Intro Lab Session 2

Using the Linux desktop in CS

Contents

These notes are available online at
`studentnet.cs.manchester.ac.uk/ugt/COMP10120/labscripts/intro2.pdf`
You may find it useful to use the online version so that you can follow web links.

In this lab session we're going to explore some of the features of Unix in a bit more depth, this time using the desktop PCs rather than your Raspberry Pi (we'll return to using that in the next lab). We'll explore some of the more advanced features of the command line and various useful tools that will help you understand how a typical Unix system is organised. Almost everything that you learn using Linux on the desktop machine is equally applicable to the Raspberry Pi, and vice versa.

2.1 Logging in

Make sure the desktop PC is booted into Linux, and log in using your University username and password (*not* the username and password you used on the Pi). 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.

2.2 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 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**

```
$ 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!
- 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!

Breakout 2.1: 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.

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 2.1 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.
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"
```

Breakout 2.2: 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 2.3: 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**

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 2.3). 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.

⚙ **less**

⚙ **man**

Don't worry too much about the details of this file for now. If you're already familiar with how IMAP and SMTP work together to provide your email service, then you'll be able to see what the contents of this template mean; if you're not, don't worry, it'll all be explained in detail in the COMP18112 (Fundamentals of Distributed Systems) course in the second semester. 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.

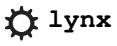
If you're confident that you now have a file called `.muttrc` containing the correct configuration, you can now type `mutt` to start the program.

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.

2.3 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`.

You'll soon be designing your own web-based systems as part of the Team Project in the later stages of COMP10120; making them accessible to visually impaired readers is something you should keep in mind. Try using `lynx` to browse some of your favourite websites, and you'll almost certainly find that the level of 'accessibility' on the Web varies considerably!

2.3.1 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.

grep

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.

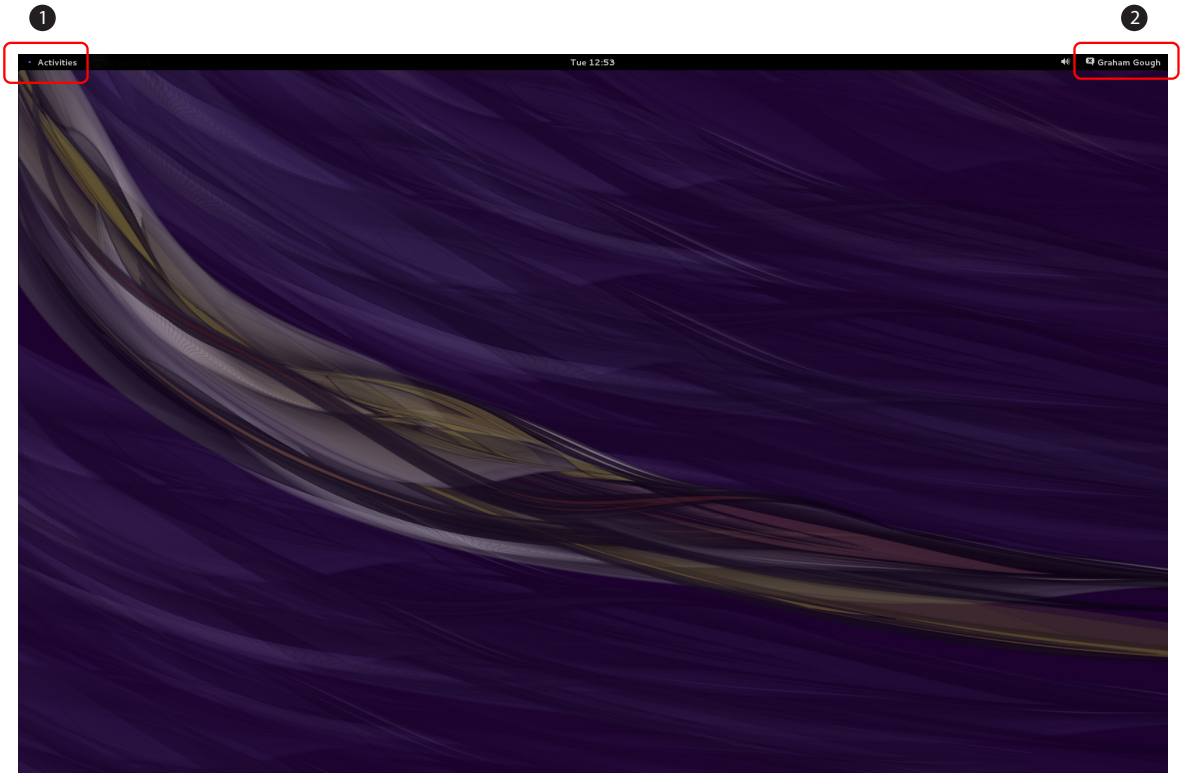


Figure 2.1
Scientific Linux's default graphical user interface and window manager, GNOME 3. The screen is almost empty but clicking on ❶ Activities will give you a screen which looks like 2.2. Clicking on your name ❷ shows the 'log out' option.

2.4 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 2.1, 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/`

but you'll probably be able to work out everything you need to get you going by poking around at the various buttons. Unlike the Raspberry Pi where you have complete control over the operating system via the `sudo` command, the lab machines are configured so that you can't do any long-term damage to the setup. Apart from accidentally deleting your own files (and right now you have very little important stuff to accidentally delete!), there's nothing much you can do that will cause problems, so feel free to explore a bit.

Perform the following tasks:

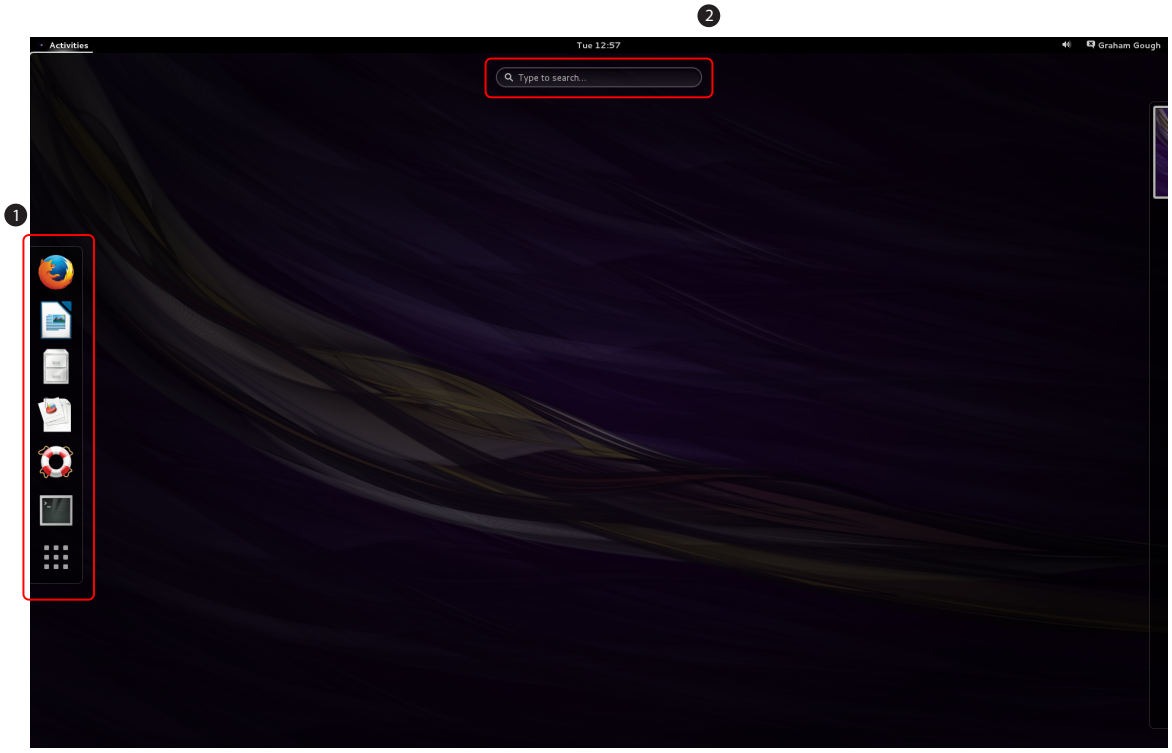


Figure 2.2
GNOME 3 Activities. ❶ the left panel contains shortcuts to some frequently-used tools. You can configure the shortcuts to include your favourite things, and ❷ a search box can be used to easily find other applications.

Breakout 2.4: 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].

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 2.3, 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.

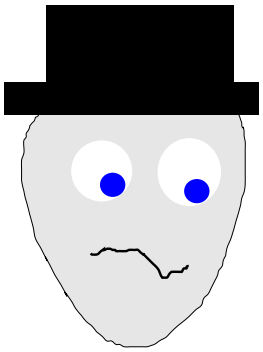


Figure 2.3
This is a picture of Mister Noodle drawn by Steve using Inkscape. It took about two minutes, though in reality had he spent any more time on it there would be no obvious improvement in the quality of the artwork.

Breakout 2.5: 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 2.1 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.

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

If you remember back to the first Raspberry Pi lab, we pointed out 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

bash

system.

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.

2.5 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 (if you don't know what WIMP means yet, go back and read Breakout 1.1 from earlier in these notes).

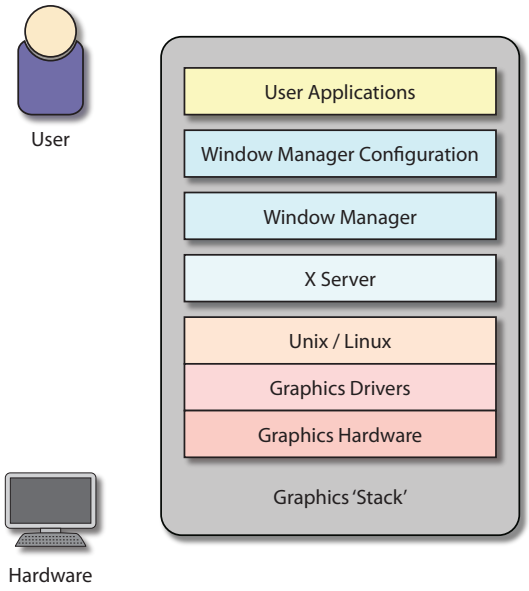


Figure 2.4
The layered structure of Linux's graphical system, with software nearest to the underlying hardware at the bottom, and software closest to the user at the top.

When you ran Quake and the snake game on the Pi in the previous lab, these programs took direct control of the graphics subsystem in order to display the game. The `startx` command runs a system called **X Windows**[W], which also takes control of the computer's graphics system, but on its own doesn't really do anything very exciting apart from allow other programs to then share the display. Along with X Windows, another system called a **Window Manager**[W]

was started, and this is what you see drawing the buttons and menus and window controls for the graphical user interface. We'll look at how X Windows really works in one of the forthcoming COMP10120 lectures, and you'll explore some aspects of the architecture of X Windows in a lot more detail in COMP18112 (Fundamentals of Distributed Systems) in the second semester. For now, it's enough to understand that there are two things going on here: first the X Windows system is running that allows stuff to be drawn to regions of the screen; and second the Window Manager which is doing all the WIMPy stuff like providing all the controls that allow windows to be moved and resized.

2.6 Window Managers

One of the interesting effects of the X architecture, shown in Figure 2.4, 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!

Breakout 2.6: 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 2.5). 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.

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).

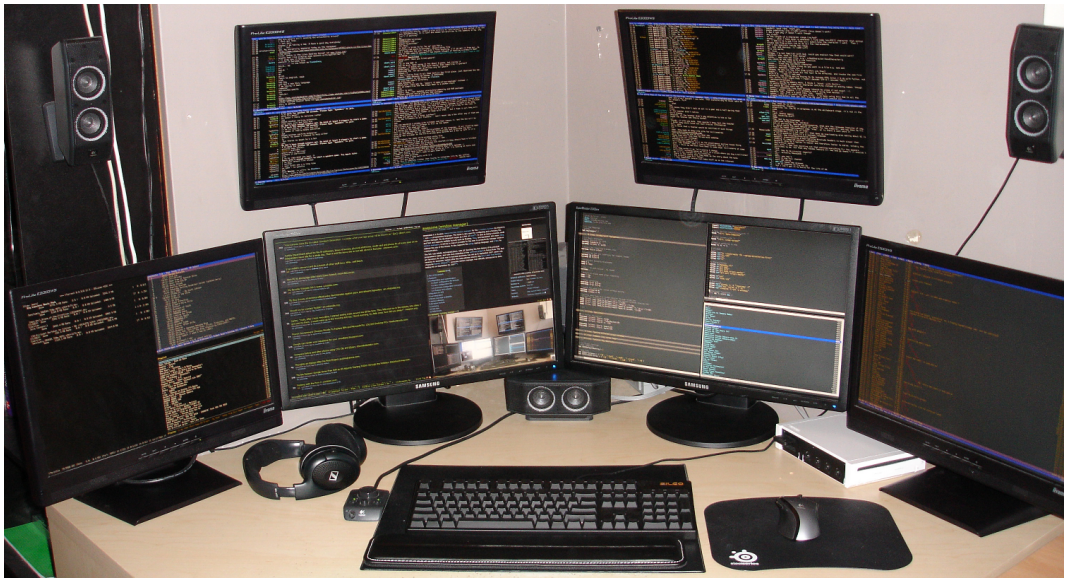


Figure 2.5
The Awesome window manager showing around 20 windows tiled over 6 physical displays. Reproduced from `awesome.naquadah.org` with kind permission of Julien Danjou, one of Awesome's primary authors.

2.7 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' 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.

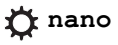
When you first run the bash shell on login, it looks for a file in your home directory called `.bash_profile` and executes any commands it finds in there as though you'd typed them at the keyboard; so this is a useful place to put the command to start the graphical environment. Use the `ls -a` command 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
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`:




```
/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!

2.8 ARCADE

We would now like you to register your access to the ARCADE server. ARCADE is the laboratory management system we use to administer coursework marks and deadlines, etc.. You can use the client query program to look up your details, laboratory timetables, marks, and so on, throughout the year. It is recommended you do this fairly regularly, if only to check that mistakes have not been made in your marks! You need to register with ARCADE before any of your lab work can be submitted, so it is much better to do it now than wait until your first deadline!

In a terminal window, run the command `/opt/teaching/bin/arcade`; after a short pause, a new window will pop up. In the large text box you should find a message telling you that you are not yet authenticated, and that you have just been sent an email. If this is so, then quit the

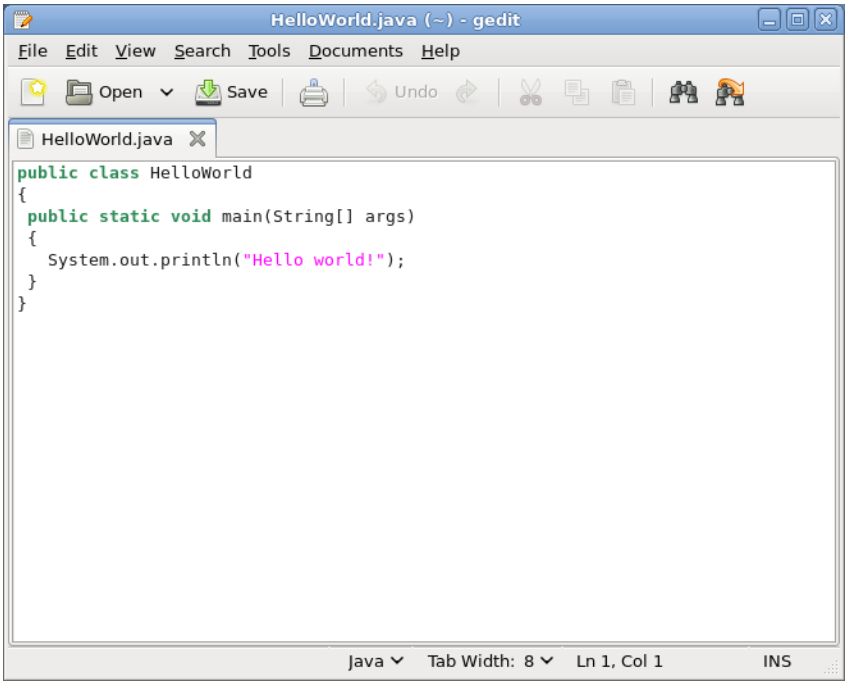


Figure 2.6
`gedit`

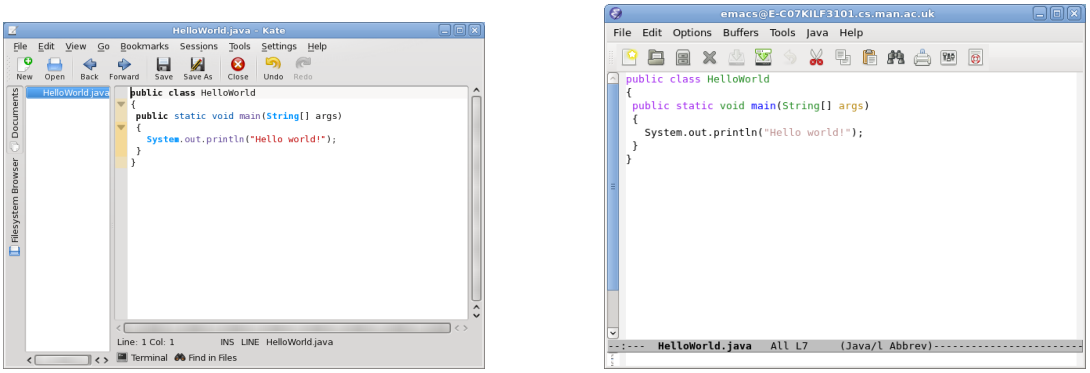
program, read the email you have just been sent by the ARCADE server, and carefully follow the instructions in it. Now start the ARCADE Query client again, and it should now connect you to the server. If this works, then your access is set up; if it doesn't then check you've followed the instructions precisely. If it still doesn't work then, as always, ask for help; *it is very important that this registration process is completed.*

Explore the various queries of the client, and familiarise yourself with the the user interface. Check that your registration details are correct. Feel free to ask for explanation if it is not obvious to you.

2.9 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 2.6 and 2.7. 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`.



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

Figure 2.7
Other editors

2.10 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 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.

2.11 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.

2.12 That's all for now

If you have reached this point before the end of the lab you may have gone too fast so please go back and review what you have done. You will be using many of the ideas we've just met in later labs, so it's important to understand them. When you are sure you have completed the work for this session, please tell the lab supervisor.