and command line skills

Contents

4.1	Getting started with Blackboard	78
4.1.1	The COMP10120 course unit site	78
4.1.2	Lab deliverables	79
4.1.3	Using discussion boards	79
4.1.4	Using wikis	79
4.1.5	Individual Learning Profile questionnaire	80
4.1.6	Writing your journal	80
4.2	Reinforcing your command line skills	81
4.2.1	Creating a directory structure	82
4.2.2	Copying, moving, and removing files	83
4.2.3	Practice makes perfect	84
4.2.4	Wild cards	86
4.2.5	Quotas	87
4.2.6	Putting commands together	87
4.2.7	Making your own commands	89
4.2.8	Doing several things at a time: background running	90
4.2.9	Printing text files: lpr	91
4.2.10	Time for a checkup	91
4.2.11	Exercises	92
4.2.12	The end of the beginning	93
4.3	Acknowledgements	94

These notes are available online at

studentnet.cs.manchester.ac.uk/ugt/COMP10120/labscripts/intro4.pdf

You may find it useful to use the online version so that you can follow web links.

This is the last of the introductory labs, and is designed to **introduce** you to the **main virtual learning environment** used across the University, **Blackboard**, and to **give you a chance to practise the command line skills that you've learned already**, plus a **few more**, so that you're ready for when the regular scheduled labs start. As always, **don't rush through the material**, and if you get at all stuck please ask a member of lab staff for help. **All the work today should be done using Linux on a desktop PC.**

4.1 Getting started with Blackboard

In the first half of this session you'll be introduced to the Blackboard Virtual Learning Environment (VLE). Blackboard is designed to support teaching by providing an online place to upload resources for course units. It also provides useful tools such as discussion boards and quizzes. All course units have a Blackboard course unit site, which provide a first place to look for unit resources; we will be looking in particular at ways in which it is used in the First Year Team Project, COMP10120.

Blackboard has several features which make it well suited to supporting COMP10120. These include:

- It's one useful way of communicating with your tutor and members of your tutor group.
- It allows us to give each group a wiki which you'll use to collaboratively document your group's meetings and the decisions you make as part of your project work.
- It provides a structure to help organise activities you should be doing on a week-by-week basis.

As a brief introduction to get you started using Blackboard, please take a look at

<http://studentnet.cs.manchester.ac.uk/ugt/COMP10120/files/BlackboardIntro.pdf>

4.1.1 The COMP10120 course unit site

Now login to Blackboard and navigate to the COMP10120 site and start to look at how the site is structured and what tools have been provided. Here are some things you should note about the structure of the site:

- One of the most important items on the main page is the coloured table of all your activities for this course-unit. The different phases are colour-coded to help you spot the one you want. The row for each week contains links to information about what you should be doing in that week and tools/resources applicable to the week. (Don't forget to read the *phase overview* first.) The *Other* column includes resources for your personal use, that you are expected to complete during this academic year. In particular you should note the *reflective journals* for each week of the first semester. You are expected to reflect on the questions detailed inside the journal each week.
- General instructions for the course unit are located in the *General Project information* area on the front page. Take a look at each of the links in this area. In particular, familiarise yourself with the *Introduction to the tasks and activities* (and then click through to the *Phases and Tutorials* page to get an idea of the course unit structure).
- Back on the course unit site, in the left hand panel you will find *Course Tools*. This link leads you to a number of tools, including the *Course Discussion Board*. There is also a direct link to this board on the front page. You should subscribe to this discussion board now.
- Also on the left hand side, under *My Groups* on the left hand side, you should find a link labelled with your tutorial group name. This is where the resources for your group can be found, including a discussion board which can be used to communicate with members of your group and a wiki in which you can document your work. Again you should subscribe to this discussion board.

4.1.2 Lab deliverables

By the end of this section of the lab session, ensure you have completed the following tasks.

- Posted a welcoming message to your tutor group (see Using Discussion Boards below).
- Create your set of practice wiki pages (see Using wikis below).
- Completed the Individual Learning Profile questionnaire.

4.1.3 Using discussion boards

Most of you will have used discussion boards, or forums, in some way or another, whether on your favourite social networking site or on some other website. Visit your Group Discussion Board on the COMP10120 site and click on *Create Thread* to begin a new discussion thread. Write a short posting to introduce you to your other group members and your tutor. Let them know where you are from, tell them a little bit about yourself.

4.1.4 Using wikis

You have almost certainly come across wikis before, and will no doubt have looked things up in Wikipedia, the world's biggest wiki, many times. Unlike many other online collaboration systems which constrain users in various ways by pre-determining the type of content that can be created (for example sites like Instagram and Flickr are designed for sharing photographs, where as Soundcloud is for sharing audio), and also categorising users as having different types of access (e.g. administrators, moderators, regular users and so forth), the technology behind wikis typically takes a very liberal approach to both users and content. They generally allow any user to make any kind of change to any kind of content, and rely on social conventions established by the community to keep things sane. In particular, every edit, deletion and addition to the wiki is stored, providing a complete history of wiki changes. Old versions of pages can be retrieved and compared to new versions of pages.

As well as creating sites like Wikipedia, wikis are frequently used by software development teams to document their project. The ability to look back to previous versions of documents and for multiple authors to collaborate on producing the documentation without having to worry too much about 'process' is particularly useful in such environments.

During COMP10120 you'll be required to document many aspects of your work using a group wiki provided by Blackboard. You can find this wiki under *My Groups* on the left hand side.

For the purposes of this mini-tutorial you are asked to create some interlinked wiki pages about yourself. It is important that you follow this through to the end as it will ensure you know how to do all the basic tasks needed to help build your own group wiki during the project.

Start by creating a page in your group wiki by clicking on *Create Wiki Page*; it probably makes sense to include your name or initials in its title to avoid confusion with pages created by other group members. You will be presented with the Blackboard editor which won't have any content yet.

Select the *Submit* button. You should now see the first page of your wiki with the content you just added. You can add more content by selecting the *Edit Wiki Content* tab.

Create a short bullet list containing the items *My hobbies*, *My music* and *My files*. Save the page again and check the results, then select the *Edit* tab again. Now turn the three items into wiki

links. To do this, use the **Create Wiki Page to make suitably named new pages** (again including something to make them personal to you, such as your name or initials). Then, back on the page with the bullet list, **select each item of the list in turn and use strange looking icon with several pieces of paper to link the text to each of your new pages.**

This is the basic process by which you build up a wiki into a series of linked pages.

You can find lots of information about Blackboard wikis at <https://en-us.help.blackboard.com/Learn/Student/Interact/Wikis>

Now go back and select the **question mark link for the My music** link and add some content to this page too. You could write about music you love (or music you hate!).

In the last part of this mini-tutorial, **you will need to add a small picture to your wiki** and a small file. First we **need some files to play with**. If you have a small picture that you took yourself, great, you own the copyright on it, otherwise look for a copyright free image on the web.

Finally, browse through the remaining documentation on Wikis at the link given above so that you have an **overview of what other information is there and can refer back to it in the future if needed**. If you have any questions about how to use the wiki tool, post a message to the Course Discussion Board.

4.1.5 Individual Learning Profile questionnaire

Before the end of the lab session make sure **you complete the Individual Learning Profile questionnaire**. You will find it in the **Other** column of the coloured Activities table on the **COMP10120 front page**. This activity is to help you to reflect so that you can understand your own basic skills and abilities required for your academic life. It may also be looked at by your personal tutor so the School can be better placed to address your particular needs during your time in the School of Computer Science. This information will only be visible to you, to your tutor and to the course unit organisers.

4.1.6 Writing your journal

One of the aims of this course unit is to **encourage you to develop the habit of thinking about the way you are learning and working**, and trying to **identify ways** in which these **could be improved**. The process of reflection is key to this, but is **not something that comes naturally to many of us**. In most week slots of the course unit site you will find an instance of the Journal activity. Any entries you put into your journal will not be visible to other students. They will be visible to course unit tutors and course unit organisers, but they will respect your privacy and only your personal tutor will at your journal.

At some time towards the end of this week, start your journal entry for the current week. Some brief notes about the process of Reflection can be found at https://online.manchester.ac.uk/bbcswebdav/courses/I3023-COMP-10120-1161-1YR-020952/Project_instructions/Reflection.html



That's all we want to say about Blackboard. **The second part of this lab is about using the command line in Linux.**

4.2 Reinforcing your command line skills

This section is designed to help you practise your command line skills in preparation for next week's start of regular lab activities. You've already used most of these commands in previous labs, but please don't rush through this section since we'll be explaining their behaviour in a bit more detail, and introducing you to some of the extra options they provide, as well as some of the pitfalls that lie in wait for the over-zealous command line user.

You've probably figured this out already, but it's worth making explicit here: for Unix commands, it's usually the case that no news is good news. So when you run a program from the shell, if you get no response other than your command-prompt back, that almost always means that the command has done what you asked it to (whether you asked it to do what you *wanted* it to do is, of course, an entirely different matter!). Generally speaking, for most simple Unix commands, you can assume that the absence of an error message means that something has worked. And of course, if you get an error or warning message back from a command, it is crucially important that you read it, understand it, and act on it, rather than just ploughing on regardless. If you ignore errors and warnings, bad things happen. This is true in the Unix command world, and probably isn't a bad philosophy for life in general either.

Anyway, back to the exercise and some practice of manipulating files and directories. In previous labs you've already encountered three directories of particular interest:

- The **root directory** (/) is the top level of the **file system**.
- Your **current working directory** which is the one you are 'in' at the moment (and is shown by the output from `pwd`). This can also be referenced using a single dot (`.`), as in when starting up Quake Arena using `./ioquake3.arm` in your first Pi lab.  `pwd`
- Your **home directory** (~) which is the top level of your own filestore and where `cd` (**change directory**) with no arguments will take you. The value of this is also available as `$HOME`, so the following all have the same effect:  `cd`

```
$ cd
```

or

```
$ cd ~
```

or

```
$ cd $HOME
```

pour revenir
sur home directory

And no matter what is your current working directory, you can list the files in your home directory with either of these commands.


```
$ ls ~
```

or

```
$ ls $HOME
```

liste les fichiers

You should recall the difference between an **absolute path** (one that starts with a `/`) and a **relative path** (one that does not start with `/` but is instead 'found' relatively by starting from the current working directory). You've met the 'double dot' notation (`..`) to mean 'go up one level', as in `ls ..` or perhaps more often `cd ..`, or even `cd ../x/y` and so forth.

Speaking of `ls`, you will from time to time need to use its `-a` switch argument to ask it to show 'hidden' files beginning with a dot, and/or `-l` (a letter `l`, not a digit `1`) to make it show details about the files it lists.  `ls`

4.2.1 Creating a directory structure

Use the `mkdir` (**make directory**) command to create some directories. Type:

⚙️ `mkdir`

```
$ cd  
$ mkdir INTRO
```

→ créer des dossiers

and check that this directory has indeed appeared using `ls`.

It's important that **directories we ask you to make for your work have exactly the names we specify**: Unix will let you use any names you like, but so that lab staff know where to look for work when you get marked, and so that the system you'll be using to submit your work knows what you're submitting, it's very important that you follow these conventions for your lab work. Any files and directories you create for your own purposes outside of lab work can of course be named and organised however you like.

If you made a mistake, e.g. `intro` instead of `INTRO`, you **can remove** the directory *while it is still empty* with the `rmdir` command: e.g.

⚙️ `rmdir`

```
$ rmdir intro
```

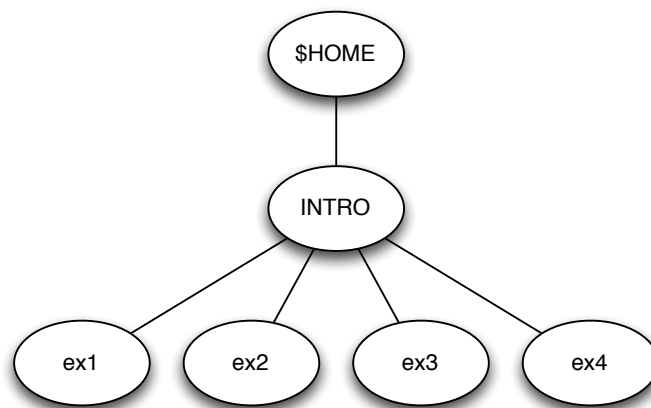
And then try to make it correctly.

Now go into your `INTRO` directory and let's make some directories for four imaginary **INTRO exercises**.

```
$ mkdir ex1 ex2 ex3 ex4
```

Now return to your home directory.

Your directory structure should now look something like this:



The **easiest way to check this is to use `ls` from your home directory with the `-R` flag**. This shows the **whole tree below your current working directory** (as with other commands we've encountered before such as `chmod`, **here the `-R` is short for recursively**—if you've not looked up what this means yet, now is a good time to do that).

⚙️ `chmod`

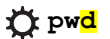
```
$ ls -R
```

→ affiche tout

Next **make a directory in your home directory for the COMP16121 course unit with sub-directories for each of the 10 exercises associated with that course** (these aren't imaginary exercises, you'll be starting on them next week). You *must* use the same convention as above: capital `COMP16121` and lower case `ex1` and so on.

You **must get your directory structure right before continuing**—be extremely careful that you get this right.

Now **try the following sequence of `cd`s**, checking where you are by running `pwd` **after each one**, and make sure you understand what is going on: if you're at all unsure about what has happened, please grab a member of lab staff to get an explanation—it really will save you a lot of hassle in the long run.



```
cd
cd INTRO/ex1
cd ..
cd ex2
cd ../ex1
cd ../../INTRO/ex2
cd ../../..
```

4.2.2 Copying, moving, and removing files

This subsection re-introduces three commands used for copying, moving and removing files. We'll first describe each command and then you'll get an opportunity to practise using them.

Copying files: `cp`

The `cp` (copy) command has two forms.



The first general form is

```
$ cp [FILENAME] [FILENAME]
```

For example

```
$ cp file1 file2
```

makes a copy of the file `file1` and calls it `file2`. If a file called `file2` already exists, the existing `file2` will be overwritten with a copy of `file1` and lost without warning. Using the `-i` option to `cp` will **ask you if this is** about to occur.

The second form is slightly different:

```
$ cp [FILENAME(S)] [DIRECTORYNAME]
```

For example

```
$ cp file1 file2 file3 dirname
```

This **copies the files `file1`, `file2` and `file3` into the directory `dirname`**, again overwriting any files already there with the same names.

Removing/deleting files: `rm`

The command `rm` (remove) is used to delete files.




```
$ rm [FILENAME(S)]
```

throws away the specified files. **Always take great care when using `rm`: unlike putting things in the 'trash' or 'recycle bin' in a desktop environment, the effects of `rm` are not reversible, and you don't get any warning before files are deleted forever.** Like `cp`, `rm` has a `-i` option which asks you if you really mean it. This option can be particularly useful if you are using wildcards in your command line arguments.

Breakout 4.1: Taking out the trash

You now know enough about the behaviour of the filesystem to know what's actually going on when you put files in the 'recycle bin' or 'trash can' in a desktop environment such as you get with OS X, Windows or Gnome. When you 'move to trash' in these environments, you're not deleting the file but instead using an equivalent of the `mv` command to move the file from its current location into a special directory that represents the trash can. When you tell the graphical environment to 'empty trash', you're actually invoking something equivalent to `rm`, which actually does delete the file from the filesystem.

Moving / renaming files: mv

The `mv` (move) command is similar to `cp`, but it just moves the files rather than makes copies. Again  `mv` we have the two forms

```
$ mv [FILENAME] [FILENAME]
```

and

```
$ mv [FILENAME(S)] [DIRECTORYNAME]
```

The effect is similar to copying followed by removing the sources of the copy, except it is more efficient than that (most of the time). For example

```
$ mv file1 file2
```

is like doing

```
$ cp file1 file2
```

```
$ rm file1
```

and

```
$ mv file1 file2 file3 dirname
```

is like doing

```
$ cp file1 file2 file3 dirname
```

```
$ rm file1 file2 file3
```

Using `mv` will preserve the timestamps on files, whereas the combination of `cp` and `rm` will not, since new files are being created.

4.2.3 Practice makes perfect

Now for some practice. Go to your home directory by typing:

```
$ cd
```

and copy the file called `fortunes` in the `/usr/share/games/fortune` directory to your current working directory, by typing

Breakout 4.2: fortune



At the moment we're just going to be using the fortunes file as something to copy and move around, so its contents are not important, but it's one of the source files used by the Unix **fortune**^W command, which we will be playing with later.

fortune^W is a simple program that displays a random message from a database of (supposedly) humorous quotations, such as those that can be found in the US in **fortune cookies**^W (hence the name). It also contains jokes (of a sort!) and bits of poetry.

fichier *destination*

```
$ cp /usr/share/games/fortune/fortunes .
```

Note that the **dot (meaning, of course, your current directory) is essential**. If you now do an `ls`, you should see that the file called `fortunes` has appeared in your directory:

```
$ ls
```

If no file called `fortunes` has appeared, the following will probably provide an explanation. If it did appear, read this anyway, just to check that you understand what you did right.

The **cp command needs at least two arguments**. In this case, the **file you are copying** is `/usr/share/games/fortunes`, and the **directory you are copying it to** is `'.'` (that is, your current working directory; remember every directory has a reference to itself within it, called `'.'`) If you missed out the dot, or mis-spelt `/usr/share/games/fortunes`, or missed out one of the spaces, it won't have worked. In particular, you may well have got an error message like:

```
cp: missing destination file
Try 'cp --help' for more information.
```

or

```
cp: /usr/share/games/fortunes/frotunes: No such file or directory
```

If you get the first message, it means you used the command with the wrong number of arguments, and nothing will have happened. The other is an example of what you might see if you mistype the first argument. If you do get an error message you need to give the command again, correctly, to copy the fortunes file across.

You should now have a copy of the file in your home directory. **You'll have to get into the habit of not having all your files in your home directory**, otherwise you will quickly have an **enormous list of stuff that will take you ages to find anything in**. The use of subdirectories provides a solution to this problem, which is why you created some earlier. **Moving this file to the 'correct' place gives you a chance to practise the mv command**.

Move the file **fortunes to your INTRO/ex4 directory**.

Now go to your `INTRO/ex4` directory and check that the file has appeared there.

To make sure you understand `cp`, `mv`, and `rm`, go through the following sequence (in your `INTRO/ex4` directory), checking the result by looking at the output from `ls` at each stage:

```
$ cp fortunes fortune1
$ ls
$ cp fortunes fortune2
$ ls
```

```
$ mv fortune1 fortune3
$ ls
$ cp fortune3 fortune4
$ ls
$ rm fortune2
$ ls
$ rm fortune1
$ ls
```

You'll notice that `rm fortune1` behaves differently to `rm fortune2`; if you can't figure out why, ask a member of lab staff for help.

4.2.4 Wild cards

An **asterisk** (commonly referred to as **star**) in a filename is a **wild card** which matches any sequence of zero or more characters, so for instance, if you were to type (don't actually do it!)

```
$ rm *fred*
```

then all files in the current directory whose names contain the string 'fred' would be removed.

Try the effect of

```
$ ls fortune*
```

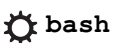
and

```
$ ls *tun*
```

Now try

```
$ echo *tun*
```

Our previous use of `*` has always been in conjunction with `ls` which might have led you to think that the wild card was being expanded by `ls`. In fact the expansion is done by the *shell*, `bash`, which means that the effect is true for anything you type on the command line. Wildcards are a very powerful and useful feature of the command line, and as with anything powerful and useful can be used or mis-used, so it's important that you know what you're doing with them.



One place where you must take care with wild cards is the **dotfiles**—these are files whose names begin with a dot (`.`), because the asterisk will not match a `.` at the start of a file name. To see what this means try the following

```
$ cd
$ ls *bash*
```

and

```
$ ls .*bash*
```

and see the different output.

4.2.5 Quotas

The command

```
$ quota
```

shows you what **your file store quota is**, and **how much of it you are actually using**. This is only of academic interest now, but **may become very important later in the year!** You may find that **you are unable to save files if you use more than your quota of file store**. It is important that, if this happens, **you do something about it immediately**.

4.2.6 Putting commands together

Redirection

Before you forget that you're in your home directory, **change back to your INTRO/ex4 directory**.

One of the simplest (and most useful) of Unix commands is `cat`. This command has many uses, one of which is to concatenate a list of files given as arguments and display their contents on the screen. For example:



```
$ cat file1 file2 file3
```

would display the contents of the three files `file1`, `file2` and `file3`. The output from `cat` goes to what is known as the **standard output** (in this case the screen).

If you type:

```
$ cat
```

nothing will happen because you haven't given a file to `cat`. When run like this, it takes its data from the **standard input**—which in this case is the keyboard—and copies it to the standard output. **Anything that you now type will be taken as input by `cat`**, and will be output when each line of the input is complete. In Unix, end of input is signalled by `<ctrl>d`. (Recall that typing `<ctrl>d` in your login shell will log you out—you have told the shell to expect no more input). So, after typing `cat` above, if you type:

```
The cat
sat
on the
mat
```

and then press `<ctrl>d` you will see the input replicated on the output (interleaved line by line with the input). The first copy is the 'echo' of what you typed as you typed it, the second copy is output from `cat`. This may not seem very useful, and you wouldn't actually use it just like that, but it illustrates the point that `cat` takes its input and copies it to its output. Using this basic idea we can do various things to change where the input comes from and where the output goes.

```
$ cat > fred1
```

will cause the standard output to be directed to the file `fred1` in the working directory (the input still comes from the keyboard and will need a `<ctrl>d` to terminate it). **Try creating a file `fred1` using this technique, and then check its contents**.

```
$ cat < fred1
```

will take the standard input from the file `fred1` in the working directory and make it appear on the screen. This has exactly the same effect as

```
$ cat fred1
```

You can, of course, use `<` and `>` together, as in

```
$ cat < fred1 > fred2
```

which will copy the contents of the first file to the second. Try this and check the results.

We can, of course, do this type of redirection with other commands. For example, if we want to save the result of listing a directory's contents into a file we just type something like:

```
$ ls -l > fred1
```

(this overwrites the previous contents of `fred1` without warning, so be careful of this kind of use).

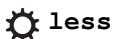
In the previous Intro lab session we met the idea of a **pipe**, using the `|` symbol to connect the **standard output** of one command to be **pip**ed to the **standard input** of a second.

We can construct another (admittedly rather artificial) pipeline example using just `cat`:

```
$ cat < fred1 | cat > fred2
```

The first `cat` takes its input from the file `fred1` and sends its output into the pipe. The second `cat` takes its input from the pipe (i.e. the output from the first `cat`) and sends its output to the file `fred2`. (How many other ways can you think of to do this?) This isn't a very sensible thing to do, but it does illustrate the principle of piping, and more realistic examples will appear in the exercises.

Standard output sent to the screen may come so fast that it disappears off the top before you have had a chance to read it. There is a simple way around this problem by piping the output into the command `less` which arranges to stop after each pageful (or screenful, or window-ful) of output. For example,



```
$ ls -la | less
```

would be a wise precaution if the current working directory held more than a screenful of entries. When `less` has shown you the first screenful, press the space bar to see the next screenful, or return to see just the next line. Use `q` as usual to quit.

Now would be a good time to remove all those junk files like `fred1` etc.

Before we leave the subject of pipes we meet two of the less obviously useful Unix commands, `fortune` and `cowsay`. We met `fortune` briefly in Breakbox 4.2, try running it a few times now. (Hope you didn't type the command more than once. If you did, think how that could have been avoided.)



Now try running the command `cowsay`. Nothing happens, because the cow is waiting for you to tell it what to say. Type anything you like and then `<ctrl>d` to denote the end of input. The cow should then utter your words:



```
< Hello World >
```

```

      ^__^
      (oo)\_______
      (_____)\/    )\ /\
          ||----w |
          ||     ||

```

Now try putting `fortune` and `cowsay` together to get the cow to 'speak' the fortunes. Utterly useless but it illustrates the use of pipes.

Breakout 4.3: Scripting versus Programming



You may be wondering what the difference is between a ‘script’ and a ‘program’, or between the idea of ‘scripting languages’ or ‘programming languages’. It’s quite difficult to pin down exact meanings for these, since their use has shifted over time and different people use the terms to mean subtly different things. Scripting languages and programming languages both allow people to create sequences of instructions for a computer to execute. Generally speaking when people refer to scripts or scripting languages they are referring to mechanisms for automating tasks rather than for performing complex computations. So if you wrote something that once a month deleted any files that ended with `.bak` from your filestore, you would probably use a scripting language, and most likely think of it as a script. If you were to write the next 3D blockbuster console game to outsell *Grand Theft Auto V*, you’d probably use a programming language and think of it as a program. At the extreme ends of the spectrum, the distinction is quite clear; in the middle it gets a bit muddy.

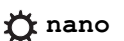
4.2.7 Making your own commands

Pretty much anything that you can type at the command line can also be stored in a file to create a simple program called a **shell script**, so if you find yourself frequently connecting together simple commands to perform a particular task, you might find it useful to create a script for use in future, rather than retyping everything each time. If you recall back to Section 2.3.1 we made a simple command called `mankyweather` using an **alias**. Aliases are fine for things that you can express in a single line of text, but clumsy for more complex combinations of commands; a shell script instead allows you to use as many lines as you like.

Use an editor to create a **shell script** in the file `~/bin/wisecow`. Make a directory in your home directory called `bin` using the command:

```
$ mkdir ~/bin
```

and in that directory create a file called `wisecow`. You’re welcome to use whatever text editor you like for this, but you might find that for short edits like this you’re better off using something like **nano** rather than one of the more heavyweight graphical editors.



Put the following text into the file:

```
#!/bin/bash  
  
fortune | cowsay
```

The first line tells the operating system to use the program `/bin/bash` when this script is run, i.e. it will start `bash` with an argument telling it to get its commands from this file and execute them pretty much as though they had been typed into `bash` in the usual way.

Now try to run your new program:

```
$ ~/bin/wisecow
```

Oops, that won’t have worked! Before the operating system will believe us that this really is a thing that we can run, we need to give the file **execute permission**. Use `ls` to see what permissions the file has at the moment. To make it **executable** we use `chmod` as follows.

```
$ chmod +x ~/bin/wisecow
```

Now check its permissions again. If all is okay, you should be able to run this time with

```
$ ~/bin/wisecow
```

and your wise cow should have spoken.

Now here is the really cool bit: in an earlier lab you met the idea of `$PATH`—the list of all places where the operating system will search when you type a command without specifying its full pathname. See the value of this now:

```
$ echo $PATH
```

Notice one of the directories listed is your very own `~/bin` directory: this is where you can put *your own* commands.

So, now type just

```
$ wisecow
```

and bask in the wisdom of your newly created cow guru.

4.2.8 Doing several things at a time: background running

The command `xclock` fires up a clock displaying the current time; try it now.

You will notice that there's a problem with running clocks from a terminal window. If you type:



```
$ xclock
```

a clock does indeed appear. However, you then can't type any more commands in the terminal window, because it is waiting for the clock program to finish, and of course the clock program is designed to run forever (or at least until you logout). In this situation you can, of course quit the clock, using `<ctrl>c` in the terminal window. However, if you want to keep the clock running and still get on with other things in the same terminal window you can just add an ampersand (`&`) on the end of a command, so in this case,

```
$ xclock &
```

Adding the ampersand ensures that the program is run in the background, that is, separately from the terminal window. This way you can have several clocks at once and continue typing other commands carrying on from where you were.

Try running an `xclock` in the background now.

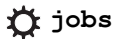
Work out from the manual page for `xclock` how to use its various options, and experiment with producing a range of clocks. (You should ignore the section of the manual page entitled `X DEFAULTS` – for now.)

One situation where the use of background processes is particularly useful is when you fire up an editor, such as `gedit`. You can start the editor and still continue to do other things in the same terminal window. Try this now

```
$ gedit &
```

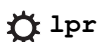
You can also start `gedit` (and many other editors) with a filename argument to edit a particular file; in fact this is probably the way you will normally use it.

Sometimes you might forget the `&` and want to put a process into the background while it is already running. This is easy to do; first you **suspend** the process by typing `<ctrl>z`, then type `bg` to put into the background. Try this now with a `gedit` window. You can bring a background process back into the foreground by typing `fg`. If you have more than one suspended or background job then you just add the background process count as an argument; you can find which process has which count by using the shell command `jobs`.



4.2.9 Printing text files: `lpr`

The command `lpr` can be used to send files to a printer. In its simplest form, you simply run:

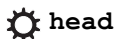


```
$ lpr file1 file2
```

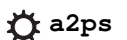
to print the given files. The printing service we use is the University **Pull printing** service, which allows you to collect your printing at *any* University pull printer. This is described in more detail at

<http://www.itservices.manchester.ac.uk/students/printing/>

You could use `lpr` now to print out the `fortunes` file, but that file is quite big and we don't want to waste a lot of paper. So please don't! However, it would be nice to practise using `lpr`. So instead print out just the first 50 lines of it. Look at the man page for `lpr` and discover what it does if no file names are given. Now look at the man page for the command `head` and figure out how to make it output the first 50 lines of the file `fortunes`. Experiment with this to make the 50 lines appear on your screen. Now send the 50 lines to the printer—without using a temporary file. Go and collect your print output from a nearby printer (there are printers in SSO, G23, and other places).



`lpr` is a basic printing tool for printing text (and it is also clever enough to print most types of images nowadays). A more sophisticated printing program is `a2ps`. This produces a nicer output, and it can recognise different types of text file—including program source code files—and present the various keywords of the programming language and the program identifiers in different fonts to help with readability.



Look at the man page for `a2ps` and use `a2ps` to print the file

```
/opt/info/courses/INTRO/SimpleJavaProgram.java
```

When you collect this print output you will see that `a2ps` has formatted it nicely. This is a good way to obtain printouts of your own work, should you wish to.

All students have a printing account which is used to 'pay' for their printing. The School has pre-credited your account with enough credit for you to print all the material needed for your courses (with some to spare) without having to pay for anything. For administrative reasons, your initial credit will be £4, and this will be topped-up during the year. The first top-up is likely to occur during Reading Week.

The printing allowance we give you means that you will be able to to print 500 sides during the year if you use the double-sided A4 mono option (at 8p per page). Note: the allowance would only cover 400 sides if you use the A4 single-sided option, at 5p per page, so clearly printing double-sided is your better option.

4.2.10 Time for a checkup



Check your setup. Now it's time to check that your CS account's environment is set up properly. In the earlier introductory labs we got you to make various changes to the configuration files that determine how your account behaves (the so-called 'dotfiles').

From a terminal, run the command


```
/opt/teaching/bin/check-my-setup
```

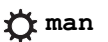
If you skipped any of the steps in the intro labs, or didn't quite get things right, then this script will help you detect any configuration issues that might bite you later, and will help you fix any problems it finds (if you're curious about the script works, use `less` to check out its contents).

If you don't understand what the script is telling you to do, please do take this opportunity to find a member of lab staff to explain things.

Once you've fixed any problems that the script has identified, please re-run the script to check everything is now okay, and then repeat this process until the script reports that everything is 'good'.

4.2.11 Exercises

Here are a number of exercises to experiment with. Use `man` to find full details of the relevant commands you need to use. When you have your answers, email them to your personal tutor. Just send a single email with everything in. If you can't complete all the exercises, no problem, just send what you've done.



1. As you know, `ls -l` gives you extra information about files. Skim through the man page for `ls` to see what it means. Check the ownership and permissions of your own files. For more about ownership and permissions, look at the manual pages for the `chown` and `chmod` commands.

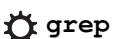
Question: Why don't you own '.' in your home directory?

2. Look at the man entry for `rm` and find out what would happen if you did `cd` and then `rm -rf *`. **WARNING! DO NOT ACTUALLY TRY THIS!** We once had a system administrator who, after logging in as the **superuser** (that's a special user called **root** that has the permission to do *anything*), typed the above command by accident. What do you think happened? (Hint: on many Unix systems, the superuser's home directory is `/`).



Question: What would it do in your home directory? What would it do if the superuser made this error?

3. Another useful command is `grep`, which displays all lines of a file containing a particular string (in fact, the string can be a pattern with wild-cards and various other things in). The form for a simple use of `grep` is



```
grep [PATTERN] [FILENAME(S)]
```

This will result in a display of all the lines in the files which contain the given pattern.

A useful file to use for experiments with `grep` is `/usr/share/dict/words`, which is a spelling dictionary. Try to find all words in the dictionary which contain the string 'red'.

Question: what was the command you used to do this? (Please don't email your tutor the results from the command!)

4. Use a suitable pipeline to find out how many words in the dictionary contain the string 'red' but not the string 'fred'. (Hint: The answer to the previous question gives all the words containing 'red', look at the manual page for `grep` to find out how to exclude those words containing 'fred'. The `wc` (short for 'word count') program counts words (amongst other things). Use pipes to put them all together.)



Question: what was the command you used to do this? How many words did you find?

You have now finished the lab, but we encourage you to try the exercises contained in the file `/opt/info/courses/INTRO/extras`. Don't worry if you find them tricky—they are.

4.2.12 The end of the beginning

That's it! You've now reached the end of the introductory labs.

What follows next week in your various courses are labs and coursework that will actually form part of your assessment, and ultimately, your Degree.

If you're new to Unix/Linux it's a good idea to spend a bit of time reflecting on and re-reading through what we've already done in order to ready yourself. Make sure you feel comfortable with the concepts we've presented. If something seems a little weird, or hard to follow, spend a bit of time and go over that bit of the lab script again. Remember, as well as your scheduled class time in the University, you're expected to also do some work in your private study time at home.

And most importantly, if you're feeling overwhelmed or even just a little concerned about your understanding of what you've met so far, don't worry. Tell us and we'll help. Please don't suffer in silence—speak to your personal tutor, or email one or all the team (see below) that created these labs. We're here to help, and we're very happy to do that. But before we can, you need to tell us!

Finally, we want to wish you the best of luck, not just for the coming weeks, but for the coming semesters and years, in everything you do here in the School.

Steve, Graham, Toby and John.



steve.pettifer@manchester.ac.uk



graham.gough@manchester.ac.uk



toby.howard@manchester.ac.uk



john.latham@manchester.ac.uk

4.3 Acknowledgements

These notes are largely our own work, but have been inspired in places by previous lab exercises created by Pete Jinks, John Sargeant and Ian Watson. We're very grateful to Paul Waring, Alex Day, Alex Constantin, Hamza Mahmud and Ben Mulpeter for test driving and debugging the exercises.