Intro Lab Session 1

Getting started with Raspberry Pi

Contents

1.1 Introduction 20
1.1.1 Using the Pi 20
1.2 Processes and the Unix Shell 25
1.2.1 The process 'family tree' 26
1.3 File systems and files 26
1.3.1 The Unix filesystem 28
1.3.2 Files in more detail 31
1.4 The Colossal Cave 32
1.4.1 Installing Frotz, a Z-Machine Interpreter 33
1.5 Quake 35
1.6 RTFM 37
1.7 Shutting down your Pi safely 38
1.8 What have you learned? 38

These notes are available online at
`studentnet.cs.manchester.ac.uk/ugt/COMP10120/labscripts/intro1.pdf`
You may find it useful to use the online version so that you can follow web links.

1.1 Introduction

This and subsequent introductory labs contain a large amount of material. Don't worry if you can't complete it all during the timetabled lab session, but you should try to catch up outside the lab before your next session. As always, if you're having problems, please tell someone and ask for help!

1.1.1 Using the Pi

If all went well in the last lab session, your Pi should be ready for use, but if you did not complete the Raspbian configuration, please go back to the Welcome Lab 0 notes and finish it now.

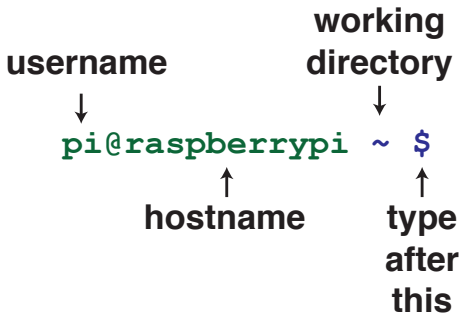


Figure 1.1
The different components of the Pi's default command prompt.

In this session we are going to complete the setup of your Raspberry Pi and explore some of its capabilities. Please connect your Pi as you did in the previous lab – do not connect it to the network yet.

At the login line enter `pi` as the username, and when prompted for the password, type `raspberry`. **Note that, so that anyone standing behind you cannot even see the length of your password, the username will appear on screen as you type it, but the password will not** (so make sure you're typing carefully!). Not echoing anything during password typing is fairly universal in Unix systems.

You should now see:

```
Last login: Mon Feb 16 14:46:27 2015
Linux raspberrypi 3.18.7-v7+ #755 SMP PREEMPT Thu Feb 12 17:20:48 GMT 2015 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
pi@raspberrypi ~ $
```

The last line of this text (which is shown in bold here but will be green and blue on your screen) is the **command prompt**[W]. 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 Mac OS X, 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 Pi consists of four components shown in Figure 1.1.

- To the left of the `@` symbol is the name of the user, and in this case that's `pi`, since you've just logged in under that name
- To the right of the `@` is the hostname of the machine, which on a Raspberry Pi is quite reasonably set to `raspberrypi` by default.

Breakout 1.1: 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 for this: 1) move hand off keyboard to mouse; use pointer to find the 'File' menu, from the file menu move the pointer to the 'Save' option; move hand from mouse back to keyboard. Or 2) Press the combination of keys that perform the 'save' function. Which do you think is faster?

And think carefully about the best tool for the job; sometimes it'll be the mouse/menu combination, but perhaps more often than you might think, a few selected commands may get the job done considerably more quickly. You've probably had more experience of doing things the GUI-way up until now so, during these labs, please use the command-line wherever possible to built up your familiarity until you are able to judge the pros and cons of both approaches to make an informed decision each time.

- The ~ tells you where in the Pi'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 Pi 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

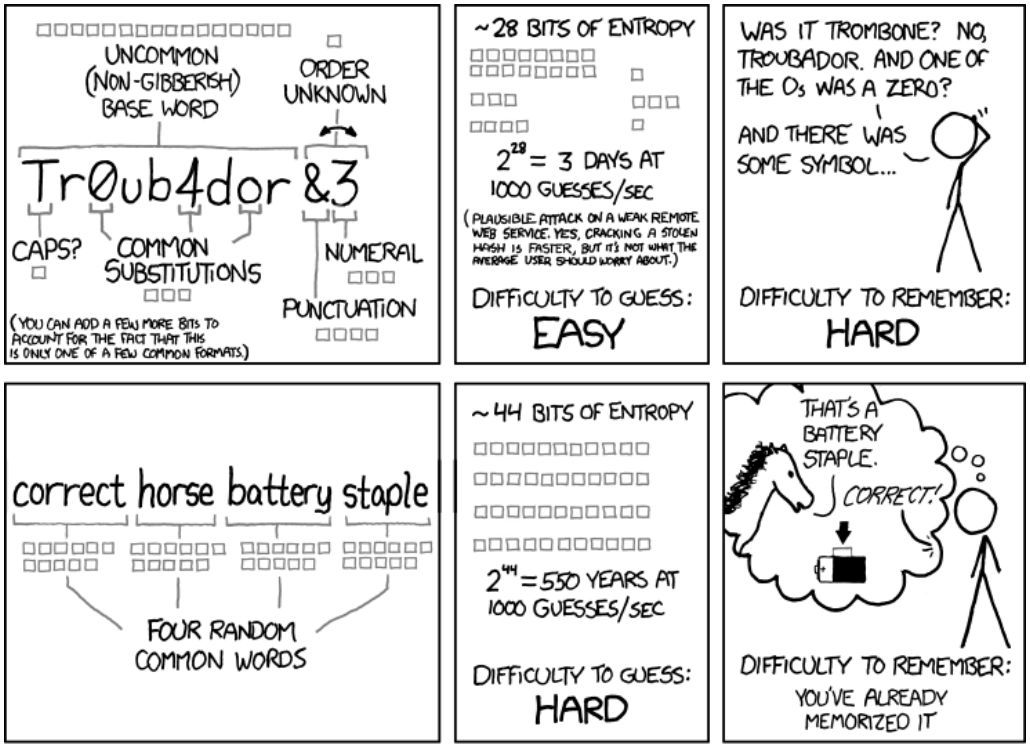


Figure 1.2
XKCD's take on password creation `xkcd.com/936/`

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

Before we do any real work on the Pi, we should make it a bit more secure than it currently is. Remember, you've logged in using the default username 'pi' and the default password 'raspberry', so anybody else could do the same; so this is the time to change the password to something that's unique.

For webcomic XKCD's view on creating good passwords, see Figure 1.2.

The command to change the password on the Pi is `passwd`; typing this at the command prompt and entering your current password (i.e. 'raspberry') and new password (twice, to be sure), should look like this:

passwd

```
$ passwd
Changing password for pi.
(current) Unix password:
Enter new Unix password:
Retype new Unix password:
passwd: password updated successfully
```

Note in the above, we're using lines that don't start with the dollar symbol to show output as a result of what you've typed, and that none of the passwords you're typing actually appear on screen for obvious sneaky over-the-shoulder-peeking reasons.

In the previous lab session we introduced you to **Danger boxes**, here is the first example, so please take note.

Don't forget your password. It is very important that you do not forget this password; there's no easy way to get back into your Pi without resetting everything.

Are you sure you know what your password is? If so then type `logout` at the command prompt. Not surprisingly this will log you out of your Pi. Then log back in again using your new password.

`logout`

Memory Loss. Although the Pi uses a fairly standard Unix operating system, it's probably not quite as secure as a normal desktop machine because of its small size and easily-removable storage media. Once the Pi has booted from the SD card, it's about as secure as any other Linux machine, but because the memory card is easily removable, it can trivially be connected to another machine as a 'removable media' device; and at that point the host machine can almost certainly see its contents, including any of the files you've created, because the filesystem itself isn't encrypted. And because the Pi is small and portable, it's easier to lose it than a laptop or desktop machine; so, be careful!

Now that your password isn't the same as the 'out of the box' Pi one, it's safe to plug your Pi into the network. You will need to refer to a couple of the illustrations of the Pi that were in Intro lab 0. Locate the blue ethernet cable poking out of the desk (labelled ❹ on Figure 9) on page 13; plug that into the appropriate socket on your Pi (number ❽ on Figure 7, page 12). 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 School's web server. Don't worry about the `sudo` part, we'll explain that later in the lab. You will be prompted for your password here, again don't worry about this for the moment.

`ping`

```
$ sudo ping www.cs.manchester.ac.uk
PING cmpsci.eps.its.man.ac.uk (130.88.98.241) 56(84) bytes of data.
64 bytes from eps.its.man.ac.uk (130.88.98.241): icmp_req=1 ttl=61 time=0.949 ms
64 bytes from eps.its.man.ac.uk (130.88.98.241): icmp_req=2 ttl=61 time=1.01 ms
64 bytes from eps.its.man.ac.uk (130.88.98.241): icmp_req=3 ttl=61 time=0.861 ms
```

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 Pi, find (in this case) `www.cs.manchester.ac.uk` on the network, and return back to your Pi. 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**.

Breakout 1.2: 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!

1.2 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 Pi 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.

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' anyway, 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 like your Pi 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 1.3.

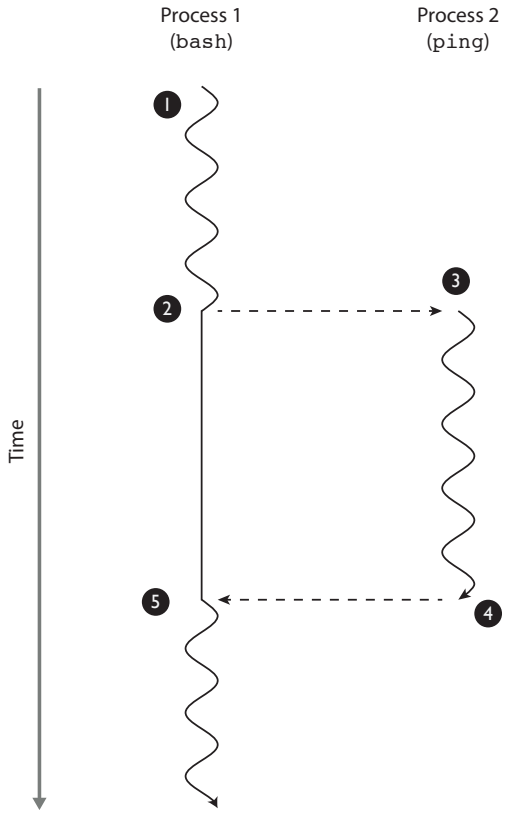


Figure 1.3
Running a command at a bash shell involves two process, one for the bash shell itself, and a second child process that is started by the bash shell in which to run the command: ❶ initially, just the bash process is running. ❷ at the point where you type the ping command and press enter, bash 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.

You will learn much more about processes and the way they are managed in COMP15111 and COMP25111; for now there's one more thing you need to understand about the relationship between processes.

1.2.1 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 Pi lab).

1.3 File systems and files

Next we're going to explore the Pi'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

Breakout 1.3: 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.

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 2013'.

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 Pi. The `ls` command lists files and directories. Type it now, and you should see that a directory called `python_games` has already been created for you, and other directories that you can ignore for the time being.

ls

```
$ ls
python_games
```

When we're using a command-line prompt, we have the notion of **current working directory** 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

```
pi@raspberrypi ~ $
```

it has now become

```
pi@raspberrypi ~/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 should see a long list of files: some are programs written in the Python programming language (these files end in `.py`), others are images or sounds used by those programs (ending in `.png` or `.wav`). At the prompt type:

```
$ python wormy.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. You'll be writing a version of this game soon enough in the Java labs.

1.3.1 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 Pi (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 OS X, however uses 'HFS Plus' (which usually appears as 'Mac OS Extended (Journaled)'), and this 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*! The same is true for the FAT32 filesystem used on most removable USB drives – because it's one of the few formats that's understood by Windows, Mac and Linux.

Most of the time this 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).

You can think of this structure as defining a tree, with / as the root (hence its name), directories as branches, and other files as leaves. You will study trees as abstract data structures, later in this year, and in Year 2, but this simplified model of the Unix file system structure will do for now.

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, shortcuts 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 Pi 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 Pi'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/pi
```

So apparently we're in a directory called `/home/pi` 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/pi` are given by their 'steps' from the root. So we now

pwd

know that the home directory for the user `pi` is in a directory called `home` which itself is a subdirectory of the root `/`.

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 5.1 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 pi`
4. We could go back to the root directory by first typing `cd /`, then `cd home/pi`
5. We could go straight from where we are now (which is `/bin`, remember) by typing `cd /home/pi` – 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

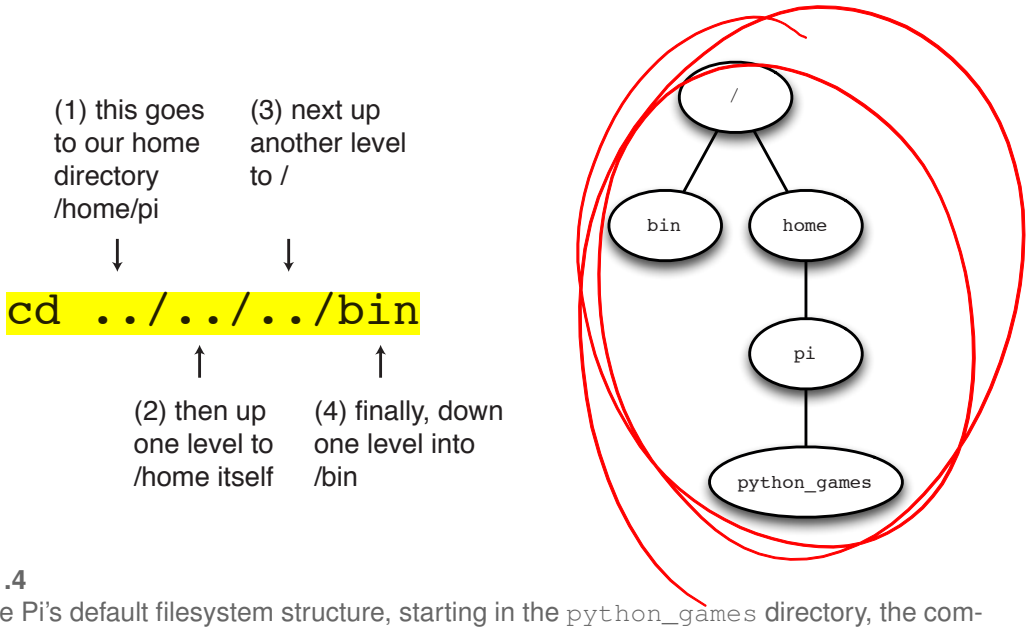


Figure 1.4
Given the Pi's default filesystem structure, starting in the `python_games` directory, the command `cd ../../../bin` will take you to the `/bin` directory (by an admittedly tortuous route!)

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 1.4.

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`

1.3.2 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.

1.4 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 1.5 for a picture of what one of these machines looked like) Colossal Cave would consume around half of the machine's memory. To put this in perspective, the tiny Raspberry Pi computer on your desk has roughly 1000 times as much memory as the PDP-10; it can run Colossal Cave with ease.

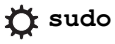
Although the original FORTRAN source code for Colossal Cave still exists, the version you're going to play with is based on a re-implementation of the game on what became known as the **Z-Machine**[w]: a **virtual machine**[w] specifically for running interactive fiction games, such as Colossal Cave.[1]

[1] Don't confuse the Z-machine, which is a virtual machine for adventure games, with the Z Machine, which is

Figure 1.5
A PDP-10 from CERN, circa 1974, reproduced with permission. `cds.cern.ch/record/916840`. We've taken the liberty of crudely superimposing a Raspberry Pi to approximately the right scale on the operator's desk, just to give a sense of the difference in size between the two machines.

1.4.1 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 Raspberry Pi, so we're going to have to fetch and install it ourselves. Fortunately, the version of Linux that we have on the Pi 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 'pi' 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 Pi's 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 OS X; 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 `pi` on the Raspberry Pi is trusted and is given `sudo` rights.

the largest X-ray generator in the world. Doing so is likely to make your lamp melt, and the trolls very grumpy.

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 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.

apt

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: Permission denied)
E: Unable to lock the administration directory (/var/lib/dpkg/), 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 should see a series of lines printed out on the screen, ending with:

```
Setting up frotz (2.43-4) ...
```

before being returned to the command prompt (note the version number for frotz may have changed from 2.43-4 by the time you do this exercise, don't worry, that's fine). 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.

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:

`studentnet.cs.manchester.ac.uk/ugt/COMP10120/files/Advent.z5`

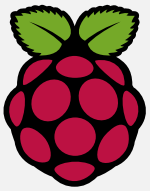
so you can fetch it from there. Oh, hang on. No web browser. Oh dear.

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`. First, use `cd` to change to your home directory if you're not there already, then

curl

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

Breakout 1.4: 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 Pi (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 Pi are updated regularly, so it's worth running `apt-get update` once in a while to refresh your Pi's list of software.

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 Pi, 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.

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.

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 (the screen will have gone blue), 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 Pi'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, OS X and Windows.

1.5 Quake

We'll finish this lab session off with one more game.

Use `curl` to fetch the file hosted at `studentnet.cs.manchester.ac.uk/ugt/COMP10120/files/quake3.tar.gz` making sure you save it in your home directory. By default, `curl` will just output whatever it has fetched to the screen. Use the option you learned about earlier to save it to a file called `quake3.tar.gz`.

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 quake3.tar.gz
```

This will uncompress the file, removing the `.gz` and leaving you with `quake3.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.

tar

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

```
$ tar tf quake3.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 quake3.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 `quake3` in your home directory. You might also want to plug some headphones into the Pi at this point too, if you happen to have a pair with you.

```
$ cd quake3
$ ./ioquake3.arm
```

This command executes the file `ioquake3.arm` which is in the current directory (`.`). After the startup screen, the game will ask you for a code, but you can just skip this and use the mouse to select the play option. The rest, we're sure, you can figure out for yourself.

Breakout 1.5: Multiplayer Quake?

You can play this version of Quake3 Arena with friends over the network. We'll leave you to figure out how to configure that yourself (hint: the `ifconfig` command will tell you what IP address has been allocated to your Pi).

ifconfig

1.6 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 Pi, have an instruction-manual system that gives more details about the available commands (and most things that you install yourself, such as `frotz` 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 1.6: 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'.

Pi Power. Like most other computers, the Pi doesn't like having its power removed without being shut down properly. Although you might get away with it, there's a reasonable chance of messing up the operating system if you remove the power while the Pi is in the middle of doing something. And because the Pi runs a multi-tasking operating system, it's almost always 'doing something', so pulling the power out unexpectedly is always a bad idea. You're unlikely to damage the Pi's hardware like this, but you may find that you lose work, and may have to reinstall the operating system. For instructions on how to shut the Pi down safely, see Section 1.7.

1.7 Shutting down your Pi safely

When you're finished playing Quake, exit the game and get back to the command prompt. Like any other computer, it is really important that you shut your Pi down properly; if you just pull the power cord out there's a chance of corrupting the filesystem. To shut the Pi down safely use the `halt` command:

halt

```
$ sudo halt
```

You'll see the a series of messages scroll past that look rather like those you saw during the boot process. This shouldn't be surprising, since what the operating system is doing now is shutting down all the things that it started up when you booted the machine, roughly in reverse order. When all these services have closed down tidily, the Pi will power itself down; the LEDs will go off, the screen should go black, and only the red PWR LED will remain lit on the circuit board. At this point it's safe to pull the Micro-USB cable out of the Pi. As always, please reconnect the USB cable ready for use by the desktop PC and reset the monitor to get input from the PC rather than the Pi.

When you have completed the work for this session, please tell the lab supervisor.

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