MSc Introductory Laboratory

Academic Year 2021-22

Christoforos Moutafis, Ahmed Saeed, Stewart Blakeway, Graham Gough, Toby Howard, John Latham, Chris Page, Steve Pettifer

Contents

1 Computing Infrastructure

Hello and welcome to the Department of Computer Science at the University of Manchester! In this first, boot-up, lab we're going to cover some of the basic things you'll need to know about the IT infrastructure in the Department of Computer Science, and how to install the Virtual Machine and get it set up so that you can get going quickly in the next lab. You will need this in all of your subsequent labs as well. 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 and specifically for the academic year 2021-22, but there may be some places where they betray their origins.

For this and every lab there will be scheduled online and on campus drop-in sessions where staff, consisting of academic staff and postgraduate students, will be 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 **Graduate Teaching Assistants**, or **GTAs** for short.

2 About these notes

2.1 Breakout boxes

Scattered throughout the main text there are info 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. 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 What is a Virtual Machine?

In this section we will briefly introduce virtual machines (VMs). We will be providing you with the Computer Science Virtual Machine that will enable you to have direct access to all the tools you will need for all of your coursework and laboratories (details will follow in the next section), so it'll be very useful! But, what is a Virtual Machine? In simple terms, you can think of a virtual machine (VM) as a computer within a computer!!! In practice, a virtual machine is a virtual computer system that provides the functionality of a physical computer.

4 Get your Virtual Machine installed

Now that we introduced virtual machines, we will discuss how to perform the inital simple set-up steps in order to subsequently install the Computer Science Virtual Machine that is prepared and tailored for the needs of the Department's undergraduates. In the following you have simple and clear guidelines on how to install the VM on one of the major platforms you may be using: Windows, Mac OS X and Linux.

All this information, can also be found at our wiki, in the relevant page CSImage VM/Getting Started, which is always kept up-to-date:

`https://wiki.cs.manchester.ac.uk/index.php/CSImage_VM/Getting_Started`

By the end of this Boot-up lab 0, you should have succesfully installed the VM on your personal computers.

System Requirements In order to use the CS Virtual Machine Image, your device must meet the requirements to run VirtualBox. VirtualBox officially supports the following host operating systems:

- 64-bit versions of Windows 8.1, Windows 10, and Windows Server 2012, 2016, and 2019.

- 64-bit versions of macOS X 10.13 (High Sierra), 10.14 (Mojave), 10.15 (Catalina), and 11.x (Big Sur) on Intel hardware.
- 64-bit versions of Linux based on kernels >= 2.6. All modern 64-bit Linux distributions will run VirtualBox.

Windows 7 and Windows 8.0 may run VirtualBox, but this is not officially supported. **32-bit host operating systems are not supported by VirtualBox**

Apple systems based on the new Apple M1 processor are not supported by either VirtualBox or VMWare Fusion at this time. The CSImage VM WILL NOT WORK on M1 based Macs.

Your computer must have virtualisation support enabled in its BIOS/EFI settings in order to run VirtualBox properly. Most modern Intel and AMD processors provide the required virtualisation hardware, but it may not be enabled by default.

4.1 Installing VirtualBox and CS Image

This part will be different for Windows users, Mac OS x users and Linux users, and we have prepared dedicated pages for each case, to guide you through, that you can find below:

- Instructions for Windows users:
  `https://wiki.cs.manchester.ac.uk/index.php/CSImage_VM/Getting_Started/Windows`
- Instructions for Mac OS X users:
  `https://wiki.cs.manchester.ac.uk/index.php/CSImage_VM/Getting_Started/OSX`
- Instructions for Linux users:
  `https://wiki.cs.manchester.ac.uk/index.php/CSImage_VM/Getting_Started/Linux`

If you need support with installing or using the CS Image please see the following link:
`https://wiki.cs.manchester.ac.uk/index.php/CSImage_VM/Troubleshoot`

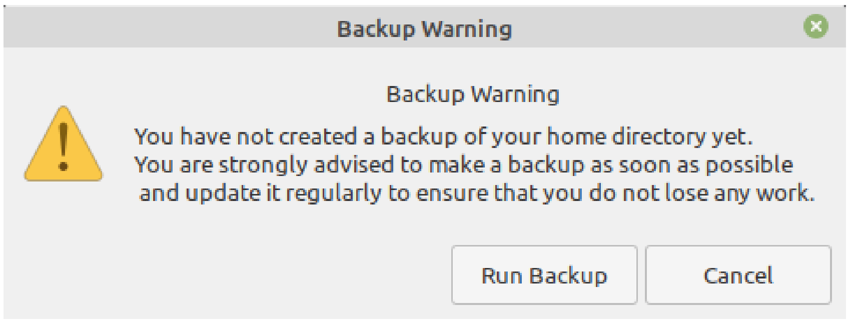
If you still need help then please contact support team at:
`support@cs.manchester.ac.uk`

Note: IT Services will not be able to assist with the image as it is a CS-specific local system.

5 VM Backup

Backing up your work is important, and it is good to get into the habit of making regular backups of your work in multiple locations. The VM contains a pair of programs to help with backing up your data from the VM to your Department of CS filestore, but you should probably take extra steps to keep your work secure in addition to the backup system.

Every time you log into the VM, a program will run to check whether a backup has been made, and how long ago the backup happened. If no backup has been made, or the last one was made over 5 days ago, you'll be prompted to make a new backup as shown in Figure 1.


Backup Warning

Backup Warning

You have not created a backup of your home directory yet.
You are strongly advised to make a backup as soon as possible
and update it regularly to ensure that you do not lose any work.

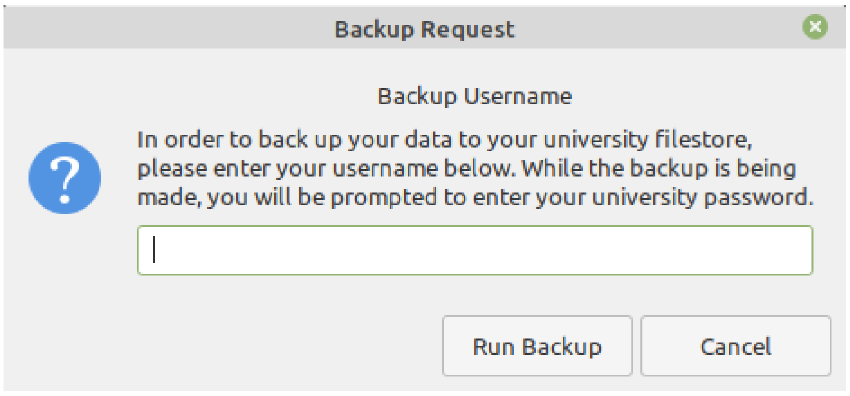
Figure 1
CSImage VM backup warning.

Note: Before you can backup your files to your Department of CS filestore, you must ensure that you have connected to the GlobalProtect VPN[w]. If you are not connected, the backup process will fail

For more information on how to connect to GlobalProtect VPN, visit the link:

`https://wiki.cs.manchester.ac.uk/index.php/CSImage_VM/VPN_Install#Connecting_to_the_VPN`

The first time you run the backup process, either by clicking on the "Run Backup" button in the backup check program window, or by double-clicking on the "CS Backup" icon on the desktop, you'll be prompted to enter your University of Manchester username (e.g. c12345ab) as shown in Figure 2.



Backup Username

In order to back up your data to your university filestore, please enter your username below. While the backup is being made, you will be prompted to enter your university password.

Figure 2
CSImage VM backup request.

For more information on backup and restoring files, visit the link:

`https://wiki.cs.manchester.ac.uk/index.php/CSImage_VM/Backup`

6 Reading your email

We use email extensively so it's vitally important that you read your University mail regularly – at least once a day (and probably much more often). Your standard University

email address is usually in the form firstname.lastname@studen
This address will be used for all communications to you from the department. You can read your mail using the Email button in the lefthand sidebar on your MY Manchester page:

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

7 Meet the Department's Wiki pages

One of the main purposes of this boot-up lab 0 is to introduce the main sources of information that you can use for getting help. Our department hosts the Computer Science Wiki. You can access the homepage by following the link below:

https://wiki.cs.manchester.ac.uk/index.php/Main_
Page

This wiki contains frequently asked questions (FAQs) that most probably you might find useful. In addition, there is information on software, and other useful information for students and stuff. The FAQs are split into sensible categories, feel free to try the search box in the top right hand corner, if you are not sure where to look.

You may also find particularly useful the General IT FAQ:
https://wiki.cs.manchester.ac.uk/index.php/StudentF

If you have any problems that are not academic-related and not answered on the wiki, please report them here: https://support.cs

It is highly recommended that you familiarise yourselves with the Department's Wiki pages.

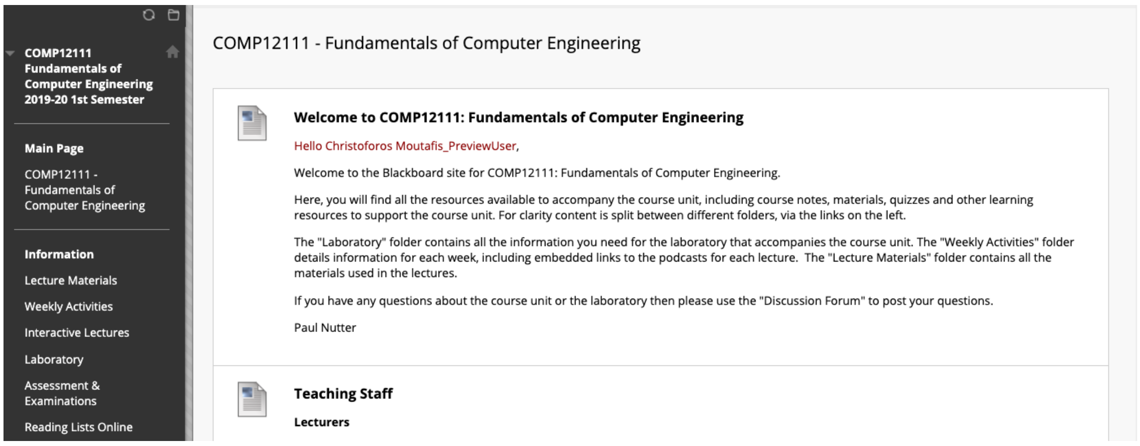
8 Blackboard

Blackboard is a a learning management system the the Department and the University utilises extensively in order to provide with easy and instant access to course and laboratory material. All of your courses/laboratories will have a Blackboard page and you can use it to get information about the course, coursework, learning material as well access to a discussion forum where you can ask questions and see answers to some of the questions your colleagues may have asked.

Think of it as your online portal for your courses!

Each of your course units you will have a Blackboard page where you will find all the resources available to accompany it. These resources usually include course notes, quizzes and other material and learning resources to support the course unit. In Figure 3 you can see an example preview of a Blackboard page for the COMP12111 unit. For clarity content is split between different folders, via the links on the left. For example, the "Laboratory" folder contains all the information you need for the laboratory that accompanies the course unit and the "Lecture Materials" folder contains all the materials used in the lectures. You may also have additional "Weekly Activities" folder with detailed information for each week, including embedded links to the podcasts for each lecture, as well as links to the various video podcasts per week.

In fact, in order to familiarise yourselves with Blackboard, each Boot-up lab will have an online quiz (on Blackboard of course!) that you need to complete. There will be no marking, but you are expected to complete it. Hopefully you will also find it fun!


COMP12111 Fundamentals of Computer Engineering 2019-20 1st Semester
Main Page
COMP12111 - Fundamentals of Computer Engineering
Information
Lecture Materials
Weekly Activities
Interactive Lectures
Laboratory
Assessment & Examinations
Reading Lists Online

COMP12111 - Fundamentals of Computer Engineering

Welcome to COMP12111: Fundamentals of Computer Engineering

Hello Christoforos Moutafis_PreviewUser,

Welcome to the Blackboard site for COMP12111: Fundamentals of Computer Engineering.

Here, you will find all the resources available to accompany the course unit, including course notes, materials, quizzes and other learning resources to support the course unit. For clarity content is split between different folders, via the links on the left.

The "Laboratory" folder contains all the information you need for the laboratory that accompanies the course unit. The "Weekly Activities" folder details information for each week, including embedded links to the podcasts for each lecture. The "Lecture Materials" folder contains all the materials used in the lectures.

If you have any questions about the course unit or the laboratory then please use the "Discussion Forum" to post your questions.

Paul Nutter

Teaching Staff

Lecturers

Figure 3
Example Blackboard view of a course (COMP12111).

9 Meet SPOT

In this section we will introduce SPOT, which is the portal the Department of Computer Science uses to release of your assessments for all of your courses. You can access your SPOT at:

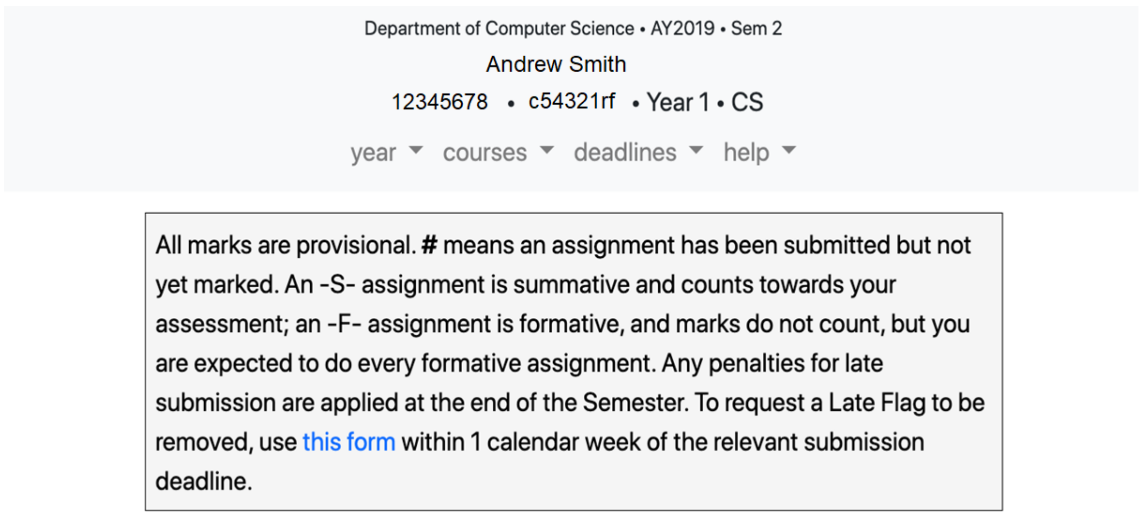
`https://studentnet.cs.manchester.ac.uk/me/spot/`

It gives you direct access to all of your coursework, all of your deadlines for each semester. In Figure 4 you can see an example student profile at SPOT.

As an example, in Figure 5 you see a list of the courses that you are signed up for. This can give you a quick overview of your courses and from this menu you can directly select the course that you are interested in (for example, in order to check your latest assessment!)

You might also find SPOT very useful as a tool to help you keep track of all of your assignments and deadlines, as you can see in Figure 6.

As soon as your assesments become available, they will appear in SPOT the next day. This is one of the main tools you will be using in your first term (and, in fact, all of your years in Manchester) and we think you will find it very useful!


Department of Computer Science • AY2019 • Sem 2

Andrew Smith

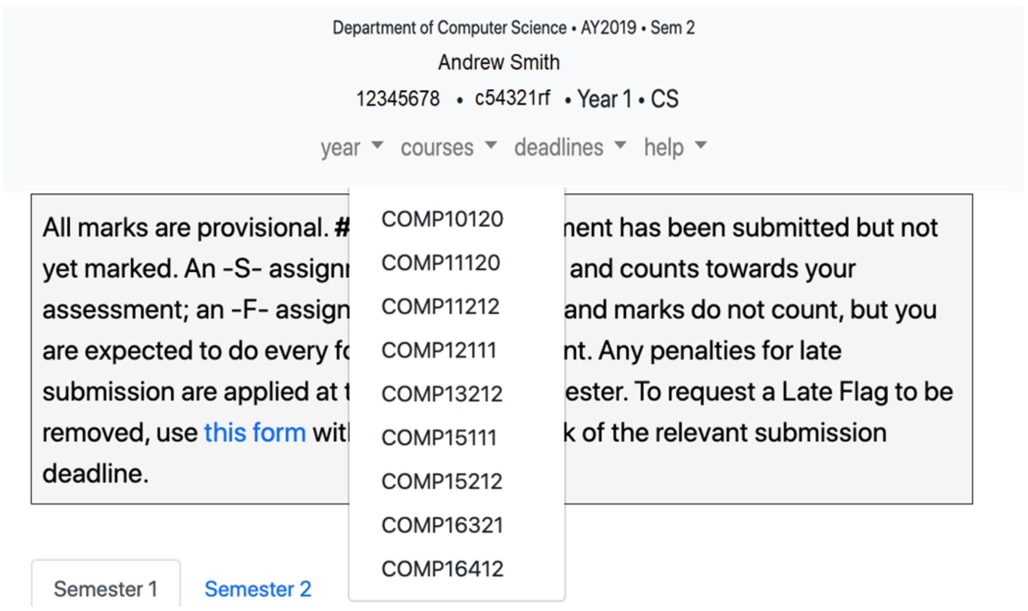
12345678 • c54321rf • Year 1 • CS

All marks are provisional. # means an assignment has been submitted but not yet marked. An -S- assignment is summative and counts towards your assessment; an -F- assignment is formative, and marks do not count, but you are expected to do every formative assignment. Any penalties for late submission are applied at the end of the Semester. To request a Late Flag to be removed, use this form within 1 calendar week of the relevant submission deadline.

Figure 4
Example SPOT profile.

10 Unix and Linux

Over the next couple of weeks you will be undertaking a number of introductory labs to familiarise yourself with the Department's computing infrastructure. Much of this is based on devices and machines running Linux, a variant of the Unix family of operating systems; here we provide some background on Unix and explains why we think it is important. It would very useful if you could read this before you the early labs, where the emphasis will be on leading you through a series of tasks to explore our setup.


Department of Computer Science • AY2019 • Sem 2

Andrew Smith

12345678 • c54321rf • Year 1 • CS

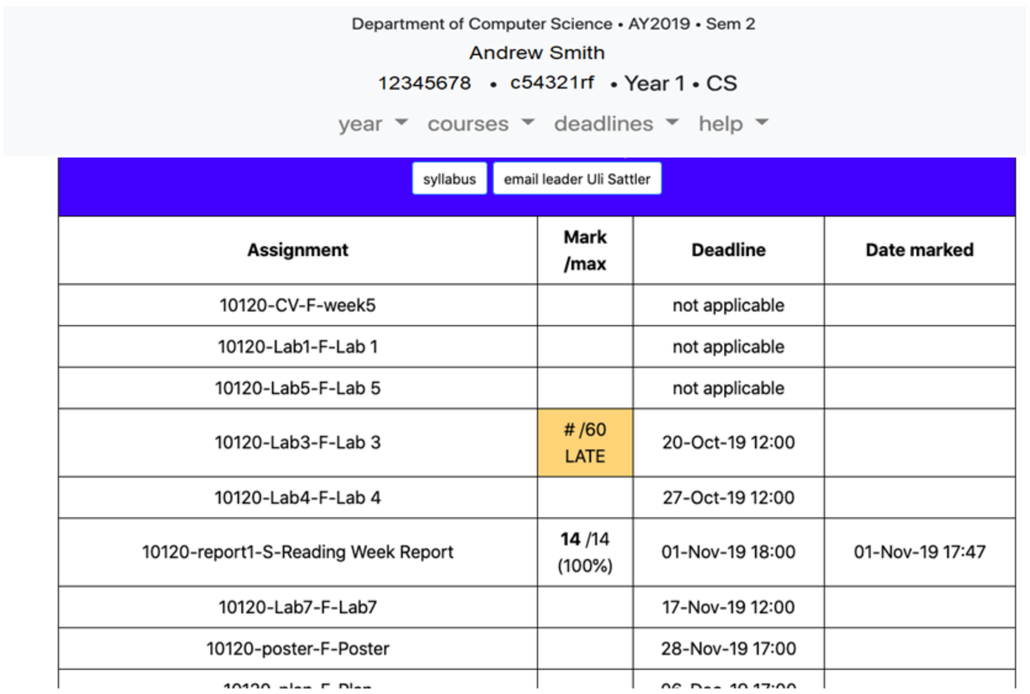
All marks are provisional. # | nent has been submitted but not yet marked. An -S- assignı | and counts towards your assessment; an -F- assign | and marks do not count, but you are expected to do every fo | nt. Any penalties for late submission are applied at t | ester. To request a Late Flag to be removed, use this form wit | k of the relevant submission deadline.

COMP10120
COMP11120
COMP11212
COMP12111
COMP13212
COMP15111
COMP15212
COMP16321
COMP16412

Figure 5
Example list of courses.

10.1 Operating Systems

An **operating system**w (OS) is a suite of software that makes computer hardware usable; it makes the 'raw computing power' of the hardware available to the user. You're probably most familiar with the **Microsoft Windows**w and Apple **macOS**w operating systems for 'desktop' computers, and **iOS**w (Apple, again) and Google's **Android**w for mobile devices; but many other more specialist operating systems exist, and you'll be studying some of these and the principles that underpin OS design in COMP15212 in Semester 2 of your first year. In the meantime, a potted history of OS development will tide us over…


Department of Computer Science • AY2019 • Sem 2

Andrew Smith

12345678 • c54321rf • Year 1 • CS

Assignment	Mark /max	Deadline	Date marked
10120-CV-F-week5		not applicable	
10120-Lab1-F-Lab 1		not applicable	
10120-Lab5-F-Lab 5		not applicable	
10120-Lab3-F-Lab 3	# /60 LATE	20-Oct-19 12:00	
10120-Lab4-F-Lab 4		27-Oct-19 12:00	
10120-report1-S-Reading Week Report	**14** /14 (100%)	01-Nov-19 18:00	01-Nov-19 17:47
10120-Lab7-F-Lab7		17-Nov-19 12:00	
10120-poster-F-Poster		28-Nov-19 17:00	
10120-plan-F-Plan		06-Dec-19 17:00	

Figure 6
Example view of a SPOT profile, where you can track assignments and deadlines.

10.2 Unix Origins

In the late 1950s, an American company called **Bell Laboratories**ᵂ decided that they needed a system to improve the way they worked with their computer hardware (it's probably quite hard to imagine what interacting with a computer *without* an operating system might be; but it wasn't pretty and involved manually loading and running programs one by one). Together with the **General Electric Company**ᵂ and the **Massachusetts Institute of Technology**ᵂ, they set about the design of an operating system they called **Multics**ᵂ: the

'Multiplexed Information and Computing Service'. Multics was hugely innovative, and introduced many concepts that we now take for granted in modern operating systems such as the ability for more than one program to run 'at once'; but it did rather suffer from 'design by committee', and the final system was seen at the time as being overly complex and rather bloated ('bloated' is all a matter of perspective of course: it's sobering to realise though that the entire Multics operating system was only around 135Kb. Today's operating systems are something like 30,000 times this size...). In the late 1960s, a group of programmers at Bell Labs created a cut-down, leaner and cleaner version of Multics that would work on more modest hardware. Legend has it that this was to allow them to play their favourite (only!) computer game, **Space Travel**W. In an early example of the trend of giving things 'punny' names, to contrast with the more clumsy Multics, they called this new system Unix. The so-called **Jargon File**W is a good source of explanations of various bits of computer slang and their obscure origins, and is well worth a read: in part to give some background history, but mostly as an insight into the minds of the computing pioneers of the past!

Even though Unix is now quite old, most Computer Scientists recognise that the designers of Unix got most of the fundamental concepts and architecture right. Given how much computing has changed since the 1960s, this was an astonishing intellectual achievement. Although Microsoft's **Windows**ᵂ is by far the most common operating system on *desktop* machines, the majority of the Internet, much of the world's corporate infrastructure, virtually all supercomputers, and some mobile devices are powered by Unix-like operating systems. So, while the polished graphical user interfaces of Windows and **macOS**ᵂ appear to dominate the world of computing, most of the real hard-core and leading-edge computation relies on an elegant operating system designed nearly 50 years ago (by a team of scientists who wanted to play a game).

10.3 Modern Unix Variants

The history of Unix is complex and convoluted, with the system being updated, re-implemented, and mimicked repeatedly over the years, primarily by commercial companies who guarded their versions jealously. Figure 7 shows a tiny fragment of the Unix's 'family tree' (the full diagram, which you can find at `www.levenez.com/unix/unix.pdf`, is *many* times the size of the portion you can see here).

Although many of the branches represent interesting innovations of one kind or another, there are perhaps two that deserve particular attention. The first of these was the decision by Apple some time around the turn of the millennium to drop their own – highly popular, but ageing – bespoke operating system (**Mac OS 9**ʷ) in favour of a Unix-based system (now the more familiar 'macOS'. Although the majority of Mac users are blissfully unaware of the fact, behind the slick front-end of macOS sits a variant of Unix). The second, and perhaps more profound of these events was the creation in 1991 by Swedish programmer **Linus Torvalds**ʷ of a Unix-like system, the source code to which *he gave away for free* ('free' here in the sense both of 'freedom to reuse or adapt', and also in the sense of 'without charge'.);

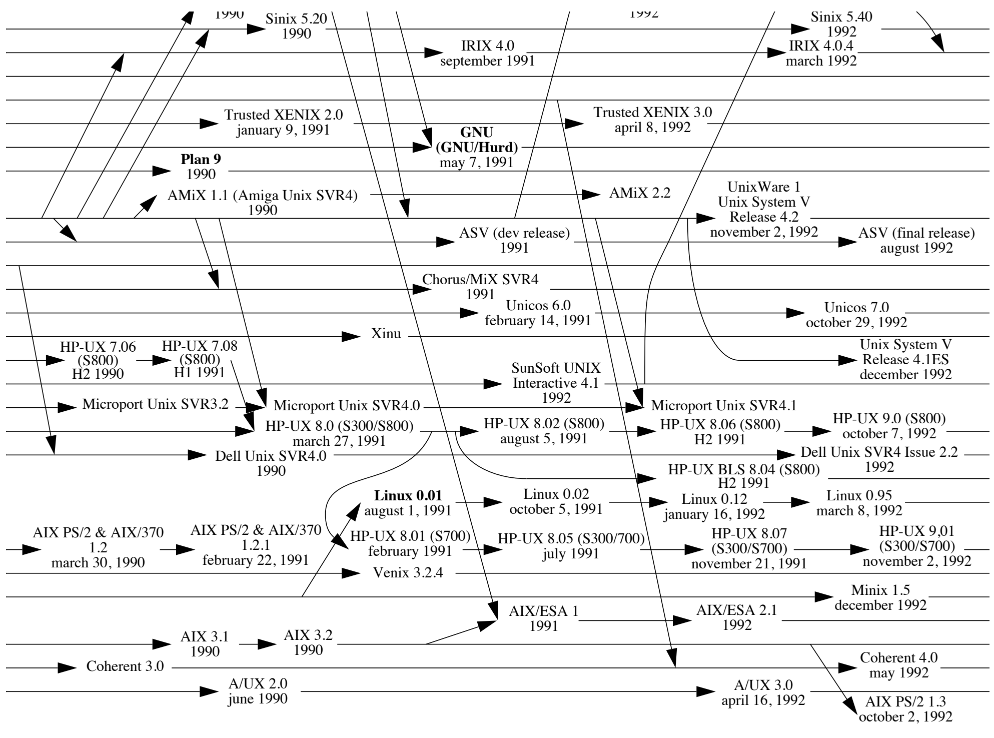
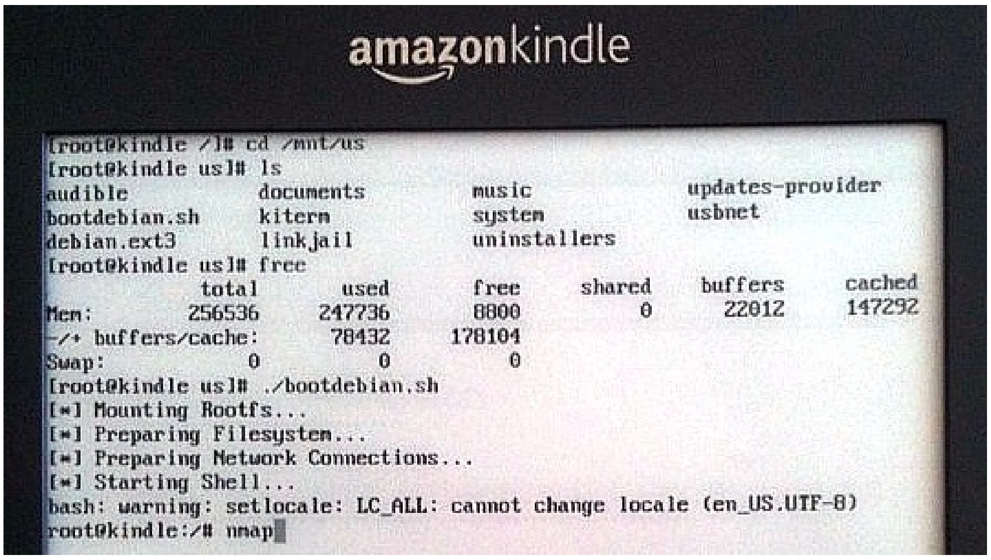

1990
Sinix 5.20 1990
1992
Sinix 5.40 1992
IRIX 4.0 september 1991
IRIX 4.0.4 march 1992
Trusted XENIX 2.0 january 9, 1991
Trusted XENIX 3.0 april 8, 1992
GNU (GNU/Hurd) may 7, 1991
Plan 9 1990
AMiX 1.1 (Amiga Unix SVR4) 1990
AMiX 2.2
UnixWare 1 Unix System V Release 4.2 november 2, 1992
ASV (dev release) 1991
ASV (final release) august 1992
Chorus/MiX SVR4 1991
Unicos 6.0 february 14, 1991
Unicos 7.0 october 29, 1992
Xinu
Unix System V Release 4.1ES december 1992
HP-UX 7.06 (S800) H2 1990
HP-UX 7.08 (S800) H1 1991
SunSoft UNIX Interactive 4.1 1992
Microport Unix SVR3.2
Microport Unix SVR4.0
Microport Unix SVR4.1
HP-UX 8.0 (S300/S800) march 27, 1991
HP-UX 8.02 (S800) august 5, 1991
HP-UX 8.06 (S800) H2 1991
HP-UX 9.0 (S800) october 7, 1992
Dell Unix SVR4.0 1990
Dell Unix SVR4 Issue 2.2 1992
HP-UX BLS 8.04 (S800) H2 1991
Linux 0.01 august 1, 1991
Linux 0.02 october 5, 1991
Linux 0.12 january 16, 1992
Linux 0.95 march 8, 1992
AIX PS/2 & AIX/370 1.2 march 30, 1990
AIX PS/2 & AIX/370 1.2.1 february 22, 1991
HP-UX 8.01 (S700) february 1991
HP-UX 8.05 (S300/700) july 1991
HP-UX 8.07 (S300/S700) november 21, 1991
HP-UX 9,01 (S300/S700) november 2, 1992
Venix 3.2.4
Minix 1.5 december 1992
AIX/ESA 1 1991
AIX/ESA 2.1 1992
AIX 3.1 1990
AIX 3.2 1990
Coherent 3.0
Coherent 4.0 may 1992
A/UX 2.0 june 1990
A/UX 3.0 april 16, 1992
AIX PS/2 1.3 october 2, 1992

Figure 7
A fragment of Éric Lévénez's Unix History chart, reproduced with permission and showing the beginnings of Linux in amongst other versions of Unix.

this became known as the **Linux Kernel**W. Combined with other free software created by the **Free Software Foundation**W, a non-commercial version of Unix called **GNU/Linux**W was born (GNU here is a recursive acronym for "GNU's not Unix", a swipe at other commercial non-Free versions; much to the annoyance of the Free Software Foundation, GNU/Linux is almost always called just 'Linux' (pronounced "Linn-ucks", see `https://www.youtube.com/watch?v=c39QPDTDdXU`).

Linux has been, and continues to be, developed cooperatively by thousands of programmers across the world contributing their effort largely free of charge (although many are now paid to work on Linux as part of their job). It is amazing to think that such a project could ever happen – and it is surely a testament to the better side of Human Nature. But what is interesting is the observation that these programmers are not motivated by commercial concerns, but by the desire to make good reliable software and have it used by lots of people. Thus, Linux is a good choice of Unix: it's Free, it's efficient, and it's reliable, and it is now used by large corporations, governments, research labs and individuals around the world. Even Google's **Android**W platform is a Linux-based mobile OS, and the **Amazon Kindle**W is also a Linux box behind the electronic ink of its user interface (Figure 8).

One of the results of the fact that Linux is Free is that sev-



```
[root@kindle /]# cd /mnt/us
[root@kindle us]# ls
audible          documents        music            updates-provider
bootdebian.sh    kiterm           system           usbnet
debian.ext3      linkjail         uninstallers
[root@kindle us]# free
             total       used       free     shared    buffers     cached
Mem:        256536     247736       8800          0      22012     147292
-/+ buffers/cache:      78432     178104
Swap:            0          0          0
[root@kindle us]# ./bootdebian.sh
[*] Mounting Rootfs...
[*] Preparing Filesystem...
[*] Preparing Network Connections...
[*] Starting Shell...
bash: warning: setlocale: LC_ALL: cannot change locale (en_US.UTF-8)
root@kindle:/# nmap
```

Figure 8
A photograph of Liraz Siri's 'rooted' kindle, showing the Linux command prompt. Reproduced with the author's kind permission from `www.turnkeylinux.org/blog/kindle-root`

eral organisations and companies have created their own distributions of it; these vary a bit (in fact, anybody is free to make any change they like to Linux, and pass it on to whoever wants it).

So, if you are to become an expert computer professional, it is important that you understand the theory and practice of Unix-based systems. Learning Unix is not only a crucial skill for any serious computer scientist, it is a very rewarding experience; the labs over the next couple of weeks are designed to help you become familiar with what will be your daily working environment.

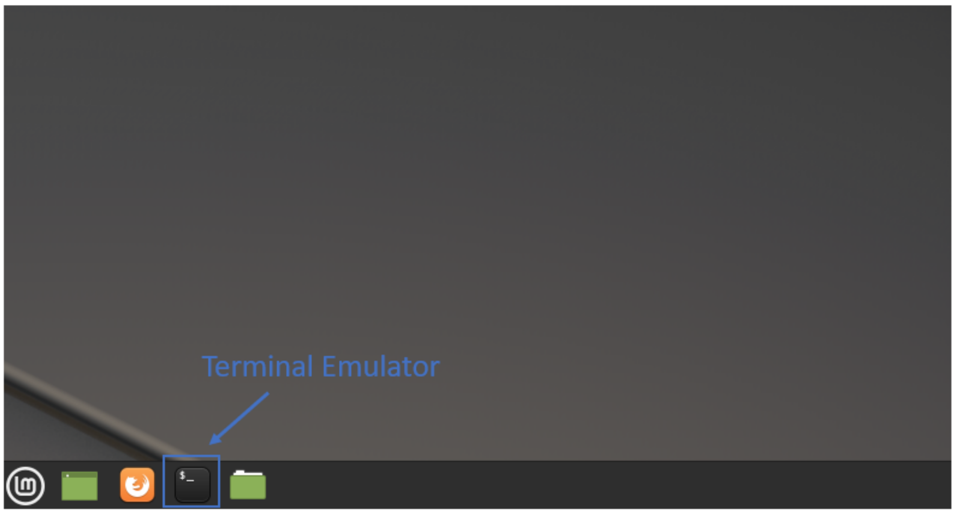


Figure 9
The Terminal Emulator window.

11 Using the VM Image

If all went well in the last lab session, your VM image should be ready for use, but if you did not complete the VM image installation in Section 4 please go back to the Boot-up Lab 0 notes and finish all the lab tasks now. **Otherwise you will not be able to complete this lab**.

In this session we are going to explore some of its capabilities. Please bootup your VM as you did in the previous lab. Open a `Terminal Emulator` by clicking on a small black screen icon at the bottom left as shown in Figure 9 or simpling pressing `Clt+Alt+T` on your keyboard.

The first line of the text (which is shown in bold here but will be green and blue on your screen) is the **command prompt**[w].

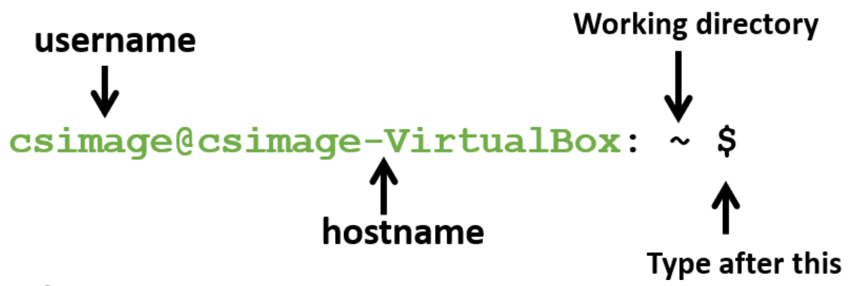


Figure 10
The different components of the VM's default command prompt.

It might look innocent enough, but in the right hands, the command prompt is one of the most powerful ways of controlling a computer. Your previous interaction with a computer was probably via a **graphical user interface**^W, or GUI, such as that provided by Windows or macOS, so it may feel a bit odd at first to be issuing instructions to a machine via a textual command-line. However, this is a crucial skill that you'll need during your studies here at University, and also in your future career. In fact, employers have often said that our students' abilities with the command-line come as a very pleasant surprise to them and set them apart from many other students.

The default command prompt on the `Terminal Emulator` consists of four components as shown in Figure 10.

- To the left of the `@` symbol is the name of the user, and in this case that's `csimage`, since you've just logged in under that name.
- To the right of the `@` is the hostname of the machine,

which on a VM is quite reasonably set to `csimage-VirtualBox` by default.

- The ~ tells you where in the VM's file system you're currently working. We'll explain this in a lot more detail later on, for now all you need to know is that the ~ symbol is called a *tilde* (pronounced something like till-duh, though it's often referred to colloquially just as a 'twiddle'), and is used here to refer to the 'home' of the current user.
- On the VM the default prompt ends with the $ symbol.

You can change this prompt to something more or less verbose later, but for now we'll leave it as it is. For simplicity in these notes, we'll use the $ symbol from now on to mean 'type something at the command prompt and press Enter'. So for example

```
$ echo Hello World
```

means 'type `echo Hello World` at the command prompt and then press Enter' (you can do this if you like; the result will be that 'Hello World' gets 'echoed' back to you on the next line of the screen). (Notice that this reference to a Unix command has caused a small cog to appear in the right hand margin, so you can find this reference easily later. If you are

⚙ **echo**

reading the notes online, the cog also contains a link to a web page describing some Unix commands.)

To confirm that you're now connected to the network, use the `ping` command, which sends a low-level network message to a designated place and checks for a response, to see if you can reach our Department's web server.

ping

```
$ ping www.cs.manchester.ac.uk
```

You should see output like:

```
PING cs2.eps.its.man.ac.uk (130.88.101.49) 56(84) bytes of da
64 bytes from eps.its.man.ac.uk (130.88.101.49): icmp_req=1 t
64 bytes from eps.its.man.ac.uk (130.88.101.49): icmp_req=2 t
64 bytes from eps.its.man.ac.uk (130.88.101.49): icmp_req=3 t
```

Each of the lines starting with '`64 bytes`' represents a short response from the machine you've just pinged, and you're shown the round-trip time for ping's data to leave your VM, find (in this case) `www.cs.manchester.ac.uk` on the network, and return back to your VM. Since we're just using ping here to give us some confidence that the network is okay, we don't need to leave it pinging away for ages, so let's stop the ping command in its tracks. Hold down the control key (marked 'Ctrl' on the keyboard), and press 'c'. This will signal the currently executing command that it should stop what its doing and return to the command prompt (quite often this is referred to as "control-c-ing" a command, and it will have the same effect on the majority of command-line tools). This key combination is often written `<ctrl>c`. **If you see this notation in these notes, don't type the individual characters, press the key marked ctrl and the appropriate letter key**.

The readout from the ping command is important - you should check which machine exactly is responding to your pings. If your network is configured correctly you should find that the output from the ping command looks similar to the one here. You shouldn't worry about subtle differences such as the time.

12 Processes and the Unix Shell

Before doing anything else, let's take a few steps back and look in a bit more detail at what you've just done; you may be surprised how much stuff happened as a result of that simple command you've just typed.

The first concept you'll need to understand is that you have been interacting with what is known in Unix circles as a **shell**[W]: a program that prompts the user for commands, accepts them, executes them and displays the results. A shell is just a program running on the Linux operating system like any other program – it's not 'built in' to the computer or the operating system in any special way, it just happens that by default, the VM is set up so that when a user logs in, the first program that gets executed on behalf of that user is an interactive shell that allows users to execute further programs themselves. The shell that we are using here is called `bash`, a name we will explain a little later.

But what do we mean by 'execute commands'? And if the shell is 'just a program', how does it get to communicate with the keyboard and screen? What is a 'command' any-way, where do commands come from?

To understand what's going on here you'll need to make sense of a concept that's fundamental to pretty-much any operating system; that of a **process**w. As you no doubt know, modern computers have one or more **Central Processing Units**w (CPUs) which are capable of carrying out simple instructions; a basic computer will have a single CPU, whereas a big server machine or supercomputer may have several tens of CPUs in a single box. To a first approximation, each CPU is only capable of following one instruction at a time, and the illusion that a computer is capable of doing a very large number of things simultaneously (e.g. streaming music, displaying web pages, downloading a video of a unicycling kitten and playing **Minecraft**w) is achieved by the operating system arranging for each of these tasks to be given access in turn to the CPU for a tiny fraction of a second. More technically, these tasks are called **processes**w. The relationship between anything that you as a user may recognise – for example a desktop application – and what's happening in the operating system in terms of processes is quite complex, since many applications are made up of several processes, and there will be a whole load of other processes doing housekeeping jobs that aren't immediately obvious to a user. But for now we'll gloss over this detail and work on the assumption that when you ask a computer to do something for you, a process will be started to deal with that task for you.

In terms of what's just happened when you ran the `ping` command a moment ago, there are at least two processes involved. The shell program itself is a process that's waiting for you to type something at the keyboard; when you pressed enter after having typed your command, the shell interpreted your input, and started up a second process to run the `ping` program for you. It handed access to the keyboard and monitor over to the process running `ping`, and then went to sleep briefly to wait for the `ping` command to finish. When `ping` finished (in this case because you aborted it), the shell woke up again, took back control of the keyboard/screen, and was ready for your next instruction. This is illustrated in Figure 11.

You will learn much more about processes and the way they are managed in COMP15212 Operating Systems; for now there's one more thing you need to understand about the relationship between processes.

12.1 The process 'family tree'

When the shell's started up a process to run your command, the new process (the one that's running the command) is thought of as a 'child' of the shell's process. The child inherits many of the properties of its parent (you'll see why this is important in the next lab).

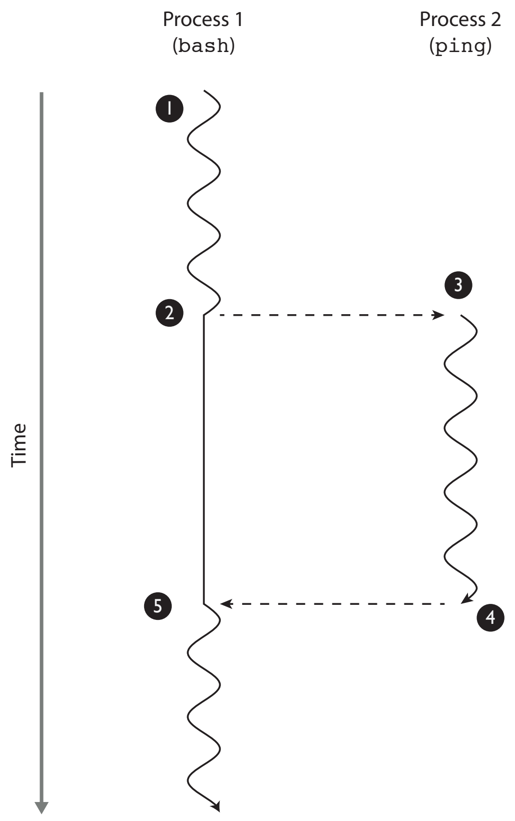


Figure 11
Running a command at a bash shell involves two process, one for the bash shell itself, and a second child process that is started by the bash shell in which to run the command: ❶ initially, just the bash process is running. ❷ at the point where you type the ping command and press enter, bash starts a second process and hands over its input and output to that new process, which executes the command on your behalf; at this point the bash process continues to exist but 'goes to sleep' until the command finishes. ❸ the new process starts up, and executes your command, until at ❹ it's either aborted by the user or finishes what it's doing, at which point ❺ the ping process terminates and hands back control to the shell, which wakes up ready to accept the next command.

13 File systems and files

Next we're going to explore the VM's filesystem a little. You'll be familiar with the idea of a hierarchy of files and folders from whatever graphical environment you're used to using on desktop or mobile devices: files represents things that you've created or downloaded such as documents, images or movies, and folders are a way of organising these into related collections. By putting folders inside folders, you can organise your stuff starting with general concepts such as 'Photographs' and ending up with much more specific collections, e.g. 'Holidays', then 'Bognor Regis 2018'.

Interacting with a standard Unix filesystem via the command-line uses similar concepts (actually, it's the graphical environment that's being 'similar' here really, since the Unix command-line existed quite some time before anything graphical appeared). Files are called files, but what are commonly represented as 'folders' in graphical environments are more correctly called 'directories' when we are operating at this level (and we'll call them directories from now on, because it'll make some of the command names and manual pages make more sense).

Let's first see what stuff we already have on our VM. The `ls` command lists files and directories. Type it now, and you should see some directories that have already been created for you. Type: ⚙ **`ls`**

```
$ ls
```

Now let's make a new directory, using the `mkdir` command, which we'll explain further later on. Type:

```
$ mkdir python_games
```

Then type `ls` again, look at the list of names printed and check that you can see `python_games`. If not, check you did the `mkdir` correctly, and ask for help if your're stuck.

When we're using a command-line prompt, we have the notion of **current working directory**[W] which is the directory that we're currently 'in' (so in this example, using `ls` like this really meant 'run the list command on my current working directory'). There are numerous Unix commands that allow you to move around the filesystem's directory structure, and it's very important that you become familiar with these early on.

Let's say we want to look at the contents of the `python_games` directory. There are several ways of doing this, but for now we'll break the process down into simple steps. Use the `cd` command to Change Directory to `python_games`:

⚙ **cd**

```
$ cd python_games
```

Here we have a **command**, `cd`, together with an **argument**, `python_games` which specifies the object on which the command is to operate.

Now look at what's happened to the command prompt. Whereas before it was just

```
csimage@csimage-VirtualBox:~$
```

it has now become

```
csimage@csimage-VirtualBox:~/python_games$
```

to indicate that we've changed our current directory to `python_games` (remember the `~` symbol means 'home directory', so `~/python_games` really means 'a subdirectory called `python_games` which is in my home directory').

Now use the `ls` command to list the contents of our new current working directory. You will see that this directory is empty as you have created it.

We actually want to play a game that isn't there, so we'll need to get it. We've put a copy of the game on the web at:

```
http://syllabus.cs.manchester.ac.uk/ugt/COMP10120/
files/worms101.py
```

Rather than browsing the web in the normal way, we sometimes want to just obtain a copy of file; we can do this using the command `curl`. Type the following:

☼
`curl`

```
$ curl http://syllabus.cs.manchester.ac.uk/ugt/COMP10120/files/worms101.p
```

and you'll see `curl` fetch the file you need 'over the web' and save it in your home directory. This is also the first time you've encountered what's called a command-line argument **switch**: the `-o` switch tells `curl` to use the next argument on the command-line as the output filename. Check that the file has arrived by using `ls`:

```
$ ls
```

Now, at the prompt type

```
$ python3 worms101.py
```

to start a simple version of the classic 'Snake' game. You can guide your green snake around the screen with the cursor keys; you score a point every time you eat one of the red squares and an extra segment gets added to the length of your snake. The game finishes if you crash into the edge of the screen or eat yourself. Once you've convinced yourself this is working (don't spend too long playing the game!), press the Escape key to return to the command prompt.

Breakout 1: Don't be a WIMP

The familiar Windows, Icons, Menus and Pointer (WIMP) paradigm used on most graphical desktop environments is enormously powerful, but it's not suitable for every task, and understanding when you're better off using the command-line or a keyboard shortcut instead will make you a lot more efficient.

Sometimes the clumsiness of the GUI comes from the fact that there's no convenient visual metaphor for a particular action; how do you graphically represent the concept of 'rename all the files I created yesterday so they start with a capital letter'?

But a lot of the time the issue is simply that it takes much longer to do some things with the mouse than it does with a keystroke or two. Every time you use the mouse, a little time is wasted shifting your hand off the keyboard and a little more time used up tracking the pointer between the on-screen widgets. For casual use, this wasted time really doesn't matter. But as a computer scientist you're going to be spending a lot of time time in front of a machine, and all the seconds wasted moving the mouse pointer around add up.

What's really fascinating here, though, is that although the keyboard versus mouse debate is one that has been running since at least the mid-1980s, there isn't a clear winner, or even any definitive guidelines as to when one is better than the other.

In any case, you should definitely learn the keyboard shortcuts for the most common operations in your favourite tools, and a handful of useful command-line tools. For example, when you're writing code you'll be saving files very regularly; maybe even several times a minute when you're debugging. There are two options

Breakout 2: Ping

The `ping` command is named after its use in the field of active sonar, where an actual 'ping' sound was sent through water, and distance calculated by listening for the echo to return. It's the classic boingy pingy noise associated with submarine movies!

Breakout 3: Unix shells

Unix has many shells: the first shell was called the 'Thompson' shell (also known as just 'sh', and pronounced "shell"), written by Ken Thompson for the first Unix system; then came the 'Bourne' shell (also called 'sh'), written for a later commercial version of Unix by Stephen Bourne. You have just been using the Free Software Foundation's 'Bourne Again' shell (a pun-name taking a dig at its commercial fore-runner), or 'bash'. The various different shells offer the user different facilities: 'sh' is rather primitive compared to the more modern ones. However, their basic functionality is always the same: they accept commands from the **standard input** (for now, we can treat that as meaning 'the keyboard'), execute them, and display the results on the **standard output** (i.e. for now 'the screen', which in this case was the entire screen, or **console**). Shells repeat this process until they have reached the end of their input, and then they die. Unix shells are rather like **Command Prompt** windows in Microsoft Windows, except that Unix shells are considerably more sophisticated.

13.1 The Unix filesystem

In Unix, as with most other operating systems, the files and directories you create can have more or less any name you like. It is very sensible to give them names which mean something and make their purpose clear. This is despite some of the traditional file names in Unix being rather cryptic – this is particularly true for most Unix commands. You'll get used to that.

File name formats. The filesystem on your VM (which uses a type of filesystem called 'ext4') is *case sensitive*, which means that `Hello.txt` and `hello.txt` are treated as different files because they have different case letters in their names. The filesystem used by Microsoft Windows since XP (called 'NTFS') is also case-sensitive. Apple's macOS/iOS uses (since High Sierra and iOS 10.3)'APFS' which is case-sensitive on iOS, and can be configured as case-sensitive/not case-sensitive on macOS. The FAT32 filesystem, used on most removable USB drives, is not a proper case-sensitive file system; although it will remember whether you called a file `Hello.txt` or `hello.txt` so files *appear* to be case sensitive, the OS itself treats them as being *the same file*!

Most of the time this complexity isn't a problem, but you should be careful of the effects when copying files from one filesystem to another, especially if you are using a USB drive to transfer files from a Linux box to somewhere else. For example, if you have two files in the same directory on Linux with the same name but with different capitalisation, one file will overwrite the other when you copy them onto your USB drive (and which one survives will depend on the order in which they are copied). One way around this problem is to use commands such as `tar` or `zip` to bundle the files up into a single archive file, and then transfer that via the USB drive.

☼ **`tar`**

As you already know, directories are created within directories to create a hierarchical structure. The character `/` (slash) is used to separate directories, so if we wish to talk about the file `y` within the directory `x` we write `x/y`. Of course, `y` may itself be a directory, and may contain the file `z`, which we can describe as `x/y/z`.

But where is the 'top' directory? Surely that has to go somewhere? On each machine, there is one directory called '/' which is referred to as the *root*, which is not contained in any other directory. All other files and directories are contained in root, either directly or indirectly. The root directory is written just as `/` (note this is the opposite slanting slash character to that used in Windows).

One really important, and slightly strange thing to get used to, though, is that computer science trees grow upside down, so although we call them trees, the 'root' is usually thought of as being at the top, and the 'leaves' at the bottom. You'll hear phrases like 'go up to the root directory, and then down one level to the home directory'. We normally think of the root directory as being at the top of the tree. (For those of you who are interested: Unix actually allows links, which means the structure can really be a **cyclic graph**W. Links are similar to, but fundamentally not the same thing as, short-cuts in Windows.)

Apart from `/` there are two more directories of special note:

- Your *home directory* is the directory where you find yourself when you log in. It might be tempting to think of this as being the 'top' of the tree (and for every-day purposes thinking this way is probably okay), but in reality your home directory is probably one or more levels away from the root of the file system. We'll find out where this is on the VM shortly.
- Your *current working directory* is the one you are working in at a particular moment.

So let's see where we are in the VM's filesystem at the moment. Assuming you're following these instructions properly you should still be in the `python_games` directory (check the command prompt to confirm this is the case). To go back to our home directory, we can use the `cd` command without an argument:

```
$ cd
```

You should see the command prompt change back to being as it was when we first logged in. So where in the filesystem is our home directory? We can find out where we currently are using the `pwd` command, which stands for Print Working Directory:

```
$ pwd
/home/csimage
```

So apparently we're in a directory called `/home/csimage` which sounds plausible enough. Notice that the `pwd` command has returned us an **absolute pathname**[w], that is, it starts with a `/` character. Absolute paths, such as `/home/csimage` are given by their 'steps' from the root. So we now know that the home directory for the user `csimage` is in a directory called `home` which itself is a subdirectory of the root `/`.

⚙ **pwd**

Let's confirm that this is true. Issue the command:

```
$ cd /
```

which just means 'change directory to the root directory' and use the `ls` command to look at the root directory's contents. You'll see several directories with names like `bin`, `boot`, `dev` and `lib`. Most of these contain 'housekeeping' files for the operating system, and at this stage you don't need to know what's in them (Appendix **??** gives you a brief description if you're interested). The one called `bin` is quite interesting though, so let's investigate that by typing:

```
$ cd bin
$ ls
```

Here the argument to `cd` is not an absolute pathname (starting with `/`) but a **relative pathname** (not starting with `/`). A relative pathname can be thought of as a 'pathname starting from here', where 'here' means the current working directory. So, because we are in `/`, the relative pathname `bin` refers to the directory `/bin`.

The above commands should have produced a fairly long list of files. Look carefully, and you'll find two names that you recognise: `ls` and `pwd`. Files like these are called **binary executable** files and are the programs run when the commands are used (which is why they are in the 'bin' directory, which is short for binary). In Unix, most commands are not 'built into the system', but are just programs put in a special place in the filesystem that are picked up by the shell when you type things. This makes Unix very easy to extend with new features; you just write a program to do what you want, and put it in the right place. We'll look at how the system knows where to find commands later, and explore several of the other commands you can see in this directory as well.

We now need to get back to our home directory. There are many ways of doing this, including the following:

1. `cd` on its own means 'take me directly to my home directory'
2. We know that the tilde symbol also means 'my home directory', so `cd ~` will also work (though at the expense of two extra keystrokes!)
3. We could go back to the root directory by first typing `cd /`, then `cd home` and finally `cd csimage`
4. We could go back to the root directory by first typing `cd /`, then `cd home/csimage`
5. We could go straight from where we are now (which is `/bin`, remember) by typing `cd /home/csimage` – use this method now

Now we're back in our home directory (check the command prompt to make sure), you may have noticed that our commands for navigating around the filesystem are missing one feature. We can go to the 'top' of our home directory easily enough; and we can go straight to very top of the whole filestore using `cd /` ; and we know how to descend into a subdirectory (e.g. `cd python_games`). But how do we go up one level? If we were in some nested subdirectory several levels below our home, and wanted to go just one level back up the tree, it would be very tedious to have to start back at our home directory and traverse back down to where we wanted to be.

Unix represents the idea of going 'up' one directory with the `..` symbol (that's two fullstops typed immediately after one another with no spaces, usually pronounced 'dot dot'.). So if you are in, say, `python_games` and want to go *back up* to the directory above, you could type:

```
$ cd ..
```

We use `..` whenever we we want to specify a move up the directory tree. So, assuming you are still in `python_games`, how could you get into `/bin` using only a *relative* path in the `cd` command? Yes, this is a slightly artificial exercise because the simplest solution would just be to use the absolute path `cd /bin`, but just go with the flow for now and work it out using a relative path. The answer is shown in Figure 12.

We can also refer to the current directory in a similar manner. Just as the directory above is referred to as `..`, the current directory can be referred to as `.`, or 'dot'. So the relative pathname `x/y/z` refers to exactly the same place as `./x/y/z`

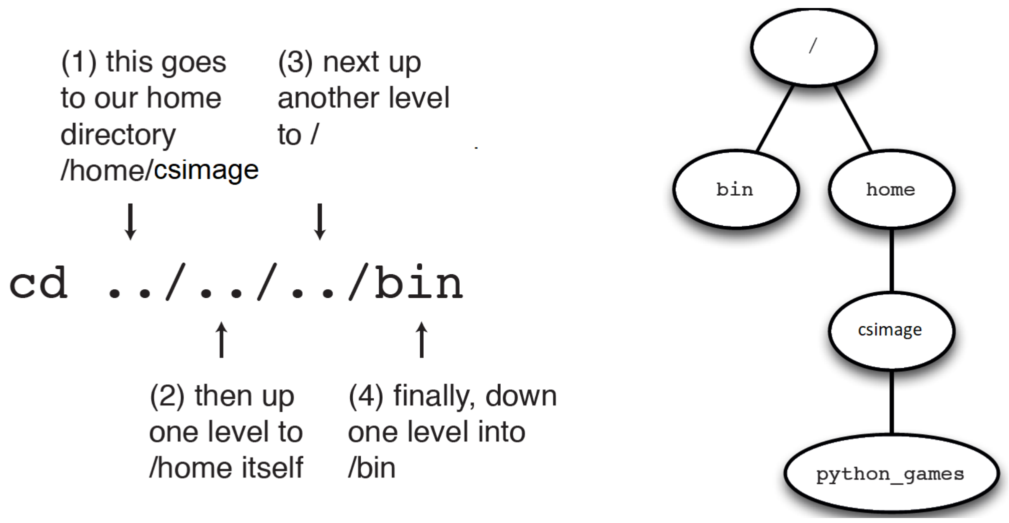

(1) this goes to our home directory /home/csimage
(3) next up another level to /
cd ../../../bin
(2) then up one level to /home itself
(4) finally, down one level into /bin
/
bin
home
csimage
python_games

Figure 12
Given the VM's default filesystem structure, starting in the `python_games` directory, the command `cd ../../../bin` will take you to the `/bin` directory (by an admittedly tortuous route!)

13.2 Files in more detail

The files we've looked at so far, and the behaviour of the commands we've used to manipulate them shouldn't feel too alien – we're just doing similar things on the command-line to actions that you'll have performed using a graphical interface before. But the Unix take on files is rather more sophisticated than this, and to understand that, we'll need to think in a bit more detail about what a file actually is. When we think of an image or a music track as being stored 'in a file', what do we actually mean?

A file is actually just a sequence of numbers on some storage device such as a hard drive. The right program can interpret these numbers, and turn them into pictures on screen, or sounds coming out of your speakers. The wrong program would be able to make no sense of a file at all; if you could 'display' an audio track on screen, or 'play' an image file as sound, it would just be a meaningless mess. So there's nothing very special about a file that makes its contents mean one thing or another; it's just up to a program to interpret what's in the file correctly. So what are the properties of a file? So far we've seen that files have a filename, and a location within a file system. We've seen that some files can be executed, whereas other files contain data. But in both these cases, files are just sequences of numbers. Although we've not explored this yet, some files can be written to or modified, whereas others can only be read from; but they are still just sequences of numbers which when interpreted correctly have a particular meaning.

The designers of Unix exploited this idea to create a very elegant way of representing the hardware of the underlying computer, and many of the properties of the operating system by treating anything that can be thought of as behaving like a file as being a file.

What, for example, might a hard disk look like to the operating system? Well, a hard disk is a device that can store a long sequence of numbers, and if you interpret those numbers correctly, they can be made to represent a filesystem. So as far as Unix is concerned, a hard disk is a bit like a file that you can read from and write to.

What about a process? Well that's a sequence of numbers in memory that happen to be instructions to the CPU to do useful things; so that's a file too (probably in this case a read-only file).

What about a keyboard? Surely that's not a file? Actually it can be thought of as having some file-like properties; it's a stream of numbers that represent the keys pressed by the user, so it too is a sort-of read-only file. And the screen? That too is file-like…because it represents a sequence of numbers that can be interpreted as the output from various processes; so it's a bit like a write-only file.

This may seem all a bit esoteric and confusing right now, but as we explore more examples of these file-like things, you'll start to see how elegant the idea is.

For now, to give this stuff about files some time to sink in a bit, let's play another game.

14 The Colossal Cave

We're now going to explore a bit of computing history, and install and play one of the very early computer games. Colossal Cave Adventure was the first 'adventure game', in which a virtual world is described using only text, and the player controls the game's protagonist using simple textual commands. The game was created in 1976 by a keen caver called **Will Crowther**^w who at time was a programmer at Bold, Berenek & Newman, the company that developed **ARPANET**^w, the forerunner to the modern Internet. He later collaborated with **Don Woods**^w, then a graduate student at Stanford University, to create the Colossal Cave Adventure as we would recognise it today. The original version consisted of around 700 lines of the **FORTRAN**^w programming language and a similar number of lines of data. When running on a **PDP-10**^w (see Figure 13 for a picture of what one of these machines looked like) Colossal Cave would consume around half of the machine's memory.

Although the original FORTRAN source code for Colossal Cave still exists, the version you're going to play with is based on a re-implementation of the game on what became known as the **Z-Machine**^w: a **virtual machine**^w specifically for running interactive fiction games, such as Colossal Cave.[1]

[1]Don't confuse the Z-machine, which is a virtual machine for adventure games, with the Z Machine, which is the largest X-ray generator in the world. Doing so is likely to make your lamp melt, and the trolls very grumpy.

Figure 13
A PDP-10 from CERN, circa 1974, reproduced with permission.
`http://cds.cern.ch/record/916840`.

14.1 Installing Frotz, a Z-Machine Interpreter

Unlike the other commands that you've used so far, the program we need to be able to play Colossal Cave Adventure isn't pre-installed on the VM, so we're going to have to fetch and install it ourselves. Fortunately, the version of Linux that we have on the VM comes with a package management system that makes this quite easy.

But first, we're going to have to understand a command called `sudo`. Everything that you've done so far has involved looking at files that either belong to the 'csimage'

⚙ **`sudo`**

user, or are parts of the system that can be read or executed by any user. But of course, installing a new piece of software involves *modifying* the operating system in some way, and that's not something that you want to do casually since mistakes could potentially mess up the whole device.

You'll be familiar with the idea of a user with **Administrator privileges**[w] from Windows or macOS; on Linux the **superuser**[w] that can do anything to any part of the system is called 'root' (because this user can modify any part of the system from the root of the filestore downwards). In the early days of Unix, administrators would log in as the root user to modify, update and repair the system. This had the major downside that all the normal safety nets that prevent you from accidentally deleting or damaging the operating system itself are deactivated, so its much easier for a slip of the finger or a brief moment of stupidity to have disastrous effects. To avoid these problems, Unix systems now usually recommend the use of the `sudo` ('Substitute User Do') command to temporarily elevate a normal user's privilege to that of super user for a single command. `sudo` privileges are not normally given to every user on a Unix system, but the user `csimage` on the VM is trusted and is given `sudo` rights.

The system that we're going to use to install this game is called `apt`, which stands for **Advanced Packaging Tool**[w]. This tool maintains a list of remote *repositories* in which packages have been put that contain all the programs, libraries

apt

and data files necessary to install a particular Linux program. It can deal with fetching packages over the Internet, as well as extracting and copying their contents into the right places on your system. It also performs a series of sanity-checks to make sure that what you're adding is compatible with whatever you've already got in place.

The system we want to install to play this game is called `frotz` (to learn why, you'll have to play the game a bit, or look it up on Google). Let's try running the `apt-get` command *without* having gained superuser privilege first. Try typing:

apt-get

```
$ apt-get install frotz
```

The operating system will respond with something like:

```
E: Could not open lock file /var/lib/dpkg/lock - open (13: Pe
E: Unable to acquire the dpkg frontend lock (/var/lib/dpkg/lo
   are you root?
```

Notice the question at the end: 'are you root?'. Well, no you're not, so Linux has rightly prevented you from performing this operation. Now we'll try again using the `sudo` command. This time type:

```
$ sudo apt-get install frotz
```

You will be prompted for your password here, so enter password for `csimage`

You should see a series of lines printed out on the screen, ending with:

```
Setting up frotz (2.44-0.1) ...
Update alternatives [...and more text...]
```

before being returned to the command prompt. It's possible that `apt-get` will fail to find the frotz package in one of the repositories it knows about; if this happens, it's usually because the repository has moved somewhere else on the internet, so you need to tell the APT system to update itself first: run the command `sudo apt-get update`, and when that has completed try installing the frotz package again, and all should be well.

☼ **`apt-get`**

The `frotz` system on its own is just a virtual machine into which you can load adventure game data, so we'll need to fetch the Colossal Cave datafile before we can play anything. We've put a copy of the game on the web at:

`http://syllabus.cs.manchester.ac.uk/ugt/COMP10120/files/Advent.z5`

which we'll get, as we did with the snake game in the first lab, using the command `curl`. First, use `cd` to change to your home directory if you're not there already, then

```
$ curl http://syllabus.cs.manchester.ac.uk/ugt/COMP10120/files/Advent.z5
```

Use `ls` to confirm that you can see the file `Advent.z5` in your home directory, then type:

```
$ frotz Advent.z5
```

to start playing the Colossal Cave Adventure. Once the game has started, type HELP to get instructions. When you've had a bit of a wander around and got the general idea of the game, you can type quit to get back to the command prompt. There are some parallels between using commands to wander around Colossal Cave and using Unix commands to navigate around the VM 's filesystem.

Earlier, we referred to Colossal Cave as a work of Interactive Fiction (IF). In truth, this is perhaps stretching the term somewhat, since the genre has matured considerably in the decades since this first adventure game. For a much more compelling example of Interactive Fiction with beautifully written prose, and funny and challenging puzzles we suggest you have a go at playing Curses by **Graham Nelson**[W], or one of the many other games written by Interactive Fiction enthusiasts that are available for free from `www.ifarchive.org`. If you find yourself getting hooked on playing IF, the frotz interpreter is available for most platforms, including iOS, Android, macOS and Windows.

15 Extracting files

We'll finish this lab session off with exploring a standard file archiving utility called `tar`.

Use `curl` to fetch the file hosted at

tar

```
https://bit.ly/3hRVsZr
```

making sure you save it in your home directory. By default, `curl` will just output whatever it has fetched to the screen. Use the `-o` option you learned about earlier to save it to a file called `mintwallpapers.tar.gz`. As we are using a shortened *url*, you would also need `-L` flag and the complete comand to get the file will be:

```
 curl -L https://bit.ly/3hRVsZr -o mintwallpapers.t
```

Notice that this file ends with `.tar.gz`. The `.gz` suffix tells us that this file has been compressed using a utility called `gzip`, so the first task is to uncompress the file. Type:

gzip

```
$ gunzip mintwallpapers.tar.gz
```

This will uncompress the file, removing the `.gz` and leaving you with `mintwallpapers.tar`. A 'tarfile' is a bundle of individual files that have been assembled together into a single file for convenience (you'll probably have already encountered 'zip' files, which have a similar purpose). The name 'tar' is an abbreviation of 'Tape Archive', since the `tar` command was originally used for making backups of filestores onto magnetic tape. It remains, however, a very versatile way of bundling up lots of things, and you'll find `tar` files all over the internet.

To see what's in this archive, run the command

```
$ tar tf mintwallpapers.tar
```

and you'll see a long list of the archive's contents scroll past on the screen. The first argument to `tar` is a bit of an odd one, since it's a collection of options, which unusually for Unix are not prefixed by individual minus signs (recall the `-o` option we used for `curl`; that's a far more common way of specifying options to tools). In this case the options mean:

- the **t** causes `tar` to list the 'table of contents', for the archive, without extracting anything, and
- the **f** tells tar that the next argument is the file containing the archive.

To actually extract the contents of the archive we issue the command:

```
$ tar xvf mintwallpapers.tar
```

where

- **x** means ‘extract’.
- **v** means ‘be verbose, and show what you’re extracting as you do it’.
- **f** again means ‘and here is the file to work on’.

When tar has finished working you’re presented again with the command prompt. Use `ls` to confirm that you now have a directory called `mint-backgrounds-ulyana` in your home directory. Feel free to explore the contents of this directory in your spare time.

Breakout 4: APT

The APT system is a very convenient way of managing packages, since it will automate the process of finding, fetching, configuring and installing software on your VM (or indeed, on other Debian-based Linux installations). The RPM system does something similar for distributions based on Red Hat's version of Linux.

The various repositories that contain the packages for your VM are updated regularly, so it's worth running `apt-get update` once in a while to refresh your VM's list of software.

⚙ **`apt-get`**

You should also at some point run `apt-get upgrade`, which will cause all the packages that have already been installed to be upgraded to the latest version that the APT system can find. This lab will work just fine with the versions of software that are pre-installed on your VM, and the upgrade process can take quite some time (hours, possibly), so you mustn't do it now or you won't be able to complete this lab in time. Try it at home, or outside of a lab session.

16 RTFM

Although we've introduced several Unix commands in this lab, we've only done so quite superficially in this session, giving you just enough detail to get through the tasks we've given you. Each of the commands is much more powerful than what you've been exposed to so far. Though you won't need to know every possible option off by heart, there are a lot of useful things you can learn about them quite easily.

Most Unix systems, including the one on your VM, have an instruction-manual system that gives more details about the available commands (and most things that you install yourself also install their own manual pages). Try running:

```
$ man ls
```

for information on the `ls` command, and use the same trick to find out more about the other commands you've seen in this session. If you need more help on how to use the `man` command, you can always use:

⚙ **`man`**

```
$ man man
```

When you're looking at a manual page, pressing the Space Bar will advance you on a page, and the Up and Down cursor keys will move you back and forth line-by-line. Press `q` to quit viewing a man page.

Breakout 5: RTFM?

The acronym RTFM stands for Read The Flipping Manual, and is sometimes used as a response when somebody has asked a lazy question on a forum or by email where decent documentation already exists and is easily accessible. The F is sometimes interpreted as meaning something less polite than 'Flipping'.

16.1 Using VM and University's lab machines

In case you want to use VM machine at home and also one of the university's lab machines in person then you need to make sure that you have run the `CS Backup` script on your VM before going to a lab. It will backup the files under `~/csimage/` and when you log into a lab machine, the CSImage work will be there.

Similary, you can restore your work to the VM by running the given below command on a VM command terminal (not the lab machine) and it will update your home directory with the changes that you may have made while using a lab machine.

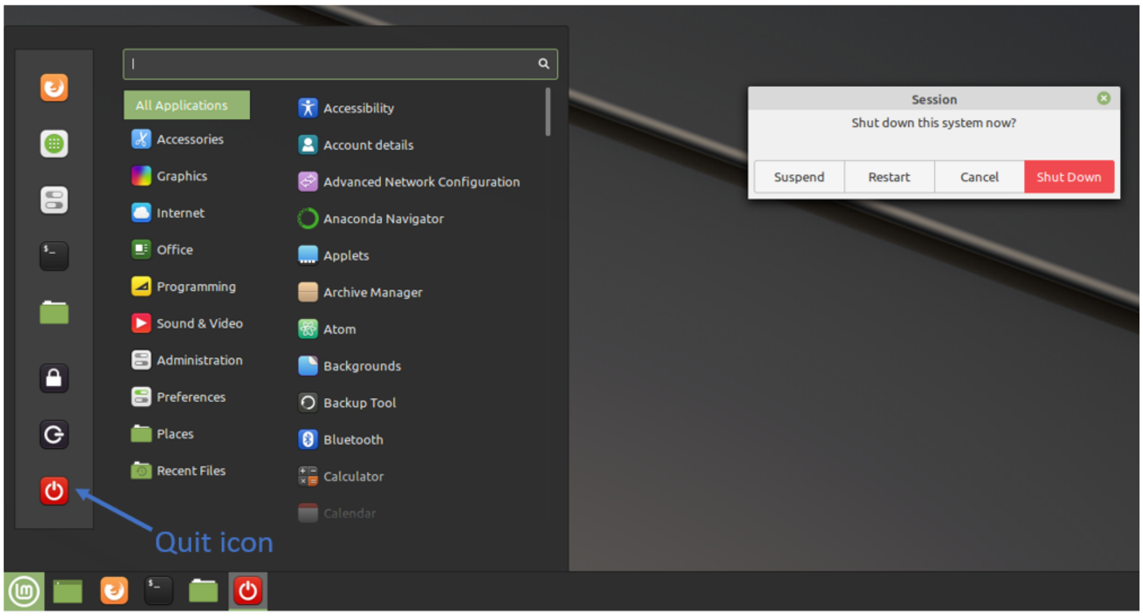
```
rsync -avzh <username>@kilburn.cs.man.ac.uk:csimage
~/
```

Note: Don't forget to replace <username> with your University of Manchester username (e.g. c12345ab)

17 Shutting down your VM safely

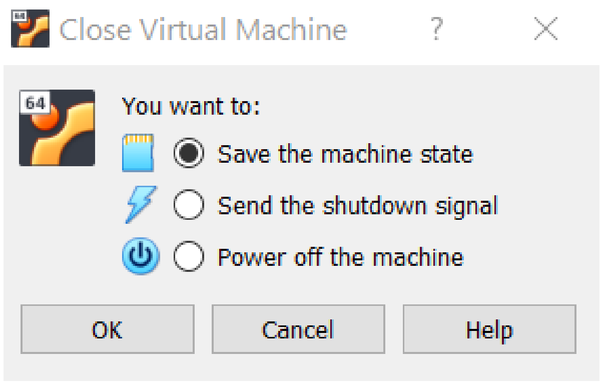
Like any other computer, it is really important that you shut your VM down properly; if you just close the VM there's a chance of corrupting the filesystem. To shut the VM down safely, click on the icon at the bottom left corner and then select the icon as shown in Figure 14):

Alternatively, you can also save the current state of your VM without losing your work by simpling clicking on the the cross icon at the top right corner of the **VirtualBox** and then selecting "save the sate machine" from the menue as shown in Figure 15



Shut down this system now?

Figure 14
Shutting down your VM.


Close Virtual Machine

You want to:

(●) Save the machine state

( ) Send the shutdown signal

( ) Power off the machine

Figure 15
Saving the current state of your VM.

18 What have you learned?

It might seem like you've been playing games for most of the lab, but if you've followed the instructions carefully and read through all the text you'll have learned a lot of new things. These include:

- the anatomy of a VM
- how to safely start and stop your VM
- running commands, e.g.: `echo`, `ls`, `cd`, `pwd`, `sudo`, `gunzip` and `tar`.
- how the filestore is structured
- basic `apt` commands

19 Logging in

Make sure you have started the VM, and log in using the provided username and password . Open 'Terminal Emulator' window as you did in the previous lab. Type `pwd` to find out which directory you are in. It should be something like `/home/csimage`.

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.

20 Listing files

First, we will use `ls` which is a utility for listing the files in a directory. For example, you can list the contents of `/usr/bin` by typing

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

and you should be given a much smaller list of things from the `/usr/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 `/usr/bin` directory, showing only files that start with the letters `ma` and then are followed by zero or more other characters'.

You could narrow this down even further by typing `ls /usr/bin/man*` in which case you'll only get files from `/usr/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.

20.1 Autocomplete

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!

21 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. For this, we will be using a special command-line based web browser called `lynx`.

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

☼ **`lynx`**

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

☼ **`which`**

```
$ which lynx
```

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

Back to browsing the web, try browsing the Department's web pages using `lynx` by typing

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

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 naviga-

Breakout 6: File extensions

If you've mostly used Windows or macOS 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 `mintwallpapers.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.

tion 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`.

21.1 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 com-

Breakout 7: 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.

plex 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/` 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.

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 8). 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` com-

⚙ **less**

Breakout 8: 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`**

mand you used in the previous lab session actually used `less` to display the manual pages.

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
```

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.

21.2 Searching patterns through files

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`**

`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:

```
    Manchester Sunny Intervals
```

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

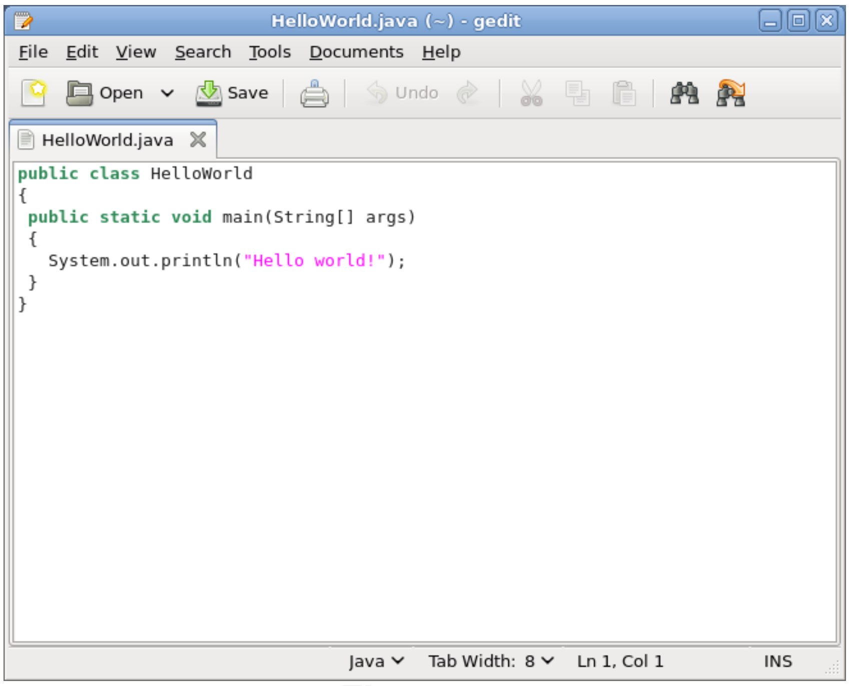
```
$ mankyweather
```

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.

22 Text Editors

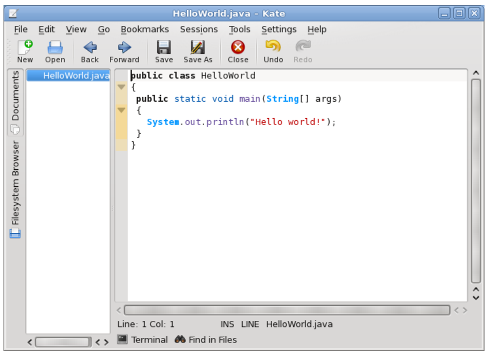
A great deal of the lab work you will be doing over your time here will involve you creating text files of various kinds, often source files in a programming language such as **Python**^{w}, **Java**^{w}, **PHP**^{w} or **C**^{w}, or **HTML**^{w} files for use on the web. There are specialist tools called **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 16 and 17. 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`.


HelloWorld.java (~) - gedit
File Edit View Search Tools Documents Help
Open
Save
Undo
HelloWorld.java
Java
Tab Width: 8
Ln 1, Col 1
INS

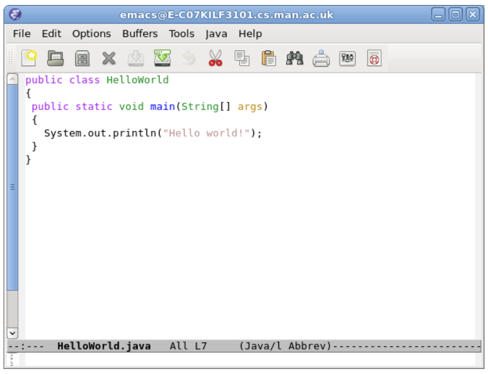
```
public class HelloWorld
{
 public static void main(String[] args)
 {
   System.out.println("Hello world!");
 }
}
```

Figure 16
gedit



```
public class HelloWorld
{
 public static void main(String[] args)
 {
   System.out.println("Hello world!");
 }
}
```

(a) kate


emacs@E-C07KILF3101.cs.man.ac.uk
File Edit Options Buffers Tools Java Help
--:--- HelloWorld.java   All L7   (Java/l Abbrev)------------------------

```
public class HelloWorld
{
 public static void main(String[] args)
 {
   System.out.println("Hello world!");
 }
}
```

(b) emacs

Figure 17
Other editors

23 Shell environment variables

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.

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

set

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?

When you first login, it looks for a file in your `csimage` directory called `.bash_profile` or `.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 `csimage` directory called `.bash_profile` or `.profile` (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).

`.profile` should look something like this:

```
# ~/.profile: executed by the command interpreter f
# This file is not read by bash(1), if ~/.bash_prof
# exists.
# see /usr/share/doc/bash/examples/startup-files fo
# the files are located in the bash-doc package.

# the default umask is set in /etc/profile; for set
# for ssh logins, install and configure the libpam-
#umask 022
```

```
# if running bash
if [ -n "$BASH_VERSION" ]; then
    # include .bashrc if it exists
    if [ -f "$HOME/.bashrc" ]; then
. "$HOME/.bashrc"
    fi
fi

# set PATH so it includes user's private bin if it
if [ -d "$HOME/bin" ] ; then
    PATH="$HOME/bin:$PATH"
fi

# set PATH so it includes user's private bin if it
if [ -d "$HOME/.local/bin" ] ; then
    PATH="$HOME/.local/bin:$PATH"
fi
```

The way to make the change permanent is to modify your `.profile` file. Use `gedit` to modify `.profile`, type:

```
gedit .profile
```

Add the following lines at the end of this file:

```
PATH=$PATH:/home/csimage/bin/
export PATH
```

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

24 Acknowledgements

The notes for the Intro Labs 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.