MSc Introductory Laboratory

Autumn 2016

1 Computing Infrastructure

Hello and welcome. In this introductory, lab we're going to cover some of the basic things you'll need to know about the IT infrastructure here in the School of Computer Science. Some of what we tell you may well seem very obvious to you, and if that's the case we ask you to be patient. Some other things might not be so obvious...

These notes are a modified version of some that we use in our first year Undergraduate programme. We have tailored them for PGT students, but there may be some places where they betray their origins.

In this and every lab there will be staff, consisting of academic staff and postgraduate students, around to help you. If you're stuck or find something that you really can't understand, then *please ask for help*; that's what the lab staff are here for, don't just sit there getting frustrated. The postgraduate students are known as **Teaching Assistants**, or **TAs** for short, although up to 2014 they were referred to as **Demonstrators**. We will try to use the new name, but inevitably both will be used for a while.

2 About these notes

2.1 Breakout boxes

Scattered throughout the main text of these exercises there are information boxes of various kinds:

Danger! The bomb icon explains problems and pitfalls, and how to avoid them. It's really important that you read these sections, or you may regret it later.

We digress... Boxes marked with this icon contain digressions and facts that are hopefully interesting but are probably tangential to the main flow of the exercise.

Stop here... Boxes marked with this icon contain checkpoint activities that you should complete before proceeding further.

2.2 Styles and conventions

We'll be using various typographic styles to indicate different things throughout these notes. When we introduce new concepts, or use terms that have a different meaning in computer science to their everyday use, they will appear **like this**, and will be explained in the nearby text. Things that are either the input or output to commands will appear `in this font`. And many words or phrases will have a little 'w' after them, meaning that if you click on the term (if you're reading the notes online) or type the phrase into **Wikipedia**[w] (if you're reading this

on paper), you should be taken to a sensible definition in case you want to learn more about something that we've touched on here.

Where you see text in square brackets [LIKE THIS] it will mean that when you're typing things you should replace the text with something that the text describes; for example [NAME OF THIS CITY], would become Manchester.

3 The CS labs setup

All the desktop PCs in the labs in the Kilburn building are 'dual-boot': they can be started up running either Linux or Windows. This is for flexibility – the labs for most course units use Linux, but some use Windows, and of course, outside the labs, you're free to use whichever you prefer for any aspect of your studies that does not require use of a particular operating system. If you're not familiar with Linux, don't worry. We'll be telling you a bit about it in this lab, and you'll be looking at it in much more detail in subsequent labs.

4 Using Windows

We'll start with Windows, so the first thing we need to do is to make sure your PC is running Windows. Have a look. If it is, and the screen looks like Figure 1, then you will not need to reboot, but please read the next bit anyway, because at some point you may need to reboot from Linux to Windows.

If your PC is currently in Linux, showing a black screen with a login prompt, you need to reboot the PC. To do this, press `ctrl`, `alt` and `delete`.

This will probably cause strange messages to appear on the screen, disappear, and be replaced by yet more strange messages. Now look at Figure 2: when you see something like that on the screen press `space` you will need to respond. Meanwhile, be patient, watch it all happen, and don't worry what it all means.

After a while, everything should settle down and the screen will look like Figure 2; press `space` now!

This is where you decide whether to start up Windows or Linux. *If you do nothing when this screen appears, the computer automatically boots into the default operating system after a fixed timeout period. By pressing the space bar (any other key would have done) you have stopped this timeout process.*

Now use the up/down arrow keys on the keyboard to highlight the line reading `Windows 7`, and press the `enter` key. After a short while, the Windows 7 welcome screen will appear, as shown in Figure 1. Now use `ctrl-alt-del` again as instructed and you should see the Windows 7 login screen, as shown in Figure 3.

To log in to your personal account, type in your username (this is an 8-character name usually starting with an 'm' that you were given at registration). Your password will be the one you set at registration. If your password doesn't work, or if you've forgotten it, you need to fix this urgently. You can login on a machine running Windows, with the username `register` and password `register`, then follow the instructions.

If you forget your password later on, you can also use one of the following methods:

- If you have access to a web browser, use the password recovery page at `https://iam.manchester.ac.uk/recover`

Figure 1
The Windows7 Welcome screen.

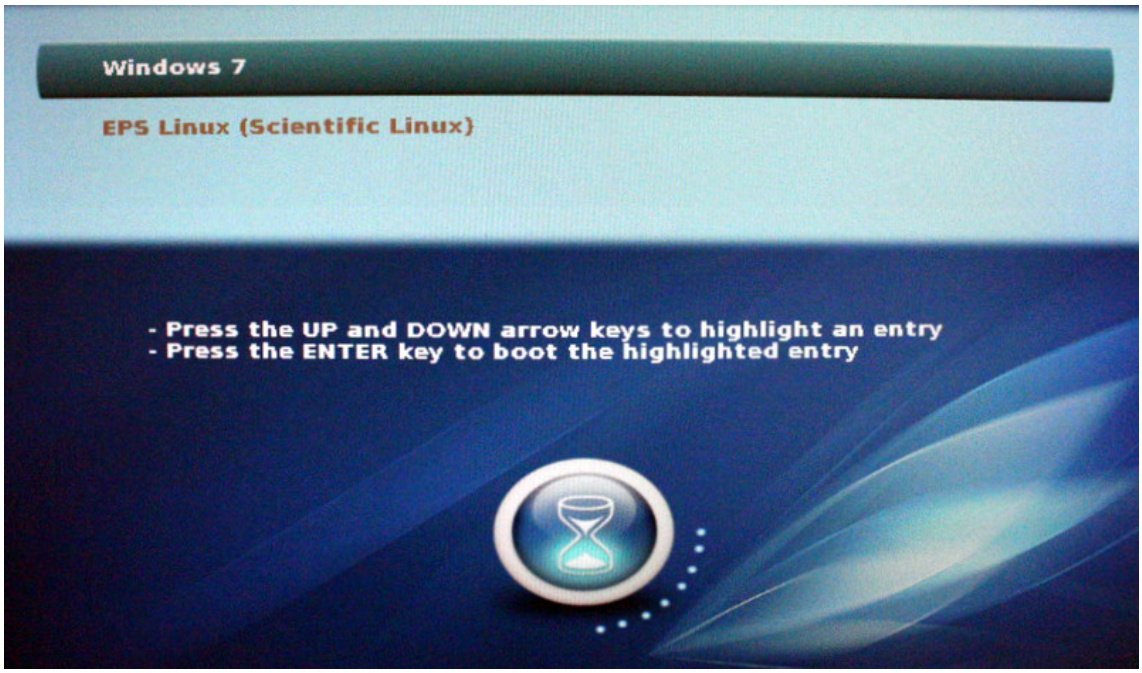


Figure 2
The boot selection menu screen.

Figure 3
The Windows 7 login screen.

- You can also contact the University IT helpdesk (opening hours are Monday to Friday 9am to 5pm) by phone on 0161 306 5544, or dial 65544 from a University internal phone; or you can visit a helpdesk either in the Main Library (Building 55 on the Campus Map), or in the Alan Gilbert Learning Commons (building 63).
- Remember, if you need help, ask!

Once you're logged in, go to your My Manchester page in a web browser, at

`https://my.manchester.ac.uk`

You will have to authenticate again to access this page, which should look something like Figure 4.

5 Reading your email

We use email extensively in the School, so it's vitally important that you read your mail regularly—we work on the assumption that you'll be checking your School email at least once a day (and probably much more often). On the My Manchester page, follow the mail link (indicated by the hand drawn arrow in Figure 4) to access your email on the University's Office365 system. This is a system that gives you 25GB of email storage space and an integrated calendar. You should see a page looking something like Figure 5.

Office365, like most modern email systems, supports the IMAP protocol – which means that you can access your mail from any device that can run an IMAP mail client. Some examples of such clients are: Mozilla Thunderbird, Mac Mail, Windows Live Mail and mail apps on mobile devices. No matter what client you use, you need to tell it the appropriate settings. The mechanism for finding these setting can be found in our CS student IT wiki, which is located on the

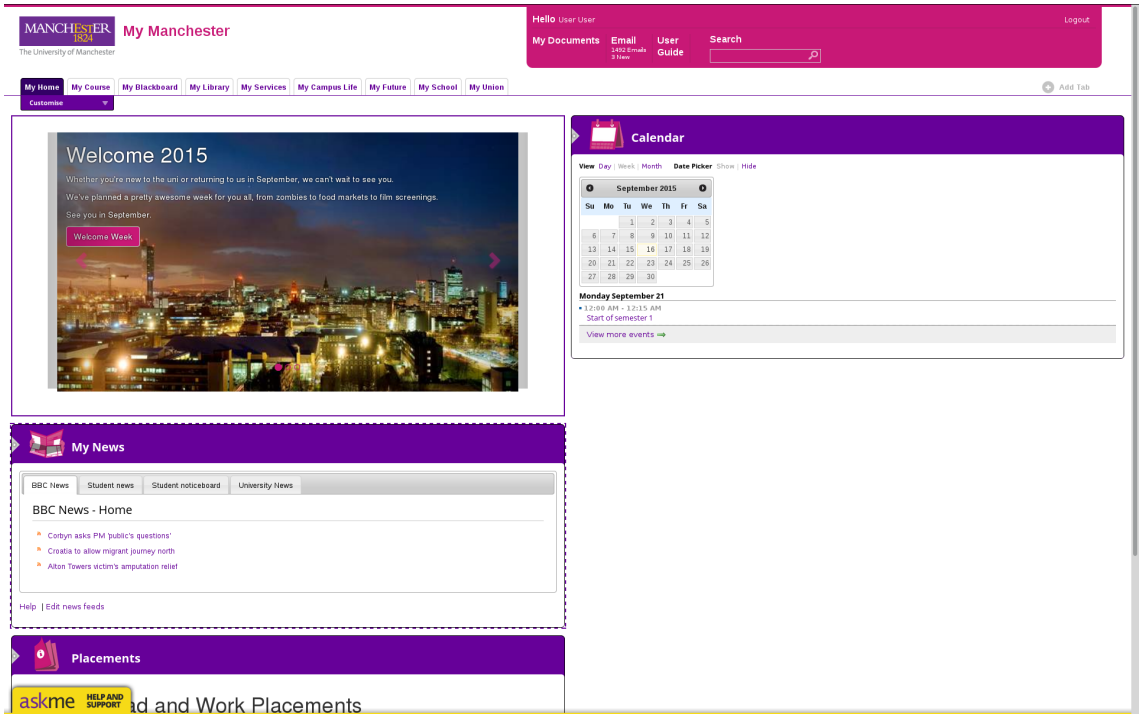


Figure 4
Your MyManchester page.

web at `wiki.cs.manchester.ac.uk/index.php/StudentFAQ/IT`. Look for the answer to the question 'How do I configure my favourite IMAP mail client?'. This wiki provides help about the IT infrastructure within the School; it is one of several School FAQ lists, which can be found at `wiki.cs.manchester.ac.uk/index.php/StudentFAQ`. Please make use of these valuable sources of information.

Once you've found the settings, use Office365 to email this information to yourself, to your University account, which is usually of the form:

`firstname.lastname@student.manchester.ac.uk`

You should also write this information down somewhere you can find it, as you will need it PG later when you may not have access to your email.

You'll need this information later so please don't delete this email after you've read it.

Finally, if you use an IMAP mail client on your phone or mobile device, you can configure it to use the settings you've just found, and check that you can read and send email successfully. It's probably best to try this later as the mobile signal in the lab is not good.

That's all on Windows for now, but of course you're free to boot an available PC into Windows and use it at any time unless you are in a lab that requires the use of Linux.

6 Rebooting into Linux

So let's reboot the PC, and start it up in Linux. First, log out of Windows by selecting the Windows icon in the bottom left of the screen and then `Log off`. Get back to the login screen, click the small icon in the lower right of the screen (see Figure 6) and select `Restart`.

The system will shut down and after a while we'll be back to the boot selection menu screen

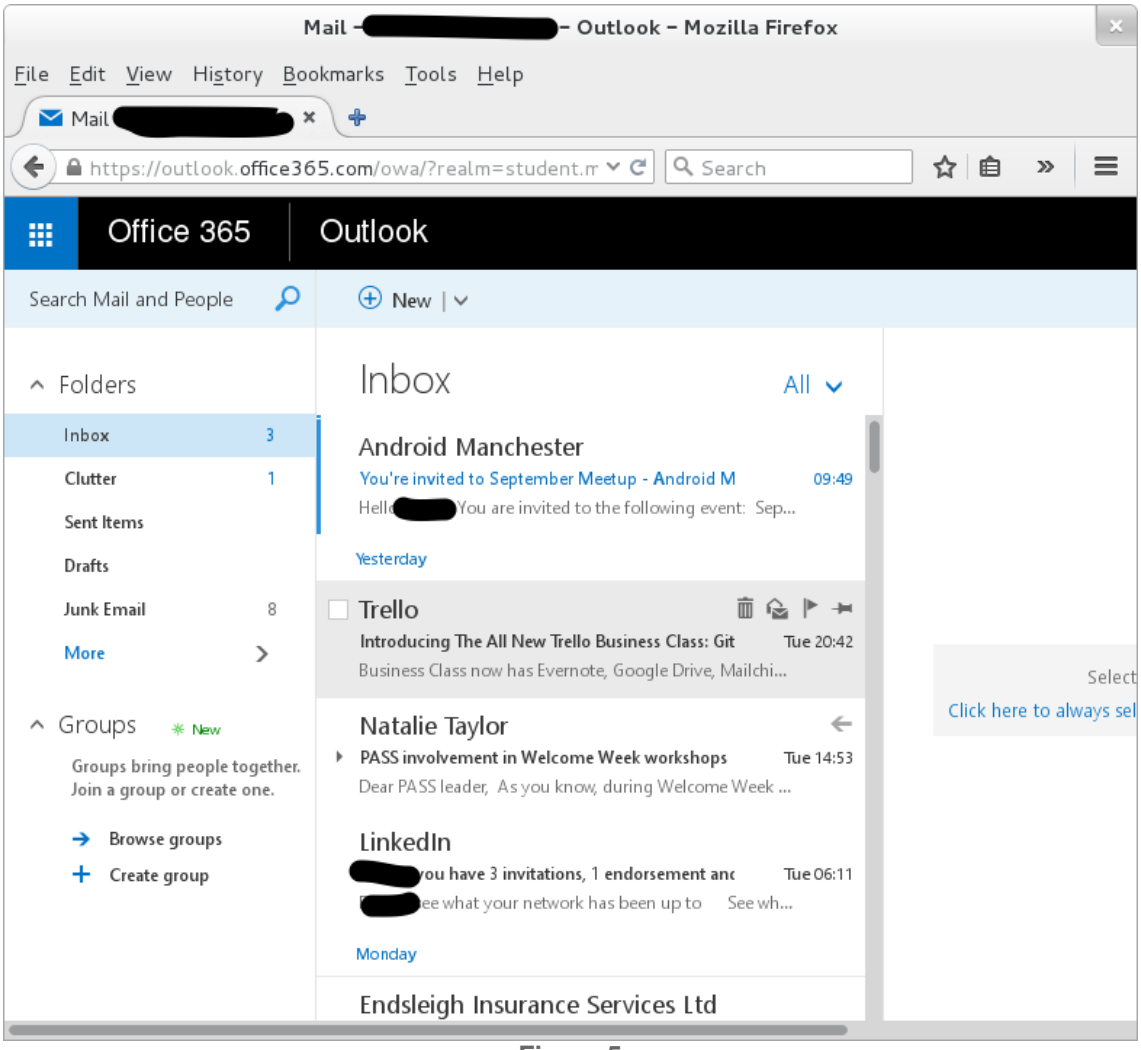


Figure 5
Your Office365 email.

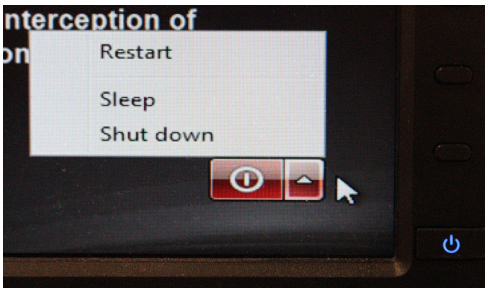


Figure 6
Restarting the PC.

we saw earlier in Figure 2. This time, press space immediately, then use the up/down arrow keys to select `EPS Linux (Scientific Linux)`. Linux will now start, and after a short while you should see a black screen, with a white login prompt.

7 Logging in

Make sure the desktop PC is booted into Linux, and log in using your University username and password . Remember that nothing will appear on the screen when you type your password. You should be greeted with a similar, but rather longer, command prompt to the one you saw in the previous lab.

You are now logged in to a PC that is part of our School Linux network. Type `pwd` to find out which directory you are in. It should be something like `/home/mbaXXXXX`, where the part after the `/home/` is your username. This is your home directory, which is not actually stored on the desktop PC but on a central fileserver. This means that, whichever machine you use in the lab, you will always see the same home filestore. **pwd**

The environment you are now in is known as **terminal mode**. This is a way of interacting with the computer via a screen containing only text, without the now familiar windows and images. All interaction is done using a **command line interface** (CLI), typing commands into a program known as a **shell**. When the terminal occupies the entire screen, as it does here, it is known as **console mode**. Later we will be using a graphical environment, but for now we will stick to terminal mode interaction and start off by reading mail.

8 Reading email in terminal mode

You're probably familiar with reading email using either a web-based interface, a graphical desktop application (such as Outlook, Thunderbird or OS X Mail) or using an app on a smartphone or tablet. Today you're going to do something slightly different, and configure a text-based mail client so that you can read your University email while using a terminal. The email client we're going to use is called `Mutt`, which is fairly simple to configure and straightforward to use (according to its author, Michael Elkins, "All mail clients suck. This one just sucks less"). There are plenty of other similarly lean text-based **email clients**[w], and you may at some point want to check out `Alpine` as a sensible alternative to `Mutt` or for the historically-curious, `Elm` (if you want a *really* hardcore terminal-mode experience of mail, look up **Mailx**[w]). **mutt**

First, let's confirm that `Mutt` is actually installed.

To see if `Mutt` is installed and is accessible to you, use the `which` command. Type: **which**

```
$ which mutt
```

This should respond with `/bin/mutt`, telling us that the `mutt` command has been put in the `/bin` directory on our system.

List the contents of `/bin` by typing

```
$ ls /bin
```

and notice that here we're using `ls` to look at the contents of a directory other than the one we're currently in by passing the directory name as an **argument**. A whole load of things should scroll past on the screen; most of them won't mean anything to you right now, but **ls**

don't worry, we'll look at some of the important ones soon enough. Now that's a lot of stuff to look through, and depending on the size of your screen the command we're looking for may have scrolled off the top. So let's try to narrow our results down a bit. Type:

```
$ ls /bin/ma*
```

and you should be given a much smaller list of things from the `/bin` directory; only those starting with the letters `ma`. The asterisk symbol is interpreted as being a 'wildcard' that stands for 'anything of any length, including length zero', so the command you've just typed means 'list the contents of the `/bin` directory, showing only files that start with the letters `ma` and then are followed by zero or more other characters' (notice that the `man` command that you used in the last session is there amongst the results).

You could narrow this down even further by typing `ls /bin/man*`, in which case you'll only get files from `/bin` that start with the letters `man`. Note that if you leave off the asterisk from your command, you'll be asking for files that are called *exactly* `ma` or `man`, which isn't what you want here.

So far we've been getting you to do a fair amount of typing, and now we have to admit that you've been typing a lot more than you actually need to (it's good practice though, so we're not feeling too guilty at this stage). The default Linux command line has a feature similar to autocomplete that you'll have seen on web forms and in graphical tools, that saves you typing full commands by suggesting possible alternatives.

Type `ls /` but don't hit `Enter`, and instead press the `Tab` key twice. You'll be shown a list of sensible things that could follow what you've typed—in this case it's the list of the contents of the system's root directory. Now type the letter `u` (so that the line you've typed so far should read `ls /u`) and hit `Tab` once. This time your command will be expanded automatically to `ls /usr/` since that's the only possible option. Press `Tab` twice now, and you'll get shown the contents of `/usr/`. Type `b`, and press `Tab` to expand the command to `/usr/bin/`, and then press `Enter` to execute the command.

The **autocomplete**[W] function you're using here is more commonly called **tab complete** by Unix users. If you press `Tab` once and there's exactly one possible option that would autocomplete what you've typed so far, then that option gets selected; if there are multiple possible things that could complete your command, then `Tab` will complete as far as it it can, then pressing `Tab` a second time shows you all of them, giving you the option to type another character or two to narrow down the list. Learning to use this will save you a lot of typing, because not only does it reduce the number of characters you type, it also helps you see the possibilities at the same time. Very usefully, it also saves you from making lots of typing mistakes.

Here are some other handy command line tricks for you to try out (give them each a go now so that you remember them for later):

- You can use the up and down arrow keys to cycle back and forth through the list of commands you've typed previously.
- The left and right arrows do what you expect, and move the insertion point (often referred to as the **cursor**) back and forth. Pressing `<ctrl>a` will move you to the start of the line, and `<ctrl>e` to the end of the line (much faster than moving backwards and forwards character-by-character).
- `<ctrl>c` aborts the current line, so if you've typed a line of gibberish, don't waste time deleting it one character at at time, just `<ctrl>c` it!

Breakout 1: File extensions

If you've mostly used Windows or OS X via a GUI, then you're probably used to files such as `cheese.jpg`, where you would interpret `cheese` as being the file *name* and *jpg* as being the file *extension*. Some operating systems—notably Windows—have the notion of a **filename extension**[W] of a particular number of characters built in; for example things ending with `exe`, `bat` or `com` mean that they are executable files. In Unix, a file extension is merely a convention that's not enforced or meaningful to the operating system. So although it's common to give files a suffix that makes it easy for a human to guess what kind of file it is, Unix itself just treats these as part of the file name. In fact, you can have multiple 'file extensions' in a name, to indicate a nesting of file types. In the previous lab the file `quake3.tar.gz` is a **tar** archive that has been **gzipped**, but the presence of the `.tar` and `.gz` parts are really just there to tell the user how to treat the file.

- Typing `history` lists all the commands you've typed in the recent past, useful if you've forgotten something.
- Pressing `<ctrl>r` allows you to retrieve a command from your history by typing part of the line (e.g. if you searched for '`whi`' now, it'll probably find the '`which mutt`' line you typed a while back). Pressing `<ctrl>r` again steps through possible matches (if there is more than one).
- Pressing `<ctrl>t` swaps the two characters before your cursor around. What, really? Yes: you'll be surprised how often you type characters in the wrong order!

Back to configuring your email client. Before we use `mutt`, we need to point it at the incoming and outgoing email servers, and we'll do this by creating a configuration file.

We've created a template file for you to get going with. Make sure you are in your home directory, then use the `curl` command as in the last lab session to fetch the template from
`https://studentnet.cs.manchester.ac.uk/ugt/COMP10120/files/mutt-template`

Remember, you're going to need to use a switch argument to tell `curl` what it should call the file it's fetched: call it anything you like, but `mutt-template` is a perfectly good name (if you're feeling uncomfortable about a file that doesn't have a file-extension, see Breakout 1 for more information). Let's look at the file to see what's in it. Type

curl

```
$ less mutt-template
```

and you should see the following written to the screen:

```
#
# mutt configuration
#
# Change the following three lines to match your
# University of Manchester account details
set my_user_name="firstname.secondname@student.manchester.ac.uk"
set my_imap_server_name=[SERVERNAME].outlook.com
set realname = "Real Name"

# Change the following line to a different editor if you prefer.
```

Breakout 2: Spaced out filenames

Because of its roots in the early days of computing long before the advent of graphical user interfaces, Unix filenames tend not to have spaces in them because this conflicts with the use of a space to separate out commands and their arguments. The Unix filesystem does allow spaces in filenames, but you'll have to use a technique called 'escaping' if you want to manipulate them from the command line; this involves prefixing spaces in filenames with the backslash character \ to tell the command line not to interpret what follows the space as a new argument. For example, a file called `my diary.txt` would be typed as `my\ diary.txt`. It's a bit ugly, but it works fine.

Breakout 3: Less is more

As we've mentioned before, many of Unix's commands are plays on words, puns, or jokes that seemed funny to the command's creator at the time. Though this gives Unix a rich historical background, it does rather obscure the purpose of some commands. A prime example of this is the `less` command, used to page through text files that are too large to fit on a single screen without scrolling.

Early versions of Unix included a command called `more`, written by Daniel Halbert from University of California, Berkeley in 1978, which would display a page's worth of text before prompting the user to press the space bar in order to see *more* of the file. A more sophisticated paging tool, called `less` on the jokey premise that 'less is more' was written by Mark Nudelman in the mid 1980s, and is now used in preference to `more` in most Unix systems, including Linux.

more

```
set editor = "nano"

################################################
### Shouldn't need to change any more from here on ##
################################################
set imap_user = $my_user_name
set from = $my_user_name
set folder = "imaps://$my_imap_server_name:993"
set spoolfile = "+INBOX"
set smtp_url = "smtp://$my_user_name@$my_imap_server_name:587"
```

less

man

The `less` command is used to display textual content from files and other sources (if you want to know why it has such an odd name, look at Breakout 3). One of `less`'s features is that it 'pages' through text, so that if the file you are looking at won't fit on one screen, pressing the space key will move you on to the next 'page'; you may notice that the `man` command you used in the previous lab session actually used `less` to display the manual pages.

Don't worry too much about the details of this file for now. We just need to edit the file to

contain your details rather than the fake ones in the template you've just downloaded. But let's play it safe: rather than editing the actual file you downloaded, just in case you make a mistake, let's first make a copy of the file in your home directory.

Quit `less` (using the same technique you used to quit the `man` command in the last lab session), and then enter

```
$ cp mutt-template mutt-template-copy
```

Did you type all of that? If so, you've wasted several precious key presses! You could have typed `cp mu`, and then pressed `Tab` to expand it to `cp mutt-template`, and then do the same thing again to create the start of the second argument, finally adding on the `-copy` bit yourself. It's a good habit to get into and will save you a lot of time over the next few years.

The basic form of the `cp` command takes two arguments, the first being the file you want to copy, and the second being the name of the file that will be created. Confirm that there is indeed a new file in your home directory using `ls`, and that its contents are what you expect using `less` (how would you find out what else the `cp` command could do?). **cp**

To modify the file, you'll need to use a text editor. Type

```
$ nano mutt-template-copy
```

to invoke the `nano` editor. Although fairly basic, the nano editor has all the features you'll need to make these changes, and helpfully shows you the various keyboard shortcuts to do particular things such as saving and quitting at the bottom of the screen (remember, the **caret symbol** (`^`) is shorthand for '`ctrl`', so `^X` means '`<ctrl>X`').

Now use it to make the following changes:

- Edit the line that starts `set my_user_name` to include your University email address.
- Edit the line that starts `set my_imap_server_name` to include the server name that you obtained from the Outlook client in My Manchester.
- Edit the line that starts `set realname` to include your real name, in whatever way you want it to appear in outgoing emails. Please use your proper name here and not a funny nickname.

When you've made the changes, write out the file to your filestore and quit back to the command line. Then use `less` to confirm that the file now looks exactly as you want it to.

Now, `mutt` expects the file containing its configuration information to have a particular name, and that's not `mutt-template-copy`, so we'll need to do something about that. The Unix `mv` command is used to rename files or directories (it's short for 'move'), so use that to change the name of the file to `.muttrc` by typing, not forgetting the dot at the start of the second filename **mv**

```
$ mv mutt-template-copy .muttrc
```

`mv` may seem like an odd name for a command that is used to rename a file, but it actually has a number of uses, including moving a file from one part of the file hierarchy to another. You'll see more examples of this in a later lab session.

Rather like `cp`, `mv` takes two arguments; but instead of making a copy of the file, `mv` just changes the name of the file given as the first argument to that of the second.

Type `ls` to confirm that the file name has changed as you'd expect.

Oh. But it's gone! Actually, no, it's still there, it's just hidden! There's a Unix convention that filenames starting with a full-stop symbol don't appear when you type `ls` in its basic form, because these are normally configuration files that you don't need to see on a day to day basis (the '**rc**' part of the `.muttrc` name stands for **resource configuration**, another Unix convention). So to see these files you'll need to add an extra switch argument to `ls`. Use the `man` command, with an appropriate argument, to find out what this switch is, and then use the switch to confirm that the `.muttrc` file does indeed exist.

Using this switch on `ls` will reveal several other so-called **dotfiles** that have been lurking in your home directory all along.

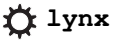
If you're confident that you now have a file called `.muttrc` containing the correct configuration, you can now type `mutt` to start the program.

It should be reasonably clear how you use `mutt` to send and receive email; if you get stuck there are plenty of online tutorials to help you out. Send yourself a test email to make sure that everything is working, and when you're confident you've mastered the basics of sending and reading using this tool, quit `mutt` to get back to the command line. One thing you should note is that `mutt` doesn't have its own editor for composing emails, so will use `nano` unless you change this to something else in the `.muttrc` file.

9 Browsing the Web

Although you will have experienced The Web so far as a highly graphical system, the technology that underpins it is for the most part text-based, and it is (just about!) possible to browse web pages using a terminal-mode application. It might seem like an odd thing to do, but there's an important point to be made here, so bear with us.

Try browsing the School's web pages using `lynx` by typing




```
$ lynx https://studentnet.cs.manchester.ac.uk
```

Rather like `mutt`, the `lynx` program has just about enough on-screen help for you to be able to browse around a little without any additional instructions from us. **You may find that when you follow some links, nothing very much appears to have happened; but scroll right down the page and you should see the content that you're looking for.**

You'll probably find using `lynx` an unsatisfying experience: tolerable, and probably okay in an emergency, but not how you'd ideally like to browse the web. And you might be wondering why we've even bothered to get you to try viewing the web through a text-only interface. Apart from the absence of images and videos etc., the main difference between using something like `lynx` and a regular browser such as Chrome, Firefox, Safari or Internet Explorer, is that you'll notice that web pages have been made into much more linear affairs than when they are rendered in a graphical environment. While you might expect to see the navigation links neatly arranged on the left or top of the page with the main content prominently displayed in the centre, seen through a purely textual interface it's all one big stream of stuff, and it's very hard to distinguish between the navigation links and the main content.

Now consider what the web 'looks' like if you are visually impaired or blind and have to use a screen-reader (a voice-synthesiser program that vocalises the text that's on-screen) to interact with your computer. Whereas a sighted person can easily cope with a two-dimensional layout that allows you to be aware of multiple things at the same time (i.e. you can be reading the

main content of the page, but conscious of the fact that there's a navigation bar on the left for when you need it), if instead you are listening to a voice reading the contents of the page out to you, it's only possible to be hearing one thing at a time. And what's more, you have to remember what has been read out in the past in order to make sense of what you are hearing now; you can't just 'flick back' a paragraph or two by moving your eyes, instead you have to instruct the screen reader to backtrack and re-read something. So the experience of using the web if you are visually impaired has some things in common to interacting with web-pages using `lynx`.

9.1 Pipes and Redirects

One of the fundamental philosophies of Unix—and one that is a sensible philosophy when you're building any computer system really—is that the operating system is composed from lots of simple sub-systems, each of which performs one clearly defined task. To do something more complex than any of the individual tools allows you to do on its own, you are expected to combine components yourself. At the command line, Unix makes this quite simple, so let's give it a go.

First, use lynx to look at the BBC's weather page at `www.bbc.co.uk/weather` and have a quick browse around to get familiar with what it looks like. Then quit `lynx` and get back to the command prompt before typing:

```
$ lynx -dump http://www.bbc.co.uk/weather
```

Note the addition of the `-dump` argument before the URL this time. Instead of running as an interactive browser, `lynx` should have just output the text that it would have displayed for that page to the screen, and then ended. Now, most of the text of the page will have scrolled off the top of the screen, so let's use the `less` command to allow us to page through `lynx`'s output in a more controlled manner. Type:

```
$ lynx -dump http://www.bbc.co.uk/weather | less
```

Did you type all that? Hopefully not—remember you can use the up and down arrow keys to get previous commands back at the interactive prompt, and then just modify or extend them to save wearing out your fingers.

To explain what's happened here, you'll have to understand the concepts of **standard in** and **standard out**, which are a neat and extremely powerful idea that is fundamental to the way tools (and programs generally) work in a Unix environment.

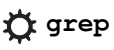
Every Unix program has access to a number of ways of communicating with other parts of the operating system. One, **standard in**, allows a stream of data to be read by the program; another, called **standard out**, gives the program a way of producing text. By default, when you execute things at the command prompt, the shell arranges for a program's standard in to be connected to whatever you type at the keyboard, and for its standard out to be connected to whatever display you're using at the time (this is a bit of an over simplification, but it'll do for now). It's quite easy to arrange for standard in and standard out to be connected up differently though, and that's what you've just done.

The vertical bar '`|`' before `less` is called the **pipe** symbol, and it is used to join the output of one command to the input of another; so in this case we have connected the standard output from `lynx` directly to the standard input of `less`. When `less` is invoked without a filename argument, it expects to get its input from standard in.

As well as being able to join commands together, you can use the idea of manipulating standard in/out to create or consume files instead. Try:

```
$ lynx -dump http://www.bbc.co.uk/weather > weather.txt
```

and then use `ls` to confirm that a file called `weather.txt` has been created, and use `less` to look at its contents (which should be just the text from the weather web-page we've been looking at already). Here the '`>`' symbol **redirects** the standard out of the `lynx` command so that instead of going to the screen it gets put into a named file.



To finish off this first contact with pipes and redirects, we'll use a new command called `grep` along with lynx to create a simple command of our own that tells you what the weather is like in Manchester (there are very few labs with windows onto the outside world in the Kilburn Building, so this may be more useful than you think!).

`grep` is a hugely powerful and useful utility, designed for searching through plain-text files. Learning to master `grep` will take more time than we have in this lab, since you'll have to understand the idea of **regular expressions** to make full use of it (we'll come to those in a later lab). For now, we'll use it in its very simplest form. Type:

```
$ grep BBC weather.txt
```

and you should see a list of all the lines from `weather.txt` that contain the word 'BBC'. Use `less` to have a look for other terms to '`grep`' for (you might want to try something like 'Sunny' to give you a list of all the places where the weather is nice, for example).

Rather like `less`, if `grep` isn't given the name of a file as its last command-line argument (in this case we used `weather.txt`), it will operate on standard input instead of grepping through a file (yes, it's quite okay to use grep as a verb from now, no one will look at you funny). Use this knowledge to join together `lynx` and `grep` so that the output is a single line describing the weather in Manchester at the time we run the command. The output should look something like:

```
   [33]Manchester 22°C 72°F
```

As a final flourish, let's create a new a way of accessing this new 'weather in Manchester' tool that you've created. Type:

```
$ alias mankyweather="[YOUR COMMAND GOES HERE]"
```

replacing [YOUR COMMAND GOES HERE] with the full command line you created to display the Manchester weather, being careful not to introduce extra spaces around the equals sign `=`. Then try typing

```
$ mankyweather
```

to see the result. Okay, so this probably won't replace your favourite weather web page or app, but it's early days yet! Note that this alias will disappear once you exit the shell in which you created this, for example when you logout and login again. We will see in a later lab how to make such **aliases** permanent.

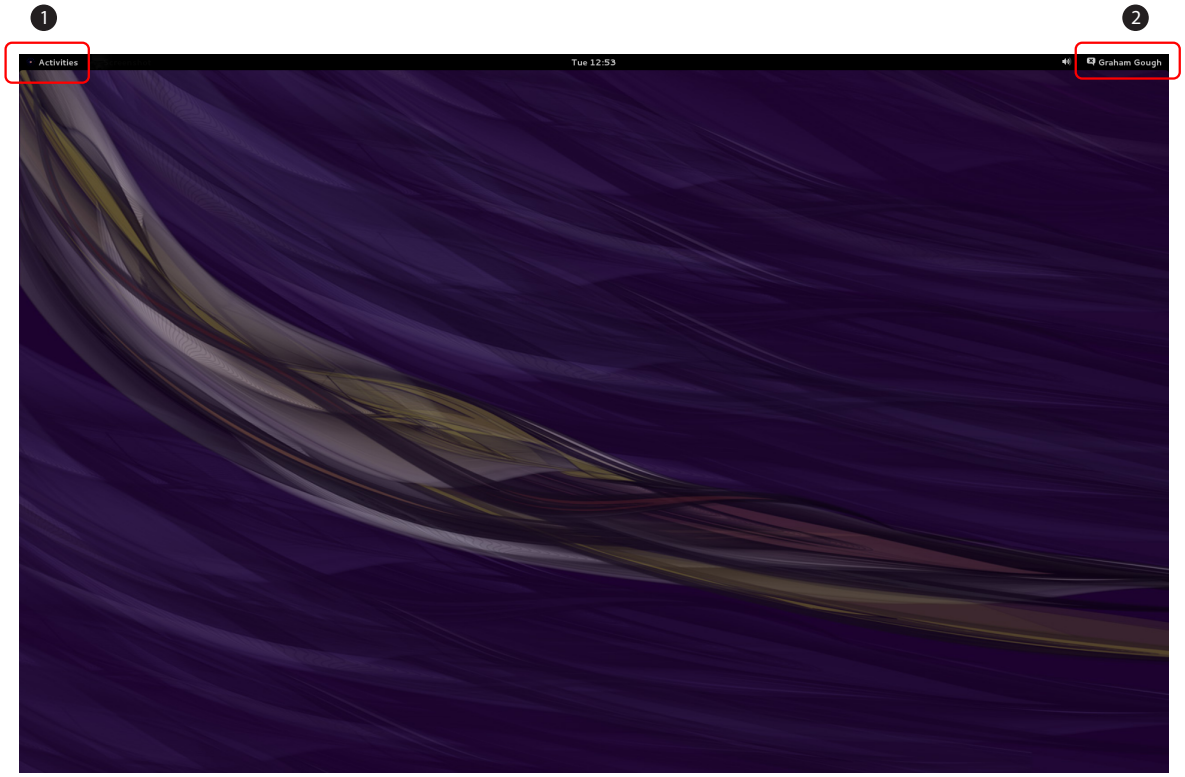


Figure 7
Scientific Linux's default graphical user interface and window manager, GNOME 3. The screen is almost empty but clicking on ❶ Activities will give you a screen which looks like 8. Clicking on your name ❷ shows the 'log out' option.

Breakout 4: GNOME and Mutter

It's quite common to refer to GNOME as a 'window manager', but technically it is much more than that; it's actually a collection of tools, applications and other programs that together form a graphical desktop environment. The window manager component of GNOME 3 is called **Mutter**[W].

10 X Windows and GNOME

Next you're going to start up one of Linux's many graphical user interfaces. Type:

```
$ startx
```

You'll see a chunk of text scroll up the screen briefly before being presented with something that looks like the screen shot in Figure 7, although the background may look slightly different.

Take a few minutes to explore the graphical environment. Even if you've never used Linux before, you'll probably find the general principles of this environment quite familiar: there are icons on the desktop giving you access to the computer via a graphical file browser, and at the top of the screen a menu-bar allows you to start various applications and utilities. The full manual for this environment—which is called GNOME 3—is available online at

`https://help.gnome.org/users/`

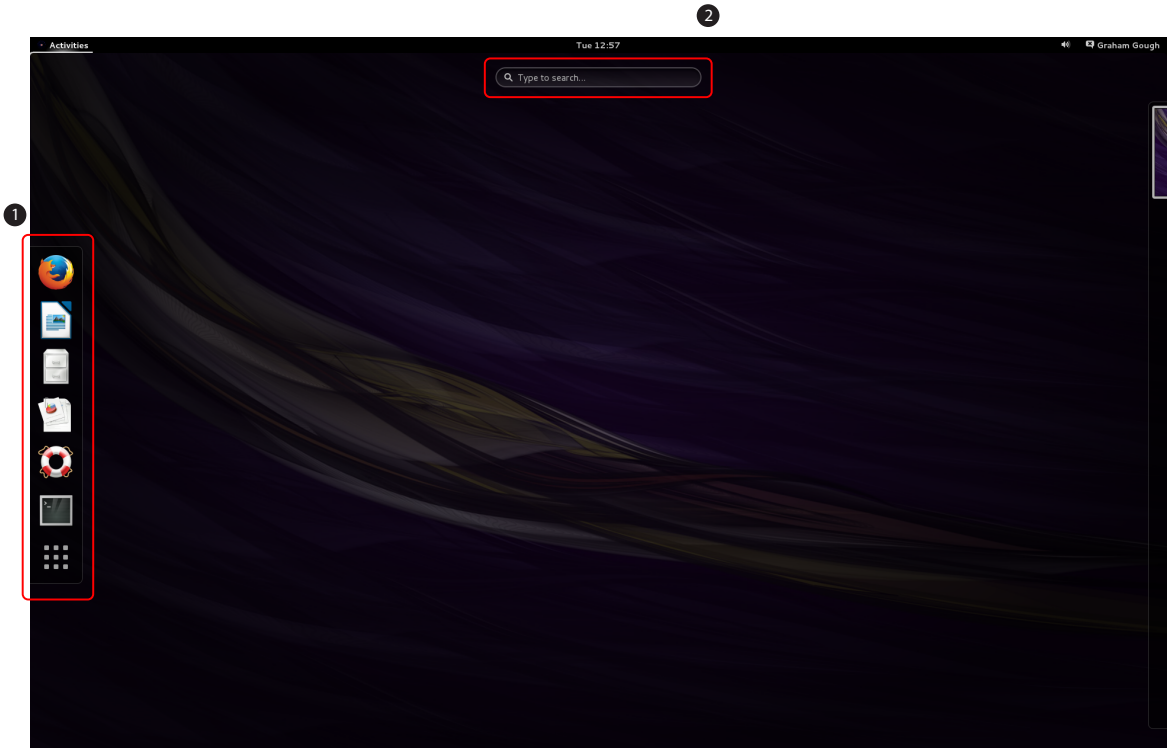


Figure 8
GNOME 3 Activities. ❶ the left panel contains shortcuts to some frequently-used tools. You can configure the shortcuts to include your favourite things, and ❷ a search box can be used to easily find other applications.

Perform the following tasks:

1. Find two different ways to start a terminal window.
2. Find two different ways to start the Firefox web-browser.
3. Use Firefox to visit the School UG home page: `studentnet.cs.manchester.ac.uk/ugt/`
4. Work out how to change the desktop theme and choose one you like.
5. Create a keyboard shortcut for starting Firefox, to provide a third way of starting it.
6. Find the **Vector Graphics**[W] drawing application called Inkscape, and use it to draw a simple self-portrait. We just want you to spend a couple of minutes getting used to the kind of things that Inkscape can do—it will be very useful later in your degree programme when you're going to need to draw diagrams to go in your reports. For now any old doodle will do quite nicely (look at what Steve drew in Figure 9, we're really not setting the bar very high at all here!). Make sure you save this file, we're going to need it later.
7. Figure out how to log out of the graphical environment.

If you've completed step 7 you should now be back at the command prompt where you typed `startx` a little while back. Before returning to the graphical environment where you'll spend most of your time, it's important to understand how the graphical interface you've just been using works as part of the Unix operating system.

startx

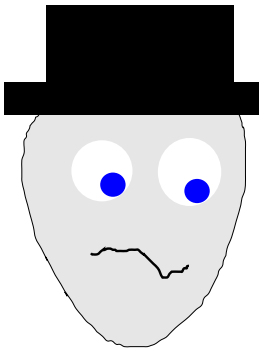


Figure 9
This is a picture of Mister Noodle drawn by Steve using Inkscape. It took about two minutes, though in reality had he spent any more time on it there would be no obvious improvement in the quality of the artwork.

Breakout 5: Inkscape and GIMP: Vector and bitmap graphics

Inkscape is a vector graphics drawing package; it allows you to draw and manipulate different shapes to create pictures and diagrams. It is ideal for drawing diagrams and figures. When you're using a tool such as Inkscape you're manipulating geometrical shapes such as points, lines and curves. One of the big advantages of this approach is that images look the same regardless of what magnification you use. In these notes we've tried where possible to use vector images, so you should be able to zoom into the pages on the electronic version without seeing any 'pixellation' happening. Some figures contain a mixture of vector and bitmap graphics; for example, zoom into Figure 7 and you'll see that the image of the desktop itself starts to become jagged (because it's a bitmap), whereas the red boxes and numbers stay nice and crisp at any magnification (because they're vectors).

GIMP, on the other hand, is a bitmap based image manipulation package; it treats images as being made up of lots of coloured dots (pixels). GIMP is great for editing photographs and creating certain types of artwork, but it's not hugely useful for drawing diagrams.

It's worth understanding the pros and cons of these two different approaches to graphics, it'll save you a lot of heartache later on and you'll end up creating more professional looking figures in your documents. The Wikipedia page on **vector graphics**[W] provides a good explanation of the different approaches.

PG It is important to understand that the shell (`bash`) that you're using to interpret commands is 'just a program' that happens to interpret input from the user, execute commands, and display the results. The graphical environment you've just used is similar—just a program (or actually, collection of programs) that runs on the operating system.

bash

But what do we mean by 'execute commands'? You've probably got the hang of the fact by now that most of the things that happen in Unix are just programs stored somewhere on the file system (remember, you found some of them in the `/bin` directory). When you press `Enter` at a shell prompt, the shell checks that what you've typed has a valid **syntax**, and then starts up a new **process** in which that program executes. The process is mostly independent of the shell program that started it, gets on with doing whatever it was designed to do, and

when it finishes it tells the shell that it's done, and the shell gives you another prompt for the next instruction. Something very similar happens when you run the `startx` command: the graphical environment starts executing, and when you select the 'log out' option, it returns you back to the shell so you can issue another command. Notice that you haven't been 'logged out' of Linux, but rather just out of the graphical environment – we'll show you how to configure things so that 'log out' in the GUI really does log you out in a little while. But first, let's take a step back now and look at what the `startx` command has actually done.

11 X Windows

Unlike OS X and Windows and most mobile operating systems, Linux doesn't really have a graphical windowing environment 'built in'; what you've seen just now is a series of programs that co-operate with one another to create the familiar WIMP environment .

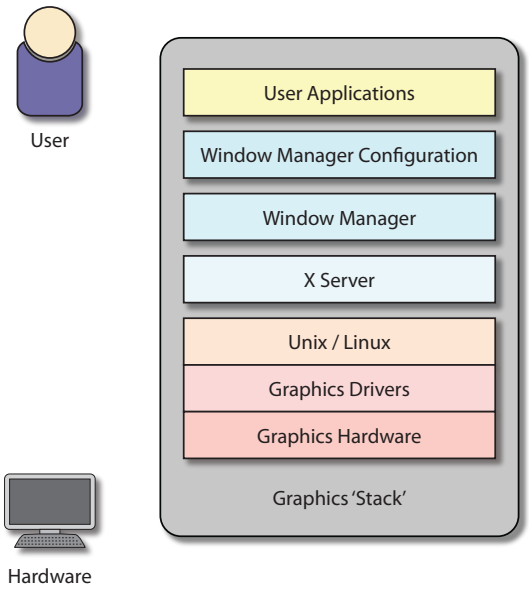


Figure 10
The layered structure of Linux's graphical system, with software nearest to the underlying hardware at the bottom, and software closest to the user at the top.

12 Window Managers

One of the interesting effects of the X architecture, shown in Figure 10, is that it separates out the system that draws stuff onto the screen from the one that deals with creating buttons, sliders and windows, and enables you to choose a window manager that best suits the way you work; some people like 'rich' environments like GNOME, whereas others like 'lean' cut-down window managers.

The `startx` command can be used to fire up window managers other than GNOME, but the syntax used for doing this varies quite a lot from one Linux distribution to another, and in the case of the version of Scientific Linux we're using, its behaviour is a bit confusing. To make it a little easier for you to experiment with different window managers, we've provided you with a bespoke command `csstartx` that makes things a bit simpler. Because this is a program that

we've added outselves to the Linux distribution, we've followed the convention of not putting it in one of the system's own directories of commands (such as `/usr/bin`), which means that it won't get found automatically in the way that other commands you've used so far will, so for now you'll have to explicitly type its full absolute path name, and we'll show you how to modify this behaviour shortly.

Type:

```
$ /opt/teaching/bin/csstartx startkde
```

and you should find that a different graphical desktop environment, KDE, starts up. Experiment with this for a minute or two just to get a feel for what it's like, and then quit back to the command-line prompt.

As well as GNOME and KDE, which both follow fairly similar interaction paradigms to the graphical environments of other operating systems, there are many alternative graphical interfaces for Linux.

Use the table that follows to explore some of these by using the value from the 'Command' column as the first command-line argument to `csstartx` instead of `startkde`. In each window manager make sure you figure out how to create a terminal window, so you can get some work done!

Name	Command	Description
Fvwm	`fvwm`	A lean window manager with virtual desktops **Hint: Click the left mouse button.**
AnotherLevelUp	`ALU`	A customised version of fvwm developed here in CS by John, and still his favourite desktop environment (may contain bright colours). **Hint: Click the left mouse button.**
Fluxbox	`fluxbox`	A lean but highly customisable window manager. **Hint: Click the right mouse button**.
Openbox	`openbox`	Similar to Fluxbox; lean and highly configurable. **Hint: Click the right mouse button.**
Awesome	`awesome`	A very very lean *tiling* window manager. **Hint: Right click on the black background. To exit press windows-shift-Q.**

As you become more familiar with Unix principles, keep the fact that you can easily swap window managers in mind. Most likely there will come a point where the graphical niceties of environments like GNOME become unnecessary, and perhaps even a distraction from getting work done, and you might find that a slimmed down window manager suits you better as a more experienced 'power user'. For the rest of these exercises, though, we'll assume you're using GNOME (if you're confident enough to use something else, then translating our instructions to make sense in whatever environment you've chosen won't be too big a problem).

13 Starting a graphical environment automatically

Now, if you're going to use the graphical environment as your primary interface (and, as the jobs we ask you to do get more complex, you're going to need to!), you may find it slightly annoying to have to log into a lab machine, start the graphical environment, log out of the graphical environment when you're done *and then remember to also log out of the terminal environment before you leave (because if you don't do this, other people will have access to your account!)*.

Back in the previous lab session we explained that when you log into a Unix machine, an interactive shell is created for you to run commands from, and that the shell is 'just a program'

Breakout 6: Tiling window managers

It's likely that you are so familiar with the way that windows are managed on operating systems like Windows or OS X that you've never really thought about alternatives. Awesome and Xmonad are very minimalist window managers, probably quite unlike anything you've used before. Most WIMP environments that you'll have used so far make you the user responsible for the position and size/shape of the windows that represent tools and applications on the desktop. The upside of this is that you can arrange things exactly as you like them; the downside is that you probably use up an amount of time doing that arrangement, and often end up with a layout that wastes some of the desktop's usable space. Awesome is what's called a 'tiling' window manager; instead of giving you detailed control over the exact shape of windows, it lays them out on the screen in one of several configurations designed to maximise the use of space. Because you can't drag or resize windows with the mouse, there's no need for the usual window decorations, so you save a few pixels this way too.

There's no doubt that these window managers are at the hard-core end of the window manager spectrum, and are designed for experienced users that need a very large numbers of windows open at once, probably spread over several physical displays (as in Figure 11). Apart from the 'tiling' aspect, it gives you virtually no visual cues as to how to perform various actions, most of which are designed to be invoked via keyboard shortcuts (in fact, Awesome is designed so that you can use all of its features without needing to touch the mouse at all.) Once you've remembered all the keyboard combinations, using a window manager like Awesome or Xmonad can be extremely efficient in terms of time and screen-space.

like any other that just happens to be the one nominated to be the first thing to run when you log in.

What about nominating the graphical environment as the first thing to run instead? We've already pointed out that it's 'just a program' too, so that should be okay? Although in theory it's possible to do this, in practice it's a bad idea: shells are quite simple self contained programs, whereas graphical environments are much more complex systems relying on hundreds of files to be installed in the right places in order to work. You certainly don't want to set your system up in a way that if the graphical environment gets damaged in some way you can't log in at all.

Instead we'll show you a rather safer way to get the GUI to fire up when you log in.

When you first run the bash shell on login, it looks for a file in your home directory called `.bash_profile` and executes any commands it finds in there as though you'd typed them at the keyboard; so this is a useful place to put the command to start the graphical environment. Use the `ls -a` command to confirm that there's already a file in your home directory called `.bash_profile`, and then use `less` to look at its contents (There should also now be one called `.bash_history`, take a look at it and it should become obvious how the `history` command, and the 'reverse search' function you used earlier work).

`.bash_profile` should look something like this:

```
# .bash_profile

# Get the aliases and functions
```

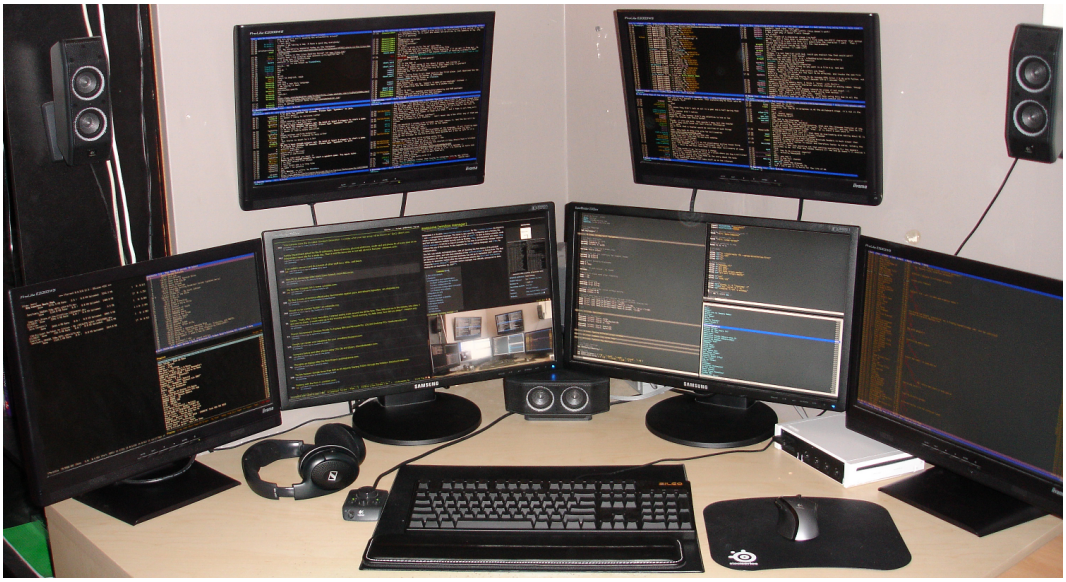


Figure 11
The Awesome window manager showing around 20 windows tiled over 6 physical displays. Reproduced from `awesome.naquadah.org` with kind permission of Julien Danjou, one of Awesome's primary authors.

```
if [ -f ~/.bashrc ]; then
. ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/.local/bin:$HOME/bin

export PATH
```

though don't worry if there are slight differences. We'll come back to what these instructions mean in a later lab.

Start up the `nano` editor, and use it to add the following as a new line at the end of your `.bash_profile`:

nano

```
/opt/teaching/bin/csstartx
```

Use `less` to check that this line has now been added to your file.

If you've already decided you want to try out something other than GNOME as your default window manager, add the appropriate command from the table after `csstartx`.

Now log out, and log back in again; you should find that the graphical environment fires up automatically.

Next, quit out of whichever graphical environment you've chosen and…oh dear, you're back at the command prompt, rather than logged out completely.

Now this could be really confusing; you've created a situation where you don't have to start the graphical environment up manually, but you do have to remember to log out twice when you've finished using it, once from the GUI and again from the console prompt. Yuck.

Fortunately there's a relatively easy fix for this.

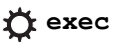
At the command prompt, type the following:

```
$ exec man ls
```

You should find that the `man` command has done exactly what you normally would expect, but that instead of returning you to the command prompt when you've finished reading the man page, you've been unceremoniously logged out! Log back in again (sorry about that).

The `exec` command changes the way in which the shell deals with whatever command follows it. Instead of starting a new process in which to run your command and waiting in the background for that command to complete, the shell gives up the process in which it itself is running, and hands it over to the command you've issued. So when that command finishes, there is no shell to come back to. And because in this case the shell was the first program that got run when you logged in, the Unix system logs you out since there's nothing else you can do.

Experiment by running `exec /opt/teaching/bin/csstartx` and then logging out of the graphical environment as you did a moment ago; this time you should find that you've automatically been logged out of the console too.



Use the `nano` editor to alter the line you've just added to `.bash_profile` so that it now reads

```
exec /opt/teaching/bin/csstartx
```

Now log out, either by typing `logout`, or pressing `<ctrl>d` to tell the shell that its input has ended. Now log back in again; if all has gone to plan then you should see the graphical environment fire up automatically; and when you quit the graphical environment, you should be returned to the Linux login prompt.

Hurray!

Before we leave this section on graphical environments, there's one quirk that we've got to deal with to avoid causing problems later on. As we said earlier, instructions in the `.bash_profile` file get executed when you log in to a Unix machine

This works fine when you login directly to a PC from the console, but will cause problems if you login remotely from another machine, as you will in the next lab. So we need to arrange things so that we only try to start up the graphical environment if the user has logged in from the console, and not via a remote connection.

Carefully replace the line you've just added containing `csstartx` at the end of `.bash_profile` with the lines:

```
case $(tty) in
/dev/tty1) exec /opt/teaching/bin/csstartx
esac
```

Note that the character after `/dev/tty` is the digit `1` and not the letter `l`.

This piece of 'shell script' reads something like 'in the case where the terminal device being used is the physical console, then execute the graphical environment'. We won't explain the exact meaning of this piece of code now but will do in a lecture; for now just make sure you've typed it exactly as written here.

You should now log out and login again to check that the graphical environment starts up, shuts down and logs you out as expected.

If you've mistyped any of the lines in `.bash_profile` you may find that when you try to log in, you're instantly logged out. Don't panic and call for some help from the lab staff; it's a common mistake and an easy one to rectify!

14 Text Editors

A great deal of the lab work you will be doing over your time here will involve you creating text files of various kinds, often source files in a programming language such as **Java**[W], **PHP**[W] or **C**[W], or **HTML**[W] files for use on the web. There are specialist tools called **Integrated Development Environments**[W] or *IDE*s that can be used for programming; you will meet these later in your programme. However, for many purposes, the simplest, and best, tool for creating such files is a simple text editor. You have already met one such tool, `nano`, which is fine for work at the console or quick modifications of existing files, but for more extensive work an editor that takes advantage of X's graphical capabilities is more appropriate.

The Linux environment in which you will be working offers many such editors, including the default GNOME editor `gedit`, the KDE editor `kate` and the grand daddy of all editors, `emacs`. These three are illustrated in Figures 12 and 13. They are all shown ready to edit a Java source file; note that they all use the fact that this is Java source to highlight key words within the text. When you have some free time, please do experiment with some of the text editors available and find one that you like; in the meantime you should probably use `gedit`.

15 Shell environment variables

When you used the `csstartx` command, you had to prefix these with their full pathname, which is `/opt/teaching/bin/`. The directory `/opt/teaching/bin` is a place where we keep lots of useful teaching related tools, so it would be useful if you didn't have to type it every

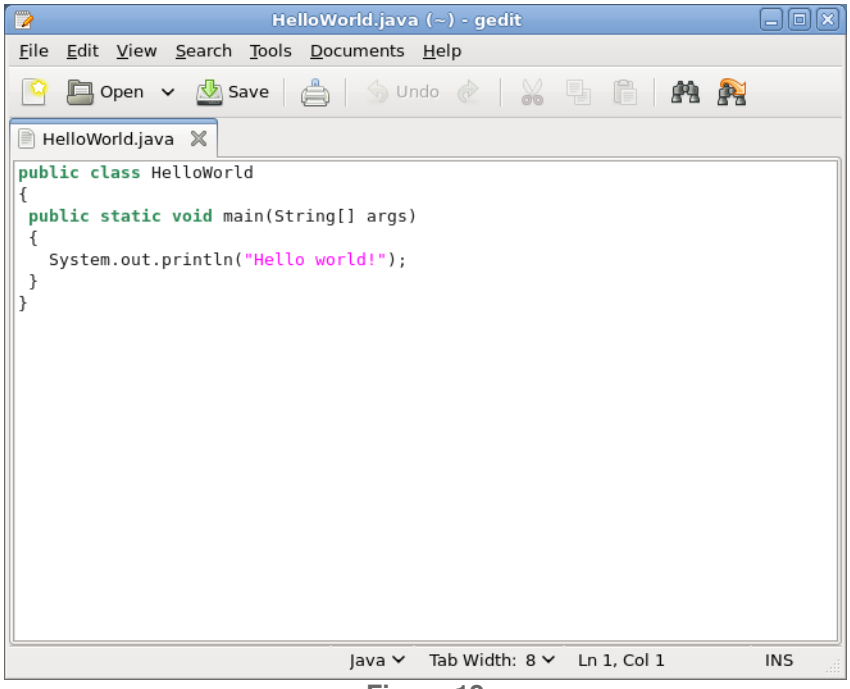
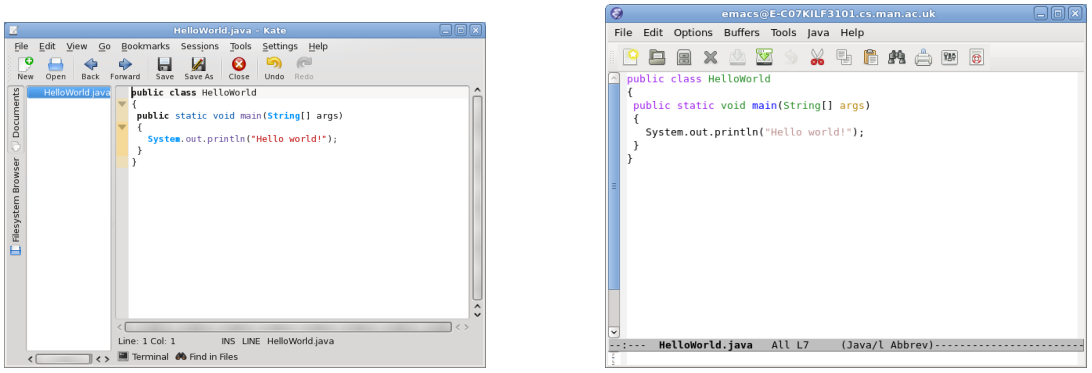


Figure 12
`gedit`



(a) `kate` **(b)** `emacs`

Figure 13
Other editors

time you want to use a program from there; luckily there's a way of doing this. When you type a command on the command line, the shell looks for a program of that name in a number of places. These places are determined by the value of a shell **environment variable**[W] called `PATH`. You can see what its current value is by using the command

```
$ echo $PATH
```

This will show a long list of directories, separated by colons (`:`).

There are many other shell variables already set for you, they can be seen by running the shell command `set` (do this now). How do you stop the output scrolling off the screen? Most of these variables won't make much sense to you at the moment, but among them are `HOME`, `PWD` and `HOSTNAME`; you can check their values using `echo`. What do think their values represent?

set

echo

You can set the value of a shell variable at the command line, for example type:

```
$ MYVAR=42
$ echo $MYVAR
```

Note that there should be no spaces either side of the = sign and that the variable's name is `MYVAR` and its value is obtained by using `$MYVAR`. If a variable is given a value on the command line in a terminal window like this its value is only available in the shell running in that window.

In order to be able to start the ARCADE client by typing `arcade` rather than by using the full pathname `/opt/teaching/bin/arcade` we need to modify the `PATH` variable to include the directory `/opt/teaching/bin`. We can do this on the command line by typing

```
$ PATH=$PATH:/opt/teaching/bin
```

This can be read (and you will see this over and over again in programming languages) as 'find the value of the right hand side of the = and give (or **assign**) that value to the variable on the left hand side'. So this takes the current value of `PATH` and adds `:/opt/teaching/bin` on the end. After you've done this you can type `arcade` in your terminal window and the ARCADE client should start. Just exit it cleanly and return to the shell.

If you now start another terminal window and type `arcade`, what happens and why?

The way to make the change permanent is to modify your `.bash_profile` file. While we are doing this we will also add two more useful directories to our PATH, namely `/opt/common/bin` and `/opt/scripts`, which are also shared directories containing useful local stuff.

Use `gedit` to modify `.bash_profile` by adding the following line *immediately **before** the existing line starting with `PATH`.*

```
PATH=$PATH:/opt/teaching/bin:/opt/common/bin:/opt/scripts
```

This won't have any effect until you logout and login again. So do that now and try starting `arcade`. Run `echo $PATH` and you will see the new value, which contains your own `bin` directory, now preceded by the three `/opt` directories.

16 Configuring Thunderbird

Rather than explain the Thunderbird configuration here we refer you to an illustrated guide to this process, which you can find at `https://wiki.cs.manchester.ac.uk/index.php/How_to_set_up_Office_365_Mail_in_Thunderbird`, written by a fellow student.

Follow all three steps in the process described in this document.

Once your Inbox appears in Thunderbird, using it to compose and send email should be fairly self-explanatory, but if you're stuck there are plenty of Thunderbird tutorials available on the web.

17 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.

- 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
  ```

  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
  ```

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.

17.1 Creating a directory structure

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

```
$  cd
$  mkdir INTRO
```

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
```

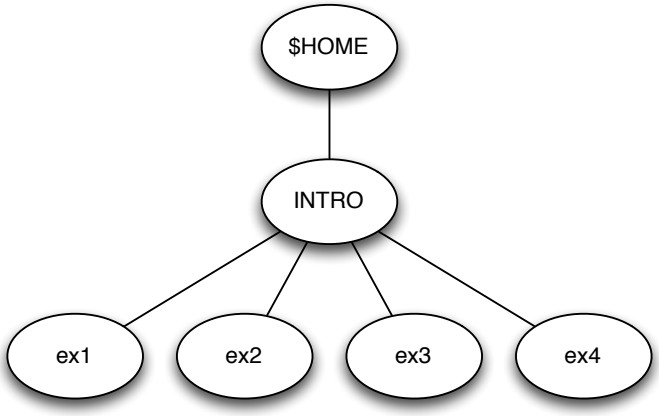
And then try to make it correctly.

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**[w]—if you've not looked up what this means yet, now is a good time to do that).

chmod

```
$ ls -R
```

17.2 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 you command line arguments.

Moving / renaming files: mv

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

Breakout 7: 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.

17.3 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

```
$  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

Breakout 8: 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.

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.

17.4 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.

17.5 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.

17.6 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 con**cat**enate 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).

PG Earlier we met the idea of a **pipe**, using the `|` symbol to connect the **standard output** of one command to be **piped** 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 8, 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.)

fortune

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:

cowsay

```
 _____________
< 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.

17.7 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 9.1 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:

Breakout 9: 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.

```
$ 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 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.

17.8 Doing several things at a time: background running

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

xclock

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`.

jobs

17.9 Printing text files: lpr

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

```
$  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 **head** 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 **a2ps** 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.

17.10 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 you 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 *` cd

   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.) wc

   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.

18 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.