Swansea University
School of Engineering

EG-253: Introducing Linux

Dr Chris Jobling (C.P.Jobling@Swansea.ac.uk)
First Semsester 2008/2009
http://www.cpjobling.org/

Acknowledgements

This document is an adaptation of the course notes used to introduce Linux in the Computer Science module CS-244 Software Laboratory. About 99% of the content was written by my colleague Andy Gimblett.

1 Introduction

1.1 Aims

This part of the EG-253 aims to introduce you to the Linux operating system. There are two separate but related threads to this material, each of which feeds into the other:

- The concepts upon which the system is based – files, processes, shells, manuals, etc.
- The commands which enable you to actually work in the system, eg `ls`, `cp`, `chmod`, `tail`, `grep`, etc.

An ability to use Linux is important for at least three reasons:

1. Future modules, e.g. Web Applications Technology (EG-259) and Operating Systems (CS-228) rely on you being able to use it – You'll find the course-work harder without knowing your way around a Linux machine. You may also find the UNIX platform useful for C-Programming in EG-244.
2. Linux and Unix in general is becoming increasingly prominent in the IT industry, particularly as a platform for internet application servers, so it's a good skill to have, and might actually get you a job one day.
3. A comparative analysis and contrast with other operating systems with which the you are familiar can only be a good thing, intellectually and critically.

We would also add that the operating system "OS X" (found on recent Macintosh computers) is built on Unix elements, and much of what you learn now is also applicable in that setting.

1.2 Lectures and Assessment

- 3 two-hour lab sessions with lectures and tutorials on Linux.
- 1 two-hour lab of self-directed practice.
- One piece of Linux coursework (10% of EG-253).

For convenience in the first lab session we will be using a so called *Live CD* version of Linux that can be run on the PCs in any PC lab. During the second lab session you will install your own copy of Linux onto a shared PC.

1.3 Recommended Reading

Linux is a big topic, and this course can only really offer pointers and initial material. The only real way to learn how to use Linux is to **do it**, but it helps to have have good resources to hand for when you get stuck (and you will get stuck). Here are some suggestions.

"Learning UNIX" Books

- D. Ray and E. Ray, **Unix: Visual QuickStart Guide**, 2nd Edition, *Peachpit Press*, 2003.
- J. Peek, G. Todino and J. Strang, **Learning the UNIX Operating System**, 5th Edition, *O'Reilly Media Inc.*, 2001.
- J. Valade, **Spring into Linux**, *Addison Wesley*, 2005. A good introduction to Linux as a desk top replacement for Windows.

Textbooks and References

- M. Welsh, M. K. Dalheimer and L. Kaufman, **Running Linux**, 4th Edition, *O'Reilly Media Inc.*, 2002.
- E. Siever *et al*, **Linux in a Nutshell – A Desktop Quick Reference**, 4th Edition, *O'Reilly Media Inc.*, 2003.
- M. Joy *et al.*, **Introducing UNIX and Linux**, *Palgrave Macmillan*, 2002.

- M. Stutz, **The Linux Cookbook: Tips and Techniques for Everyday Use**, 2nd Edition, *No Starch Press*, 2004. Published version of a former Linux Documentation Project on-line document.

- B. Maco Hill, J. Bacon *et al.*, **The Official Ubuntu Book**, *Prentice Hall*, 2007. The "Official" manual for the Linux distribution (Ubuntu 6.06 "Dapper Drake") that we will be using in this module.

Linux Documentation Project (`www.tldp.org`)

- `/LDP/intro-linux/html/` – An Introduction to Linux

- `/LDP/Linux-Dictionary/html/` – The Linux Dictionary

- `/LDP/linuxcookbook/html/` – The Linux Cookbook (Tips and Techniques for Everyday Use). No longer available: see `http://dsl.org/cookbook/cookbook_toc.html` for the first edition.

If you can afford to buy two books, consider **Unix: Visual Quickstart Guide** and **Running Linux**. If you're going to buy just one book buy **Running Linux**. If you are going to buy just zero books, read the **Introduction to Linux** at the Linux Documentation Project and the first edition of the **The Linux Cookbook**.

Others Sources

We'll see some other sources of information later, most notably **man** pages and the GNU **info** system. There's also lots more info online.

Linux is a system which helps those who help themselves. The bottom line is often **RTFM**: Read The *Fine* Manual.

2 Context and History

Linux is a "Unix-like" operating system, i.e. to some degree it looks like and behaves like the classic industrial operating system Unix. Much of what we have to say about Linux on this course is also true of Unix, although Unix comes in many forms and so there are differences too. Linux is certainly the most popular (in terms of machines installed upon) Unix on the planet today, and also the most portable. It's also free, which is nice.

2.1 Unix

Unix has been around since the late 1960s, growing out of a project at AT&T Bell Labs in New Jersey to create a multi-user operating system called MULTICS (Multiplexed Information and Computing Service). When the MULTICS project was abandoned, a few AT&T workers, most notably **Ken Thompson** and **Dennis Ritchie**, tried to revive the project (or at least its good parts) and created a system they called **UNIX** – essentially just a joke on the word MULTICS, since their new system only ran on machines which were too small to support multiple users (initially, at least).

The aims of the Unix developers were to create a system with the following attributes:

- Easily portable across different hardware. This was a very big deal at that time, as hardware would be bought bundled with a dedicated OS. This made upgrading or switching to another vendor's hardware difficult or impossible.
- Simple, small, and elegant. This aim (particularly the 'small' and 'simple' parts) has been somewhat lost over the years, perhaps inevitably, but most Unixes are still smaller and easier to understand—or at least more consistent and transparent— than other modern operating systems.
- Promoting code re-use, both by using *libraries*, and also by encouraging the use of small, dedicated *modular* application programs which can be joined together in a multitude of ways to perform some task.

  Unix utility programs tend to be small and aim to *do one thing well*. Unix then provides powerful mechanisms (redirection, pipes, shell programming) to combine these to perform more complicated tasks.

The Unix developers made a number of initial design decisions which seem to have given Unix an unusual degree of flexibility and longevity, and which has seen it at the heart of industrial and academic computing for much of the intervening time. Proponents of Unix find its design to be elegant, powerful, and productive.

The Unix philosophy rejects the idea that 'user friendly' means 'easy to *learn* how to use', favouring instead tools which have an initially steep learning curve but which provide greater power and flexibility as a result, and which exhibit a coherence of design missing in other systems. Put another way, the entire Unix philosophy revolves around the idea that the user knows what he or she is doing.

- M. Gancarz, **The UNIX Philosophy**, *Digital Press*, 1995.
- `http://www.crackmonkey.org/unix.html` – History of UNIX.

2.2 GNU, MINIX, and Linux

By the early 1980s, Unix had a firm foothold in industry and (especially) academia, but corporate factors had also caused schisms and conflicts within the Unix world, leading to many different 'flavours' of Unix, all competing, and all jealously guarding their intellectual property.

In January 1984, **Richard Stallman** (RMS), a researcher at MIT's AI lab, frustrated by the closed-ness of the systems he was using, quit his job and started writing an entirely free and **open** Unix-like operating system, calling it **GNU** (for "GNU's Not Unix" – this is an example of something called a *recursive acronym*; they pop up all over the place in the Unix/Linux world). By 'open' we mean that the source code was freely available for anybody to read and even modify, with the legal restriction that if they wished to distribute the changed code, they must keep it open.

The GNU project (`http://www.gnu.org/`) has given us a vast amount of extremely useful software. Probably the most important example is `gcc`, the GNU C Compiler, which (for instance) is used to compile virtually every part of a running Linux system, including the kernel and all of the utilities. (GCC also has a life outside Linux, and is one of the compilers of choice for developers of embedded and real time operating systems, for example.) In fact, most of the utilities you'll use day-to-day in Linux actually come from the GNU project. For this reason, RMS refers to what we call Linux as GNU/Linux, and tends to get annoyed when other people don't.

Given GNU, where does Linux fit in? Any OS needs a **kernel**, which is the central part of the OS concerned with accessing hardware resources, and with coordinating the processes running on the machine[1]. **Linux** is in fact (kind of) just the name of the kernel.

In 1991 a Finnish Computer Science student called **Linus Torvalds**, decided to write an operating system kernel. He'd been studying **MINIX**, a 'toy' kernel developed and freely published by computer scientist **Andrew Tanenbaum**, and came to the conclusion that he wanted to do his own. MINIX was seen as an excellent teaching tool, but not a kernel you'd want to run in production, and although

[1] More on kernels in TB2, in CS-228 Operating Systems. Bet you can't wait.

Linus never foresaw what was about to happen, he set out to make something a bit more 'real world'.

What happened next was that, thanks to a strange combination of social factors (ed the coming of age of the Internet), a lot of people started working on his little kernel, started making other software run with it (including, critically, the GNU tools), and over the next few years it got a name for itself as a decent and free Unix-like system. By the end of the 1990s it had achieved some sort of critical mass and was making a serious impact on industry and academia – just as Unix had previously.

There are now also many flavours of Linux (eg Red Hat, SuSE, Mandriva (formally Mandrake), Gentoo, Debian) but because (almost) all of the software released for Linux (including Linux itself) is open source, the differences which emerge in the systems tend to feed into each other, making the system richer as a whole – as opposed to holding each other back with copyright and lawsuits.

The GNU project is of course developing a kernel, called **The HURD** (another recursive acronym[2]), and whilst its design is admirable, it is not yet complete. Part of the reason for this is timing: once people started using the Linux kernel (introduced initially as a 'make do' from the point of view of the GNU people), there was a lot less impetus to finish its development. It's still there, somewhat usable, but it lacks the years and years of effort and continuous development which have made Linux into the powerful and flexible kernel it is today.

- `http://www.google.com/search?q=history.of.linux` — lots of Linux history online.

3 Linux System Overview

The Linux System is an example of a layered architecture as illustrated in Fig. 1.

3.1 The Kernel

The kernel is that part of the operating system which is concerned with all of the housekeeping necessary to keep the system running. In particular it is concerned

[2] "According to Thomas Bushnell, BSG, the primary architect of the Hurd: 'Hurd' stands for 'Hird of Unix-Replacing Daemons'. And, then, 'Hird' stands for 'Hurd of Interfaces Representing Depth'. We have here, to my knowledge, the first software to be named by a pair of mutually recursive acronyms." Quoted from the HURD web page at `http://www.gnu.org/software/hurd/hurd.html`

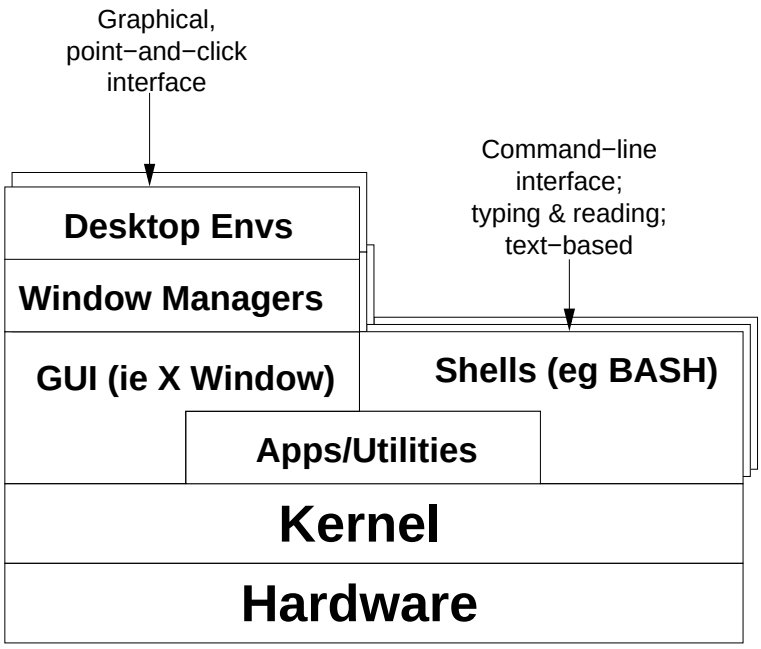

Figure 1: Linux Organisational Overview

with:

- Providing a coherent and controlled interface so that programs can access hardware without doing so directly.
- Coordinating the various processes running on the machine, allocating resources to them, scheduling them to run, etc.
- Various other low level facilities for tasks such file systems, networking, etc.

The main thing to know at this point is that whatever you ask Linux to do, the kernel is intimately involved. However, you don't use it directly – instead we access its facilities using applications which build upon it. This is exactly the same situation as in virtually any other OS, including and in particular Microsoft Windows.

We don't look at the kernel any further in this course. Students of CS-228 Operating Systems will meet kernels and their functionality in detail later in the year.

3.2 Applications and Utilities

As mentioned above, the kernel's facilities are never accessed directly by the user. Instead they're made available by the provision of **system libraries**, which are collections of function calls to be used by programs written in C and other languages[3]. Thus, built upon the kernel and the system libraries we have a large set of applications and utilities which actually make the OS useful and useable.

We'll meet many applications & utilities throughout this course – in fact most of the things we'll learn how to use fall into this category; here are a few random examples:

- `cp` to copy files from one place to another.
- `gcc` to compile C source into executable code.
- `ssh` to securely log in to or run programs on a remote machine.
- `gimp` to edit images.

[3] Again, CS-228 will tell us all about system libraries in more detail.

One class of application programs which is of particular importance are those concerned with providing an interface within which we can run other programs. These are divided into two classes and considered separately: **shells** for command-line usage, and **windowing systems** for graphical usage.

3.3 Shells

A **shell**[4] is a command-line environment within which you can run other commands. It is entirely analogous to the 'DOS Prompt' or 'Command Prompt' in Microsoft Windows, but Unix shells tend to be *much* more powerful than the vanilla Windows prompt.

There is no single all-pervading shell in Unix or Linux. Instead, there are a number of offerings, and it is up to you which one you use — each one has certain features, idiosyncracies and advantages, and everyone has a preference. Having said that, in the Linux world, the *de facto* shell, and the one we will study, is called **bash** (the Bourne Again shell). Other examples are **csh** (the C shell), **tcsh**, **sh** (the 'classic' Bourne shell), **ksh** (the Korn shell) and **zsh**.

We said above that most of the things we'll learn how to use on the course are applications and utilities. Well, the other major component of this course is shells and how to use them (well, **bash**) effectively.

3.4 X Window – GUI

The 'classic' Graphical User Interface on Unix systems is called **The X Window System**, or more usually (though inaccurately) 'X Windows' or (better) just 'X'.

There are other Unix GUI projects[5], but X is basically universal. Any graphical program you run under Linux is running 'under X'.

We won't particularly look at X on this course, but it's worth mentioning two related classes of software in order to avoid confusion: window managers and desktop environments.

3.5 Window Managers

X provides a framework within which one can develop graphical programs, but the X Window specification doesn't say anything about how windows, scroll-bars,

[4] Apparently, the etymology of the word 'shell' is simply that it hides the underlying details of what's happening from the user.

[5] For example, **Fresco** – `http://www.fresco.org/`

buttons, etc. should look, how windows are placed, how they're resized, etc. Instead, additional pieces of software called **window managers** provide such 'look and feel' policy. So when you run X you will in fact also run a window manager which controls how things look and behave.

There are, of course, a number of window managers, and the same application can look drastically different depending on the choice of WM. Common examples are **Black Box**, **WindowMaker**, **SawFish**, **KWM**, **IceWM**, etc. There are many, many more to choose from, although most people just use the one that comes with their desktop environment (see below). The author's window manager of choice is the pleasingly esoteric and difficult **Ion**.

3.6 Desktop Environments

Over the past decade there has been an increasing perception within the Linux developer community that whilst Linux is wonderfully powerful, its high 'barrier to entry' prevents new users from converting from the wonderfully 'user-friendly' and comfortable graphical worlds of Microsoft Windows or Apple Macintoshes. Although X and window managers provide a graphical framework, it's still not very pretty or very friendly – very much an engineer's environment.

In order to address this situation, a lot of development has gone into **desktop environments** – large collections of software, built upon a particular window manager and providing (or aiming to provide) a very consistent look-and-feel and making most of the tasks most people want to do with their system as easy as possible. These projects have been reasonably successful, to the degree that most Linux users these days use a desktop environment, rather than 'just' a window manager.

The history behind their development is beyond the scope of this course, but today there are two major desktop environments in use in the Linux world: **KDE** (using **KWM**) and **GNOME** (using **SawFish**). As with much in Linux, the choice of which to use is yours and each has its advantages and disadvantages. We will point out, however, that GNOME is the environment you will find on the Ubuntu Live CD.

4 Getting Started with Linux

In this section we'll take our first steps in Linux, and meet some of the fundamental concepts and commands without which there is no hope.

4.1 Logging In, Logging Out

Unix has been a multi-user system from day one, so whenever you use a Linux system you need to log in. This should be a familiar operation to everyone, however – enter your username and password, and wait for the windows to pop up[6].

You may be less familiar, however, with the concept of logging in to remote machines (i.e. machines other than the one in front of you). We'll see how this done later, but for now bear in mind that whenever you want to start a session on a Linux box, whether it's local or remote, you'll need to provide a valid username and password.

Once logged in, everything should be essentially familiar, and yet somewhat strange and different. You should play around with the menus a bit, start some programs up, close them down, etc. in a manner entirely familiar to anyone accustomed to Windows.

Exactly how you log out will depend on the desktop environment you're using, but under Ubuntu it is as simple as pressing the red *Quit*button at the top right of the screen (next to the time and date).[7]

4.2 Starting A Shell

What we're really interested in, however, is power usage, and that means the shell. In order to start the shell, look for a menu item with a name along the lines of **Terminal**, **xterm**, **shell**, or **bash**.[8] Once it starts, you should have a text window with a prompt, into which you can enter commands.

In order to check that this is indeed **bash** and not some other shell, type `help` and hit return. This *should* provide you with some text, of which we are interested

[6]If you are using the *Ubuntu Live CD*, you don't actually log in. Simply insert the CD in the drive and restart Windows to boot Linux. We recommend that you press the `F3` key to select the British keyboard layout and then type the `Enter Key` to select the *Start Ubuntu* (default) boot option. Once Ubuntu is fully loaded, you will find your self logged in to a GNOME desktop as user *ubuntu*

[7]It is possible to set up Ubuntu Live CD so that when you log off, your home directory is saved onto a USB memory stick so that next time you start up Ubuntu it will automatically "mount" a special file on the USB memory stick as `/home/ubuntu`. Unfortunately, the procedure for this is rather complex and you will need a new memory stick that you can partition and format as a Linux file system. If you are interested in doing this – for example because you want to use Ubuntu on your own PC or laptop but don't want to install it on your hard drive – instructions are to be found on the web (see link on Blackboard site).

[8]In Ubuntu it is `Applications> Accessories> Terminal`

only in the first line. The first line printed after you hit return (and you may need to scroll up in order to see it) should say something like:

```
GNU bash, version 3.1.17(1)-release (i486-pc-linux-gnu)
```

Of course, the version number and other stuff might vary, but what we're really interested in is `GNU bash`. If you see this, all is well, you're in `bash`. If you don't see this, don't panic, because it's easy to start `bash`. Simply type `bash` and hit return – this should launch a `bash` shell *within* whatever other shell it is you're running. If this doesn't work, something is wrong and you need to seek help...

We'll see some tips on how to use `bash` shortly, and meet some commands to try out but for now note that the basic method of usage is to type a command, hit enter, and get some response. This is entirely like the Command Prompt in MS Windows.

Before we try anything out, let's see how to exit `bash`. Basically there a two ways:

- Enter the command `exit`
- Hit `Ctrl-d` (ie, hold down the `Ctrl` key and hit the `d` key).

  We'll see why this second method works later on. We'll see how it works in CS-228.

4.3 Essential Commands

Let's take a look at some essential commands – the bread & butter stuff you'll find yourself using again and again in any session. We'll see each of these in more detail later on when we consider the aspect of the system which they particularly belong to.

- `ls` – list the contents of the current directory.
- `ls` *directory* – list the contents of the specified directory.
- `pwd` – print the path of the current directory.
- `cd` *directory*— change to the specified directory.
- `cd` – change directory to your home directory.

- `cp` *src dest* – make a copy of a file.
- `mv` *src dest* – move a file; this is also how you rename a file.
- `rm` *filename* – remove/delete a file.
- `mkdir` *directoryname* – create a directory.
- `rm -r` *directoryname* – delete an empty directory.
- `ps` – list processes being run.
- `top` – constantly updating list of processes running on system, busiest at top. Hit `q` to exit.
- `which` *progname* – print the path to the executable file *progname*
- `file` *filename* – print the 'type' of a file (ie is a text file, a binary executable, a directory, etc.)
- `cat` *filename* – print the contents of a text file to the screen in one go.
- `less` *filename* – read a text file one page at a time. Also could use `more`, which is similar but older; some people prefer it.

4.3.1 Case Sensitivity

Important Note – Linux (and Unix in general) is **case sensitive**, so `ls` and `LS` are *different* things. Most Unix commands have entirely lower-case names by convention; of course, the same is not true of files and directories, especially the ones you create yourself.

4.4 Getting More Help

Unix helps those who help themselves, by reading the manuals. It's an unavoidable fact that fundamentally, the only way to get to know this stuff is to do it, and to read the documentation when you don't know how to do something. That way you learn something more, and are one step closer to being a guru. Fortunately, Unix has traditionally had excellent and copious documentation, and Linux continues this fine tradition. This documentation is initially hard to get used to, and can seem unfriendly, but that's because it's targeted at experts, not beginners. The

closer you get to becoming an expert, the easier it is to glean useful information from the documentation. This may seem paradoxical, but if you're going to use Unix this is the way it's done.

Having said that, the internet has opened up a whole new world of documentation, "howtos", "beginner's guides", etc., so if you find the built-in documentation too hard to use, you can use the internet to get a foothold. Books such as those recommended in this document are also excellent resources in this respect.

4.4.1 The man pages

The **man pages** are the traditional and canonical repository of documentation. The idea is to have an entire manual for the entire system built in and available online in every system. `man` is, of course, short for manual. Here's what you need to know:

- To get documentation for some command type `man` *commandname*.

  Example: to read the manual page for the `ls` command, type `man ls`

- Scroll up and down using Page Up, Page Down, up and down arrows, etc. Easy.

- Exit by hitting `q` for quit.

- `man` pages are organized in a very particular fashion, and all follow the same sort of form. A typical `man` page will provide a summary of how the command should be used (including all the extra options to modify its behaviour), a detailed description of usage, a detailed description of each option, a list of known bugs, a list of related commands, and maybe some other stuff.

- To search for a `man` page on a particular topic, use either `man -k` or `apropos`.

  For example, `man -k file` will give a (big) list of `man` pages whose description contains the word `file`.

- Something you'll quite often see is a number in parentheses after a command line. For instance, you'll see that the `man` page for `ls` says `LS(1)` at the top.

These numbers refer to **chapters** in the manual. Different numbered chapters cover different aspects of the Unix system. Traditionally there are 8 chapters.

Most of the commands you're interested in live in chapter 1. File formats (eg the configuration file for some command) are described in chapter 5. Chapter 2 contains system calls and will become important when we start talking about C programming.

In order to read a `man` page in some particular chapter, specify the chapter number before the command name when running `man`.

For example, `man 2 kill`[9] will give you a different page than `man 1 kill`. (It's unusual to have pages with the same names in different chapters – `kill` is rare in this regard).

The `man` pages are a **vital** resource and you ignore them at your peril. Have a look at the `man` pages of some of the commands we saw earlier, and learn about the various ways they can be used. For instance, `ls` has many command-line options which can modify how it displays a directory's contents — you will definitely want to get to know some of them.

4.4.2 GNU info

`man` pages date from the early days of Unix. The GNU project have, for their own reasons, devised a more advanced online-manual system, called **info**. `info` is considerably more complicated than `man`[10], and requires effort just to learn how to use it. However, for regular users of GNU software (and if you get into Linux, This Means You), it's an indispensable resource.

- Basic usage is like `man`: to read the `info` page for `ls`, we enter `info ls`
- Gladly, `q` also quits `info`.
- Hit `?` to get quick help.

[9]Note: Chapters 2 and 3 of the `man` pages are not installed by Ubuntu by default. To get access to `man 2 kill` you will need to install the optional `manpages-dev` package. To do this simply issue the command: `sudo apt-get install manpages-dev` from the bash shell prompt.

[10]For example, pages are hyperlinked and navigable, like web pages – though `info` predates the WWW.

- Hit `h` to invoke a detailed tutorial on how to use `info` – if you want to learn how to use `info`, this is the way to start.

  If this splits the window into two horizontal panes, you can make the tutorial occupy all of the space by hitting `Ctrl-x` and then hitting `1`.

- You can just run `info` on its own without specifying a command to get a top-level node listing all the info available.

4.5 Bash Tips & Tricks

We'll finish this 'Getting Started' section by looking at some of the features of `bash` which can make life much easier, indeed positively marvelous. It is absolutely well worth taking the time to master these techniques, particularly **tab completion**, as it will save you an *enormous* amount of time.

4.5.1 History

`bash` remembers what you've typed, and provides a number of ways to recall previous lines, saving you from labouriously typing them in again.

- You can enter `!!` to repeat the last command entered.

- Similarly, you can enter `!-n` to run the `n`th most recent command in the history. For example, `!-2` will run the command which came before the last one you entered.

- You can hit the up and down arrows on your keyboard to navigate through the history. This is a very quick way of repeating something you did recently.

- You can enter `history` to print the entire command history (which may be very long), with each line individually numbered.

- You can then enter `!n` where `n` is some number from that history, to repeat some particular line. This is a good way to repeat the same command occasionally – if you remember that it's number 210 in your history, then every time you enter `!210`, it gets run again.

- (Advanced) You can hit `Ctrl-r` which takes you into a reverse-search mode, where you can enter **part** of a command line you typed previously, and it will provide suggestions.

  Keep hitting `Ctrl-r` to search further backward, and hit return when you've got the line you want. If you want to cancel, hit `Ctrl-g` (which often means 'cancel').

Note that each of the above are just built-in features of `bash` – they are **not** stand alone commands like the ones we saw earlier. So if you're not in `bash`, typing `!!` won't necessarily work. Of course, most shells provide this kind of functionality in some form, however.

4.5.2 Tab Completion

Tab completion is, to the author's mind, probably the best thing about `bash` – it's an incredibly smart idea, and a big big time and labour saver. The idea is this: when you hit the `tab` key, `bash` will 'guess' what you want to type, as much as it can. This is particularly useful when specifying directories and filenames.

It's best explained by example, and best understood by trying it out.

Try typing `ls /usr/` and then, instead of hitting return, hit `tab` a couple of times. This will print a list of possible completions (ie it's a very quick way to list a directory). On my system:

```
ubuntu@ubuntu:~$ ls /usr/
bin/     include/ local/ share/ X11R6/
games/   lib/     sbin/  src/
```

Now if I type `s` and hit tab again, I get this:

```
ubuntu@ubuntu:~$  ls /usr/s
sbin/   share/  src/
```

This is `bash` telling me it's found three possibilities, and that it needs more information from me about which one I want. If I keep hitting tab, it'll just beep at me. If I now type `r` and hit tab again, it sees that there is only one choice beginning with `sr` (namely `src`), so it 'completes' by typing the `c` for me, and my prompt reads:

```
ubuntu@ubuntu:~$ ls /usr/src/
```

If I hit tab again, I get a list of the contents of the `/usr/src` directory, and so it goes on.

Tab completion is great when the directory names are long, for instance 10 letters long rather than the 3 in the above example. It literally types the characters for you – fantastic!

If you get into the habit of just entering a couple of letters and then hitting tab, when you're doing anything involving files and directories, you'll find it saves you a lot of time. It also saves mistakes, as using tab completion you simply *can't* mistype a filename – if you do, the tab completion will fail and you'll very quickly realize something's wrong. This is a lot nicer than typing an 80 character path to some file, hitting enter, and *then* realizing you've mistyped something.

5 Files and File Systems

5.1 Files in Linux

In Unix, and thus in Linux, there is a philosophy that 'everything is either a file or a process'. In particular, everything in the file system is a file or at least a 'file-like object[11]'. Here is a list of the types of files one can encounter.

- Regular files – A **regular file** is an ordinary file which contains some data content, stored on disk. Conceptually, a file is just an array of bytes; if those bytes happen to be ASCII text, we call it a **plain text** file; otherwise it's a **binary** file, perhaps an executable, an image, a sound, or something else.
- Directories – In Unix a **directory** is really just another file which contains a list of the files in that directory. Generally, however, we access a directory's contents (ie its listing) using different methods from those used to, say, read an ordinary text file.
- Links – A **link** is a file which points to another file, like a 'shortcut' in MS Windows. We'll look at links in more detail shortly.
- Special files – So called **special files** are file-like objects which are used for i/o (`/dev`) or for accessing system information (`/proc`).

[11] A 'file-like object', is something which can be open, read from, written to, etc. just as if it was a file, but it's *not* a file and probably isn't stored on disk.

- Sockets – A **socket** is a file-like object used for communication between two computers across a network.

  Most programs which use sockets just create and use sockets in memory, but it's possible to create them in the file system, in order to allow any process use them.

- Named pipes – We'll meet pipes later on when we talk about processes. A **named pipe** is a pipe which looks like a file on the file system.

A path is a string of characters telling you where to find a certain file. For example, `/bin/cp` is the path to the file `cp` in the directory `bin` which is itself in the directory `/` – see section 5.7 for more information on paths.

5.2 File System Organization

You're probably familiar with the idea of 'drives' from Microsoft Windows – for instance, 'the **C:** drive' for your hard drive, 'the **A:** drive' for a floppy drive, 'the **D:** drive' as a CD-ROM or maybe another hard drive, etc. In Unix (and thus Linux), things are done somewhat differently (and, ultimately, more flexible and powerful, as one would expect).

A Linux computer has exactly one **file system hierarchy**. This is a tree of files and directories, whose root is a directory written as `/` (a forward slash), and pronounced 'slash' or 'the root directory'. `/` contains files and a number of subdirectories, each of which can contain further files and subdirectories, and so on to an arbitrary (nowadays) depth. **Every** file or directory you work with on the system appears somewhere in this hierarchy.

For example, when you execute `cp` you're executing a program which lives in the file `/bin/cp`, ie `cp` in the directory `bin`, which is in the directory `/` (by the way, I'd pronounce this as 'bin see pee' or, if I needed to be unambiguous, 'slash bin slash see pee').

See figures 2 and 3[12] for a short tour of some of the important directories you'll find on most Linux systems.

[12]Figure 3 is taken from The Linux Documentation Project's "Introduction To Linux", by Machtelt Garrels.

`/`	Root directory, everything hangs off here.
`/bin`	Many critical binary executables, such as `cp`, `ls`, etc.
`/boot`	Files which are important when the system is booting up, in particular the binary image of the kernel.
`/dev`	'Device files', which are 'special' files which aren't really files on the hard disk, but instead represent hardware devices (generally). For instance `/dev/hda` corresponds to the first hard disk. Only of interest to advanced users.
`/etc`	Configuration files for the system in general and for particular programs installed on the system.
`/home`	Users' 'home directories', ie where you get to keep all your own files.
`/lib`	Shared libraries of compiled system code, available to any program on the system.
`/mnt`	Mount points for removable media (eg `/mnt/cdrom`).
`/media`	Alternative to `/mnt` on some systems (e.g. Knoppix and Ubuntu).
`/proc`	Another 'special' directory, whose files contain information about running processes.
`/root`	The superuser's home directory.
`/sbin`	Binary executables used only by the 'superuser' (ie system administrator).
`/tmp`	Temporary storage, readable/writable by everyone.
`/usr`	Other than the very basics, most programs live somewhere off `/usr`.
`/usr/bin`	Most user-runnable programs installed on a system live here.
`/usr/lib`	Shared libraries installed by 'extra' programs.
`/var`	'Varying' files, such as system logs, mail spools, lock files, etc.
`/opt`	Optionally installed packages.

Figure 2: Some important directories in the Unix file system hierarchy

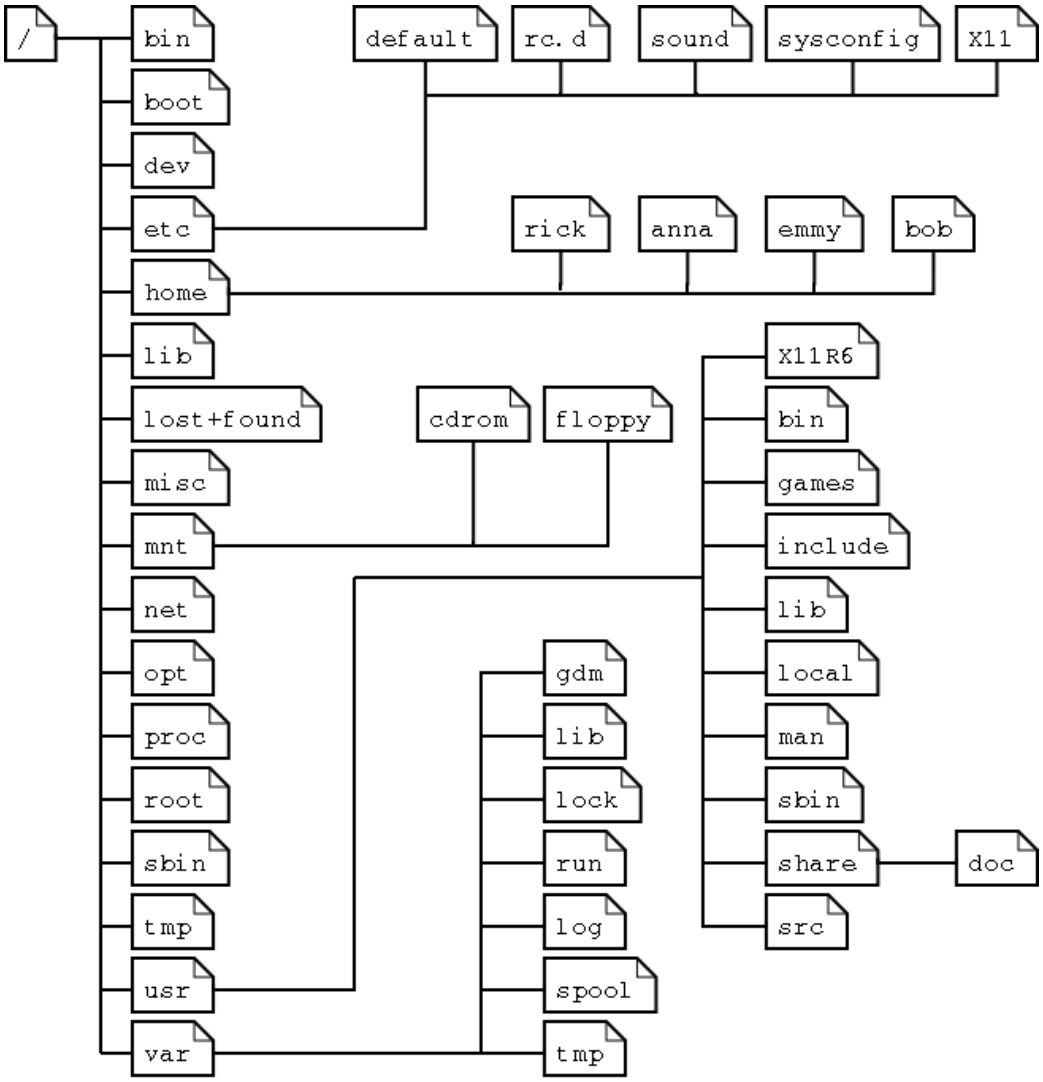

Figure 3: Typical partial file system hierarchy of a Linux box

5.3 Home Directories

Unix is a multi-user system, and as such each user of the system needs somewhere to keep their files. Typically, this is their 'home directory', located in a directory off `/home/` with the same name as their userid. So for instance, my home directory is at `/home/eechris/`/footnoteOn the Ubuntu Live CD, the user "*ubuntu*" has a home directory at `/home/ubuntu`. Your home directory belongs to you and typically can't be written to by anyone else.

The exception is the superuser, who is always called `root` and whose home directory is kept separately, in `/root/`. But unless you run your own Linux system, you don't need to worry about that. :-)

Whoever you are, you can refer to your home directory using the shorthand ~ (pronounced 'tilde' or 'twiddles'), which can save a lot of typing. Furthermore, you can refer to *other people's* home directories by specifying their name, eg ~`eechris`. Typically `bash`'s tab completion will expand such paths.

5.4 Mounting and Mount Points

We stated in section 5.2 that a Linux computer has exactly one file system hierarchy. This leads to the following questions: "how does this relate to the 'multi-drive' method used by MS Windows?", and equivalently, "how do I see my CD-ROM, my floppy disk, my second hard disk, etc.?".

The answer lies in a concept called **mount points**, which are simply directories off of which are 'hung' other file systems (ie file systems which reside on a medium other than that which / resides on). For example, most Linux systems have a directory either at `/mnt/cdrom/` or at `/media/cdrom/` (or maybe even `/cdrom`) for accessing CD-ROMs. If you list the contents of this directory without a CD-ROM present, you'll see that it's just an ordinary empty directory:

```
ubuntu@ubuntu:~$ ls -laFi /media/cdrom/
total 0
 11 drwxr-xr-x 2 root root  48 2003-03-18 17:49 ./
  9 drwxr-xr-x 5 root root 128 2003-03-18 17:49 ../
```

If you then insert a CD-ROM into the drive, its contents can be **mounted** at the mount point. In some systems (including the Ubuntu machines in the departmental lab), this will happen automatically. More generally, one has to manually mount the file system, using the `mount` command:

```
ubuntu@ubuntu:~$ ls -a /media/cdrom/
.  ..
ubuntu@ubuntu:~$ mount /media/cdrom/
ubuntu@ubuntu:~$ ls -a /media/cdrom/
.    boot  home         lib         opt   sbin  var
..   dev   install.txt  mnt         proc  tmp
bin  etc   isolinux     nocompress  root  usr
```

To **unmount** a mounted file system (eg before you could eject the CD-ROM), you call the `umount` command:

```
ubuntu@ubuntu:~$ umount /media/cdrom/
ubuntu@ubuntu:~$ ls -a /mnt/cdrom/
.  ..
```

Note the spelling - it's `umount` **not** `unmount`.

5.5 Links

Links in Linux are a similar concept to 'shortcuts' in Microsoft Windows – files which point to other files. However, things are a bit more complicated in that there are two kinds of link: **soft links** and **hard links**.

A **soft link** (or **symbolic link**) is the same as a shortcut in Windows – a file which points to another file simply by specifying its path. You can use `ls -l` to see if a file is a soft link. We see two soft links in the following example: one to a file in the same directory, and one to a file elsewhere.

```
ubuntu@ubuntu:~/tmp$ ls -l
lrwxrwxrwx 1 ubuntu ubuntu    9 Oct 13 20:24 g.txt -> h.txt
-rw-r----- 1 ubuntu ubuntu 7787 Oct 13 20:23 h.txt
lrwxrwxrwx 1 ubuntu ubuntu    7 Oct 13 20:25 mycp -> /bin/cp
```

A soft link is nothing more than this: a file which 'contains' the path to another file in the file system hierarchy. If that 'pointed to' file is moved, the link will break.

Hard links are somewhat more complex. In order to understand them, you first need to know a little about **inodes**[13].

[13] inodes are examined in greater detail in CS-228 Operating Systems.

An inode is an operating system construct which tells the system where on disk (ie in which exact blocks on the disk) a file's contents are stored. It also contains other information such as who owns the file, when it was created, etc. It is useful to think of a file as consisting of two things: the actual data of the file (stored on various disk blocks across the disk), and the inode, which contains all the 'meta data' of the file, and pointers to the disk blocks.

Thus, critically, every file is associated with *exactly one* inode, which tells the system where to find the file on disk.

Now, until now you've probably thought of the path to a file as its 'unique identifier'. But I've just said that the inode uniquely identifies the file – so where does the path come in? Well, the path is just a text string which points to the inode. And if you have more than one path pointing to the same inode, you have more than one path referring to the same file.

That's what a hard link is: another path pointing to the same inode, and thus to the same file.

That's nice and simple, but it has some interesting ramifications. For one thing, notice that (unlike with soft links), a hard link to a file has the same status as the original file: it's just another path to the same thing, and other than having a different path, is indistinguishable from the original. In fact, more than that, it **is** the original. So if I create a file, then create a hard link to it, then delete the original, the file still exists on disk, because there is another path pointing to the inode. Indeed, one of the things the inode stores is the number of links to the file. This is 1 when the file is created; it is incremented every time a hard link to the file is created; it is decremented every time a link to the file is deleted. The file is only actually deleted from the disk when this counter hits 0, ie when the last link to the file is deleted.

Inode numbers can be seen using the `-i` option of `ls`. In the following example, `a.txt` and `c.txt` both refer to the same file, because they have the same inode number:

```
ubuntu@ubuntu:~/tmp$ ls -i
3506978 a.txt  3506980 b.txt  3506978 c.txt
```

Finally, why have these two ways of linking files? If hard links are so marvelous, why not always use them? The answer is that hard links don't work across file systems: if my file system hierarchy consists of two partitions of my hard disk, I can't make a hard link in one partition which points to a file in the other. To see why not, notice that inodes are identified by numbers, and the numbers are

unique *within a file system*. If we have multiple file systems mounted together, it's possible (nay, likely) that the same inode number will crop up on two different partitions of a hard disk, referring to two different inodes.

Now, If we allowed hard links across file systems, we would have to guarantee that these two inodes on these two separate file systems, really did refer to exactly the same file (same number, same file). But that's unworkable, because the second file system might, for instance, be a CD-ROM which came from someone else – its contents are beyond our control. We *can't* make such a guarantee, so we can't have hard links across file systems. But it's nice to be able to link across file systems *somehow*, and that's where soft links come in: since they only refer to a path within the currently existent file system hierarchy, there's no problem.

Phew...

5.6 The Current Directory and Its Parent

Linux uses special names to represent 'the current working directory' and 'the parent directory of the current working directory' when specifying paths to files. These special names are `.` and `..` respectively.

In the following example, notice that the files referred to by the paths `hello.txt` and `./hello.txt` are in fact the same file.

```
ubuntu@ubuntu:~/notes$ ls -l
total 12
-rw-r-----  1 ubuntu ubuntu 1066 Oct 13 19:56 Makefile
-rw-r-----  1 ubuntu ubuntu 1872 Oct 13 01:28 hello.txt
ubuntu@ubuntu:~/notes$ ls -l hello.txt
-rw-r-----  1 ubuntu ubuntu 1872 Oct 13 01:28 hello.txt
ubuntu@ubuntu:~/notes$ ls -l ./hello.txt
-rw-r-----  1 ubuntu ubuntu 1872 Oct 13 01:28 ./hello.txt
ubuntu@ubuntu:~/notes$ ls -l ..
total 12
drwxr-x---  2 ubuntu ubuntu 4096 Oct 13 19:12 coursework
drwxr-x---  3 ubuntu ubuntu 4096 Oct 13 20:14 notes
drwxr-x---  3 ubuntu ubuntu 4096 Oct  7 10:59 slides
```

5.7 Absolute and Relative Paths

A path can be either **absolute** or **relative**.

Absolute paths begin with the `/` character and unambiguously describe just one point in the file system, which can be reached by starting at the root of the file system and following the path.

Example: `/bin/cp` unambiguously points to the `cp` file in the `bin` directory which is off of `/`, the root of the file system.

Relative paths don't start with a slash, and specify a file's location relative to the **current working directory**.

Example: `bin/cp` is a relative path which points to the same file as in the previous example **if** the current working directory is `/`, but which would point to a completely different file if the current working directory was anything else, e.g. `/tmp/`.

6 Users, Groups, Ownership and Permissions

6.1 Users and Groups

Linux is a multi-user system, and every user of the system has an unique identity, loosely referred to as their **id**, **login**, **user name**, etc. When you log in, you specify this identity (along with your password), and anything you do from then on is done 'by that user'. More precisely, each user is uniquely identified by a **uid**, which is an integer, and the name we generally use is just a **user name** associated with that uid.

This enables the system to, for instance, restrict access to only those parts of the system for which you have permission – e.g. you can read and write your own files but generally can't do so with other peoples'; a more important example is that general users tend not to be able to run system administration tasks – so you can't delete the whole file system, for example.

Linux also has a concept of user **groups**; this is an effective way to grant a group of users access to some resource without explicitly (and labouriously) granting it to each user. For example, we might have a group called `printing` – all members of the group are allowed to print, and anyone who isn't in the group, is not allowed to print.

Like users, groups are uniquely identified by an integer, called the **gid**, and an associated name, the **group name**. A user can belong to many groups, but every user belongs to at least one group – typically a one-person group with the same name as the user, or a 'catch all' group which every user in his class always belongs to (e.g., the author's default group is `staff`).

Let's have a look at some of the commands related to finding out who you are and who else is around.

6.1.1 `id`

`id` prints out information on your uid and corresponding name, your default gid and corresponding names, and the gids and corresponding names of any groups you belong to, on a single line.

```
eechris@somepc:~$ id
uid=3618(eechris) gid=1001(staff)
groups=1001(staff),10(wheel),16(cron),18(audio)
```

It has various options which control what information is shown.

6.1.2 `logname` and `whoami`

`logname` prints out the user name you originally logged in as. `whoami` prints out the user name you are *currently* logged in as. It's possible (if you know their password) to change to another userid using the `su` command. If you do so, the output of `whoami` will be the name of the user you became; the output of `logname` always remains as your original login name, however.

```
eechris@somepc:~$ su gimbo
Password:
gimbo@somepc:~$ logname
eechris
gimbo@somepc:~/eechris$ whoami
gimbo
```

6.1.3 `groups`

`groups` prints out the group names of all the groups you belong to.

```
eechris@somepc:~$ groups
staff wheel cron audio
```

6.1.4 `users` and `who`

`users` prints out the user names of all the users currently logged in to the system. There will be one per session, so from the following example we can infer that I have five terminal windows open. `who` is like `users` in that it shows who's logged in, but shows a lot more information (and has many options for controlling the output).

```
eechris@somepc:~$ users
eechris eechris eechris eechris eechris
eechris@somepc:~$ who
eechris   :0           Oct 13 14:47
eechris   pts/0        Oct 13 14:47 (:0.0)
eechris   pts/1        Oct 13 19:32 (:0.0)
eechris   pts/2        Oct 13 19:34 (:0.0)
eechris   pts/3        Oct 13 15:05 (:0.0)
```

6.2 File Ownership and Permissions

As stated in section 6.1, Linux controls access to files and directories according to users and groups. In order to do this, it uses the concepts of **file ownership** and **access permissions**, which are viewed using the `-l` option of `ls`, and modified using the `chown` and `chmod` commands respectively – see Section 7.11.

```
eechris@somepc:~$ ls -l /bin/cp
-rwxr-xr-x  1 root root 50976 Aug 20 10:14 /bin/cp
```

Here, the ownership is `root root`. The first owner is the **user** who owns the file (typically the user who created it), in this case `root`, i.e. the superuser. The second owner is the **group** who owns the file (typically the default group of the user who created the file).

The reason why we have two owners (one user, one group) becomes apparent when we look at permissions. The `-rwxr-xr-x` string tells us the file's permissions, and is split up in the following manner:

- The first character tells us the file type, and can have the following values: `-` is a regular file; `d` is a directory; `l` is a link; `c` is a special file; `s` is a socket; `p` is a named pipe.

- The next three characters tell us if the user who owns this file is allowed to read, write, and execute the file (indicated by `r`, `w`, and `x` respectively). As we can see, the user `root` is indeed allowed to read, write, and execute this file.
- The next three characters tell us the same thing for the group who owns this file – here members of the group `root` can read and execute the file, but can't write to it (ie they can't modify it).
- The final three characters tell us the same thing for *everybody else* – so anyone can read and execute, but can't modify, this file.

There are some extensions to this – e.g., if the file is in fact a directory, 'execute' means permission to move into that directory, 'read' means permission to list its contents, and 'write' means permission to create and delete files. There are also things called 'the sticky bit' and 'the setuid bit', whose exploration we leave as an exercise for the reader. See the `chmod` man page or one of the recommended books for more information.

7 Commands For File System Navigation

Now that we've seen the general concepts behind the Linux file system, let's take a closer look at some of the commands used to work with files, directories, and the file system.

7.1 `ls`

As we've already seen, `ls` lists the contents of a directory.

- `ls` without specifying a directory lists the contents of the current directory. Otherwise it will list all of the files and directories named on the command line.
- The `-l` option gives you 'long' information on the files – ownership, permissions, last change date, etc.
- The `-i` option tells you the inode of each file.
- The `-1` option prints one filename per line.

- By default, `ls` doesn't list files whose names begin with the `.` character, deeming them to be 'hidden'. The `-a` option prints all files, including these.

  Note that a lot of programs will store configuration information in hidden files or directories in your home directory. For example, without hidden files, my home directory contains 17 items; with them, it contains 144.

- The `-F` option appends to each filename a character which tells you the 'type' of that file (regular, directory, etc.)
- The `-R` option lists directories recursively.
- The `-d` option lists directory names but not their contents.
- The `-t` option sorts the output according to time of last modification.
- There are many other options – read the man page.

7.2 `cd`

`cd` changes directory. It's used in one of two ways:

- `cd` with one parameter, which may be an absolute or relative path, changes into the specified directory.
- `cd` on its own, without any parameters, changes automatically to your home directory.

7.3 `pwd`

`pwd` tells you your present working directory, ie the path to the directory whose contents would be listed if you just typed `ls` without any parameters. There's not really anything else to say about it.

7.4 `pushd` and `popd`

These are handy – they're like `cd` except they maintain, between them, a **stack** of directories visited, so you can easily get back to where you were before. If you're working in some directory whose name you can't remember, but want to do something in your home directory briefly, you can type `pushd ~`, do whatever it is you need to do, then type `popd` to return to where you were before.

7.5 cp

`cp` is used to copy files.

- The basic usage form is `cp src dest` which copies the file with path `src` to the path `dest`. If `dest` is a path to a directory, the newly created file has the same name as `src`; if not, its name is specified by the last part of `dest`.
- Many files can be copied to a directory using the following form, which only works if the last parameter is indeed a directory:

  ```
  cp src1 src2 ...srcn dest_dir
  ```

- `cp` skips source files which turn out to be directories, unless you specify the `-r` or `-R` option, in which case directories are copied recursively.
- `cp` copies silently, unless you specify the `-v` option (for **verbose** - a very common option), in which case it tells you about every file it copies.
- There are many other options – read the man page.

7.6 mv

`mv` is used to move and rename files.

- Its usage forms are similar to `cp` – basic usage consists of moving a file from one path to another (possibly a directory), but you can also move multiple files to some directory by specifying the files to be moved then the target directory, just like with `cp`.
- `mv` has a small number of options; small enough to be read in the man page in no time at all...
- Moving something within a given file system is a very quick operation, because all that really has to change are the references to the file. If you move something across file systems, however, it will take longer because the data has to copied to the destination file system and deleted from the source file system.

7.7 `mkdir`

`mkdir` creates a new directory with a given name. It has the following interesting options:

- `-m` specifies the **mode** of the directory being created – this is the permission information discussed in Section 6.2, and must be specified in the same manner as in `chmod`.
- `-p` specifies that parents of the directory to be created should also be created if they don't already exist.

  For example, if `/tmp/` doesn't already contain a directory called `hello`, then `mkdir /tmp/hello/mydir` would fail **unless** `-p` was specified, in which case `/tmp/hello` would be created too.
- `-v` tells `mkdir` to be verbose (normally it's quiet).

7.8 `ln`

`ln` is used to create links, both soft and hard (see section 5.5).

- `ln src dest` will create a hard link at the path `dest`, pointing to the same inode as `src`. It will fail if `src` and `dest` are on different file systems or if `dest` exists already.
- `ln -s src dest` will create a soft link at the path `dest`, pointing to the path `src`. It will fail if `dest` exists already.
- The `-f` option can be used to **force** creation of the new file, overwriting anything which might have existed there before. `-f` is another one of these common options that keep cropping up everywhere.
- `ln` has several other options, which can be found out about by reading its man page.

7.9 `rm` and `rmdir`

`rm` is probably the most dangerous command you can use, since using it you can delete all your files very easily, and unlike Windows, Linux **does not** have a

recycle bin[14] so if you delete something, it's gone for good. Don't bother asking the technicians to get it back – they can't.

- Basic usage: `rm file1 file2 ...filen` removes the files specified on the command line.
- The `-i` option puts `rm` into interactive mode, in which it asks for confirmation of every delete. This can be tedious, but it can also be a lifesaver.
- `rm` skips directories, unless you tell it to recurse into directories using the `-r` option.
- `rm` usually operates silently (which can be scary if it takes ten seconds and you thought it was only removing one thing), but you can tell it to be verbose using, you guessed it, the `-v` option.
- `rm` has several other options, which can be found out about by reading its man page.

`rmdir` is a related command, dedicated to removing directories, but is in some ways redundant in that everything it does, `rm` can do.

Note that, as mentioned in section 5.5, `rm` actually removes *references* to files, and the actual file is only deleted from disk when the last reference is removed. Since most files only have one reference (ie no hard links), this is usually what you expect; sometimes, however, you might think you've deleted something when you haven't in fact. However, you can see how many links a file has using the `stat` command – see section 7.12.

7.10 `touch`

`touch` performs two functions:

- If you `touch` a file which already exists, the time stamp of the file is updated to now.
- If you `touch` a file which doesn't exist, it is created as a zero length file (ie a file with no content). You'd be surprised how often this can come in handy...

[14] Actually, desktops such as GNOME typically do have a recycle bin for things you delete via the GUI, but anything you delete using `rm` bypasses it, so...

Of course, `touch` has a number of options, which you are encouraged to explore in the usual manner.

7.11 `chown` and `chmod`

`chown` and `chmod` are used to modify file ownership and permissions, respectively. In order to use either of these on some file, you must yourself have adequate permission, of course.

- To use `chown` to change a file's owner (user only): `chown username filename`
- To use `chown` to change a file's owner (both user and group): `chown username.groupname filename`
- `chmod` is a little more complicated since it has more work to do. In particular, you can specify the changes to a file's permissions in two ways: as **octal** digits and **symbolically**.
- In either case, the basic usage is `chmod perms filename`
- **Symbolic** mode is easiest to understand. The new permissions are specified as a string consisting of a number of letters representing who you're referring to (`u` for user-owner of the file, `g` for group-owner of a file, `o` for 'others', and `a` for 'all'), a symbol representing the type of change to make (+ to add permissions, − to remove permissions, and = to set these to be the only permissions on the file), and finally a string of letters representing the permission in question (`r` for read, `w` for write, `x` to execute).

  This sounds complicated, and is best illustrated by example:

  - `chmod a+x somefile` grants the execute permission to everybody.
  - `chmod og-w somefile` removes write permission from the owner-group and from 'everyone else on the system', if they had it.
  - `chmod ugo=rx somefile` specifies that everyone can read and execute the file, and nobody can write it.
- **Octal** mode specifies permissions using from one to four octal digits (0-7), derived by adding up the bits with values 4, 2, and 1. Any omitted digits are

assumed to be leading zeros. The first digit selects the set user ID (4) and set group ID (2) and sticky (1) attributes. The second digit selects permissions for the user who owns the file: read (4), write (2), and execute (1); the third selects permissions for other users in the file's group, with the same values; and the fourth for other users not in the file's group, with the same values.

Note that with octal mode, only absolute changes can be made, and relative changes (eg the change affected by `chmod a+x`) are impossible.

- `chmod 0555 somefile` has the same effect as the final example above (`chmod ugo=rx somefile`).

- Both `chown` and `chmod` accept the `-R` option, which causes them to operate on a directory recursively.

- As usual, see the man pages for more details.

7.12 `stat`

`stat` displays file status information – size, blocks, number of links, access permissions, user, owner, time of last read, time of last modification, etc.

- The `-f` option, instead of displaying status information for the specified file, displays status information for the file system upon which that file resides, eg its total capacity, current usage, etc.

- Note that using `stat` to find out (for instance) when a file was last accessed doesn't actually access the file, and thus leaves the last accessed time unchanged ; it just accesses the inode.

```
eechris@somepc:~$ stat b.txt
  File: 'b.txt'
  Size: 1672            Blocks: 8
        IO Block: 4096  Regular File
Device: eh/14d  Inode: 606987      Links: 1
Access: (0644/-rw-r--r--)
   Uid: ( 3618/  eechris) Gid: ( 1001/staff)
Access: 2003-10-13 18:12:25.000000000 +0100
Modify: 2003-10-08 17:46:31.000000000 +0100
Change: 2003-10-08 17:46:31.000000000 +0100
```

```
eechris@somepc:~$ stat -f /
  File: "/"
    ID: 0        Namelen: 255      Type: ext2/ext3
Blocks: Total:     1963387 Free: 865969
        Available:  766235 Size:   4096
Inodes: Total:      997472 Free: 735157
```

7.13 du

`du` tells you about **disk usage**, ie how much space a file or directory is taking up. It's very handy indeed.

- Basic usage: `du filename` tells you how many blocks a given file is using on disk. If the file is in fact a directory, `du` will recurse into it and tell you how many blocks each of its subdirectories is taking up, and *then* tell you the total for that directory.
- Since `du` recurses on directories, its output can get quite long. You can tell `du` to only report total sizes up to some depth using the `--max-depth` option. The sizes of the subdirectories are still counted, of course, but their 'intermediate' totals aren't reported.

  A value of 0 will just report the final total,
- The `-h` option tells `du` to produce sizes in 'human readable' format (ie kilobytes, megabytes, etc.) rather than as simple block counts[15].
- The `-x` option tells `du` to stick to just one file system, ie don't traverse any mount points.
- `du` has several other options, which can be found out about by reading its man page.

```
eechris@somepc:~/research$ du --max-depth 1 -h
20K     ./CVS
140K    ./misc_notes
57M     ./tpl
177M    ./mphil
```

[15] A block typically corresponds to 512 or 1024 bytes, depending on the file system used.

```
8.9M    ./articles
242M    .
```

7.14 `which` and `find`

When you run `cp`, exactly what is being run? The answer can be found using the `which` command:

```
eechris@somepc:~$ which cp
/bin/cp
```

How does it know it's there? The answer is that it is searching all of the directories in the shell's **environment variable** `$PATH`. We'll talk more about environment variables and the path in a later section.

`find` is a **very** powerful command for finding files based on their name, modification time, file type, etc. It is, in fact, way too complicated to fully describe here, but we can show a couple of examples of its usage:

- `find` on its own will just print a list of every file off the current directory (ie everything in this directory and any of its subdirectories, recursively). This can be handy, especially combined with piping and `grep` – see later.
- `find . -name 'hello.txt'` will look for a file called `hello.txt` off the current directory (`.`), searching in subdirectories recursively.
- `find /tmp -name '*.txt'` will list all files whose names end in `.txt` off the directory `/tmp/`, searching recursively.
- (Advanced) `find . -name '*.txt' -exec rm -vf {} ';'` will find all files whose names end in `.txt` off the current directory, searching recursively, and **delete them**.
- (Advanced) `find . -type d -name 'no*' -links 3` will find all directories whose name starts with `no` and which have 3 links to them, off the current directory, recursively.
- If you want to know more, eg how these last two work, well, have a look a the manual page – it's a classic.

7.15 `df`

As discussed in section 5.4, a Linux file system hierarchy is built by starting with a 'root' file system, and then mounting other file systems on mount points so that they appear to be part of the root file system.

The **df** command can be used to see exactly what file systems are mounted on which mount points.

```
eechris@somepc:~$ df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/root             7.5G  4.2G  3.0G  59% /
/dev/hda3             9.9G  6.4G  3.6G  65% /usr
/dev/hda5             5.0G  1.2G  3.6G  25% /home
/dev/cdroms/cdrom0    223M  223M     0 100% /mnt/cdrom
```

Here we have the hard disk's second partition (represented by `/dev/hda2`) mounted at `/`, and a couple of network file systems. We'll meet network file systems again in CS-228 Operating Systems.

The `-h` option is probably the most useful: it prints file system capacities in human-readable form, as opposed to as a count of disk blocks occupied, which is somewhat less friendly.

8 Input/Output and Redirection

8.1 stdin, stdout, stderr

When a running process wants to read from, or write to a file, it has to open that file (we'll see lots of this when we study the C programming language in EG-244). Specifically, opening a file returns to the process a **file handle**, and it is this that the process manipulates in order to perform I/O with the file.

Now, *every* process you run in a Unix system gets three file handles 'for free'. They are called **standard input**, **standard output**, and **standard error**.

- **standard input**, written `stdin` is a read-only file handle from which the process can read input. Generally this comes from the keyboard, but as we'll see shortly, it can come from other sources (eg a file or the the `stdout` of another process).

- **standard output**, written `stdout` is a write-only file handle to which processes can write output. Generally it then appears on the screen (ie as text in your shell), but as we'll see shortly, it can be directed elsewhere (eg a file or the `stdin` of another process).
- **standard error**, written `stderr` is another write-only file handle to which processes can write output, and which is generally used for reporting error information. The reason for this separate stream is that if `stdout` is redirected to somewhere other than the console, any errors written there could go unseen.

Note that a process doesn't have to use any of these file handles. However, ones which do fit very well into the Unix way of achieving code reuse and interoperability, as we shall see.

8.2 Redirection

8.2.1 Redirection of output

As a trivial example of all this, it's worth noting that `ls` doesn't specifically write its output to the screen – instead, it writes its output to `stdout` , which by default means it goes to the screen.

However, it's possible to **redirect** the output to a file on disk, instead. In the following example, we redirect the listing to a file called `file.txt`, which we subsequently read using `cat`. Redirection is achieved using the > character.

```
eechris@somepc:~$ ls -l > files.txt
eechris@somepc:~$ cat files.txt
total 172
drwxr-x---  4 eechris staff 4096 Sep 21 19:39 admin
drwxr-x--- 18 eechris staff 4096 Sep 26 10:38 apps
drwxr-x---  6 eechris staff 4096 Sep 17 12:07 cvsroot
-rw-r-----  1 eechris staff    0 Oct 20 17:13 files.txt
drwxr-x---  3 eechris staff 4096 Feb 11  2003 prog
drwxr-xr-x 14 eechris staff 4096 Oct 16 20:05 public_html
drwxr-x---  7 eechris staff 4096 Jul 11 11:34 research
drwxr-x--- 12 eechris staff 4096 Oct 16 16:49 teach
```

8.2.2 Redirection of input

In fact, the same is true of input: in the above example `cat` reads `files.txt` because we specified it on the command line; as such, `cat` ignores its standard input. However, we can instead **not** specify a file on the command line, in which case `cat` reads from standard input. What happens if we try it?

```
eechris@somepc:~$ cat
I am typing this line on the terminal.
I am typing this line on the terminal.
I type, cat reads from stdin, cat writes to stdout.
I type, cat reads from stdin, cat writes to stdout.
I hit Ctrl-C to exit.
I hit Ctrl-C to exit.
```

So we can see that all that `cat` actually does is read from `stdin` and write to `stdout` . But what if want to direct a file at `stdin` ? Well, we use the < character:

```
eechris@somepc:~$ cat < files.txt
total 172
drwxr-x---  4 eechris staff 4096 Sep 21 19:39 admin
drwxr-x--- 18 eechris staff 4096 Sep 26 10:38 apps
drwxr-x---  6 eechris staff 4096 Sep 17 12:07 cvsroot
-rw-r-----  1 eechris staff    0 Oct 20 17:13 files.txt
drwxr-x---  3 eechris staff 4096 Feb 11  2003 prog
drwxr-xr-x 14 eechris staff 4096 Oct 16 20:05 public_html
drwxr-x---  7 eechris staff 4096 Jul 11 11:34 research
drwxr-x--- 12 eechris staff 4096 Oct 16 16:49 teach
```

There's a subtle but important point here: `cat files.txt` and `cat < files.txt` look very similar but they're doing very different things. In the first version, `cat` is told it has to open `files.txt` and write its contents to its `stdout` — you could say it knows the name of the file it's working on; in the second version, all that `cat` does is read from its `stdin` — it has no idea what file it's working on. In fact, in the second case, it is the shell, `bash`, that 'knows' we are dealing with `files.txt` – the shell opens that file and writes its contents to `cat`'s `stdin` .

8.3 Pipes

We've seen that a process can redirect its output to a file, and it can also redirect its input so that comes from a file. We saw earlier that `find` can easily produce a lot of text, so why not take advantage of our new skills? The following works just fine:

```
eechris@somepc:~$ find > allfiles.txt
eechris@somepc:~$ less allfiles.txt
```

However, why bother with the intermediate file? Why not just redirect `find`'s `stdin` to `less`'s `stdout` [16]? Well, it turns out we can do exactly that, but we don't use the redirection characters; it's called **piping** and it looks like this:

```
eechris@somepc:~$ find | less
```

The `|` character sets up the pipe: everything `find` writes to its `stdout` immediately appears at the `stdin` of `less` (so note that `less` doesn't have to wait for `find` to finish).

Note that like redirection to and from files, piping is a shell facility, and the pipe in the above example is managed by the shell within which we are running `find` and `less`.

Piping is an absolutely critical aspect of using Unix – very often an apparently complicated task can be broken down into a series of small tasks connected by pipes.

8.3.1 `tee`

Sometimes you want to redirect output to a file but also see it on the screen (or to some other command for further processing). For this the `tee` utility is handy: whatever it reads from its `stdin` is copied both to its `stdout` and to a file on disk. Thus, the following line is like the above, except that the output also gets stored to `filelist.txt`:

```
eechris@somepc:~$ find | tee filelist.txt | less
```

This is also a nice example of chaining pipes together.

[16] Of course, this is making the mental leap that perhaps `less` works like `cat` in that it can read from `stdin` – and this is indeed a good leap to make

9 Process Management

Every time you run a program in Linux it creates a **process** within which that program's code is run. Exactly what constitutes a process is something we'll examine in CS-228, but for now the main point is that a process is a running program. We said earlier that everything in Unix is a file or a process, so clearly processes are important.

9.1 Process Utilities

There are a number of utilities which allow us to see and manipulate the processes running on a system – let's take a look at some of the important ones. It's important to know the following general points:

- Every process gets an unique ID number, called its **process id** or just pid.
- Processes can create other processes, which are then their **children**. For example, when you run `ls` in a `bash` shell, `ls` is running as a child process of `bash`. Thus, a **process hierarchy** is formed.
- There is always a process called `init` with pid 1 — this is the 'root' process of which all others are children.

9.1.1 `top`

`top` shows you the top running processes on your machine, ordered by CPU usage (descending). It updates its output every two seconds, so you can use it to keep track of what's happening, and see if you have any 'runaway' processes eating all the CPU.

Stuff worth knowing:

- Hit `q` to quit.
- Hit `m` to order by memory usage instead.
- Hit `r` to 'renice' a process interactively (see below).
- Use the `-n` option to specify the number of iterations `top` should run for before quitting; the default is infinite (ie wait for the user to hit `q`).

- Use the `-b` to put `top` into 'batch' mode — it doesn't update the screen, it just writes its output to standard output. This is best used in conjunction with `-n`. Try it out and see the difference.
- Use the `-d` option to change the delay time from the default of 2 seconds.
- Lots of other funky stuff – read the `man` page, people.

9.1.2 `ps`

`ps` reports process status, ie what processes are running, who owns them, etc. It's a bit like `top` running in batch mode, but more lightweight. Like `top` (and a lot of these utilities) it has many many options to control how it's used.

`ps` on its own just displays the processess associated with the current session (ie shell). This might be just two things as in the example below, or it could be a longer list if processes were running in the background (see section 9.2).

```
eechris@somepc:~$ ps
  PID TTY          TIME CMD
31523 pts/3    00:00:00 bash
31547 pts/3    00:00:00 ps
```

To see every process currently running, use `ps aux`.

`ps` is often used in conjunction with `grep` (see section 10.5) to pick out a particular process:

```
eechris@somepc:~$ ps ax | grep bash
 3400 pts/0    S      0:00 bash
30008 pts/2    S      0:00 bash
31402 pts/1    S      0:00 bash
31523 pts/3    S      0:00 bash
31633 pts/3    S      0:00 grep bash
```

9.1.3 `pstree`

`pstree` is a nice utility for seeing the 'tree' of processes alluded to earlier (ie parent/child relationships). It's often handy to run it with the `-p` option to print process numbers too.

9.1.4 `nice` and `renice`

Linux is a multi-tasking operating system, so many processes can be running at once. Of course, only one is really running at a time; the **scheduler** (part of the kernel) decides which process runs next depending on a number of factors[17].

By default, a process runs as often as it can. However, it is possible in Unix/Linux for a process to 'be **nice**' to other processes, and be a little less competetive for resources. This is particularly handy for processes which may run for a long time and use a lot of CPU, and which will normally slow the machine down so that you can't do anything else: if you make the process nice, you'll find that it still runs (albeit a bit slower), but that while it's running the rest of the system remains usable.

You can make a process nice when you create it (using `nice`) or you can change the 'nice value' of a running process using `renice`. In order to do the latter, you need to know the pid of the process (which `top` or `ps` will tell you).

```
eechris@somepc:~$ nice find | less
...
eechris@somepc:~$ ps ax | grep bash
 3400 pts/0    S      0:00 bash
30008 pts/2    S      0:00 bash
31747 pts/1    S      0:00 bash
31759 pts/1    S      0:00 grep bash
eechris@somepc:~$ renice 10 31747
31747: old priority 0, new priority 10
```

See the `nice` and `renice` `man` pages for more information about niceness and how it can be modified.

9.1.5 `kill`

If you have a process which you wish to terminate, and you know its pid, you can kill it using the `kill` command. We'll see exactly how `kill` works shortly, but for now note that there are (at least) two ways of using it.

The first version is simply `kill pid`, which will attempt to kill the process 'cleanly', ie it will allow the process to close any files it has open, free up memory, etc. **You should always try this first.**

[17] These factors, and some scheduling algorithms, will be explored in CS-228

```
eechris@somepc:~$ emacs &
[1] 31769
eechris@somepc:~$ kill 31769
eechris@somepc:~$
[1]+  Terminated              emacs
```

This method works by sending the process a `SIGTERM` (**terminate**) signal. The process should catch this signal and act accordingly; in CS-228 we'll see how to write processes which catch signals like this.

The second method is `kill -9 pid` – the `-9` option tells `kill` to completely obliterate the process without giving it chance to clear up its resources. **This should only be used as a last resort when an ordinary kill pid has failed**, for reasons which should be obvious.

This method works by sending the process a `SIGKILL` (**kill**) signal, which the process can't catch and which causes it to die immediately. Again, we'll see more of signals in CS-228; the general rule at this point though is, if a process receives a signal, it probably terminates, and `kill` is what you use to send signals to processes.

After running kill, you can check if the process died by running `ps` and piping it to `grep`, as seen above.

9.2 Process Control

Ordinarily when you run a process in a shell, it runs in the 'foreground', meaning it prevents you from running any other process in that shell while it's running. There are two ways you can manage this:

- You can temporarily **suspend** the process by hitting `Ctrl-Z` (ie hold down `Ctrl` then hit `Z`). This will then print something along the lines of `[1]+ Stopped process_name`. At this point the process is stopped, not running, awaiting your command to start up again.

  You can bring it back to the foreground (**resume** it) by entering `fg`. If you suspend more than one process, `fg` resumes the most recently suspended, unless you also specify the 'job number', which is 1 in the example above.

  You can type `jobs` to list currently suspended jobs and their job numbers.

- You can tell a process to run in the background when you create it by finishing the command line with the `&` character. This starts the process up, but then allows you to immediately enter another command.

This is particularly useful when you're **launching** a long-running application with a GUI seperate from the shell. For example, you could launch the Netscape web browser by typing `netscape` without a `&`, but if you do, the shell you launched it from is locked until you quit Netscape. On the other hand, if you type `netscape &`, it launches in the background and you can carry on using the shell. You can even quit the shell without Netscape dying, in fact.

Note that all of these things – `Ctrl-Z`, `fg`, `jobs`, and `&` are features of the **shell**.

10 Text Files

Unix and hence Linux have a long tradition that plain text is an excellent format in which to store your data. The reasoning is that plain text (as opposed to binary data) is readable by human beings and computer programs, whereas binary data is only readable by computer programs (unless the human being in question spends too much time programming). Using plain text may result in data files which are larger than their binary equivalents, and which take longer to read/write than their binary equivalents, but storage space and processing speed are continually getting cheaper and more abundant, whereas a human being's ability to read binary is pretty much a constant. Furthermore, promoting text as a "lingua franca" means you can concentrate on writing good utilities to process your text, and then hook them together to get Serious Problems solved.

That was the idea, anyway. It didn't quite work out that way (eg there are only a very few text-based file formats for storage of image or sound data – though they do exist), but the fact remains that text is one of the fundamental types of data you will come across when computing, and this is doubly true in Unix. Thus, Unix (and Linux) provide some excellent facilities for processing text.

10.1 What makes a text file a text file?

A text file is characterized by the following:

- Its contains only certain bytes of binary data (ie the ASCII characters[18]).

[18] By the way, you can see an ASCII table by typing `man ascii`

This isn't always the case, however – there can sometimes be odd 'out of range' characters (eg special characters in non-English alphabets such as Nordic) which will display in various manners depending on how you read the file; furthermore, Unicode text files don't fit into this category at all, but alas, Unicode is beyond the scope of this course.

- The contents are *lines* of text, ie they are separated by a **newline character**.

  In Unix text files, the end of a line is signified by a single character, having ASCII value 10 and written as `\n`. Thus, the following stream of characters represents the line `Be excellent` followed by the line `to each other.`

  ```
  Be excellent\nto each other.
  ```

  Note that this is **different** to the situation in the DOS/Windows world, where the end of a line is signified by **two** characters: **carriage return** and `line feed`, written `\r\n`. Because of this, when transporting text files manually between Unix and Windows, one should use the `dos2unix` and `unix2dos` utilities to convert the line endings[19].

10.2 `cat`, `tac`, and `rev`

We saw `cat` earlier – we used it to print the contents of a text file to a screen. But why `cat` and not, say `type`? The answer is that `cat` is short for **concatenate**, and that is in fact what `cat` does.

For example, if a directory contains the text files `a` `b` and `c`, then the command `cat a b c` will print all three files, one after the other, to the screen.

What exactly is happening here? The answer is that `cat` opens the first file, reads the first line (ie reads up until the first newline character), and then prints that line out to **standard output** (which we'll look at in section 8.1). It does this for each line in `a`, then it does the same for `b`, then `c`.

There are two related utilities:

- `tac` is like `cat` but each file is printed in reverse order (last line first).
- `rev` is like `cat` but each line is printed in reverse (ie right to left).

[19] Ubuntu users have to separately install `unix2dos` and `dos2unix` by using the command `sudo apt-get install tofrodos`. Network transport protocols such as FTP, however, can tell which one to use on the host system, so if you're transporting via FTP you shouldn't need to do this.

10.3 `less` and `more`

`less` and `more` are so-called **pager** utilities — they take as input a text file, and they allow you, the human, to read that file one page at a time. `more` is the older of the two, and in its most basic form simply pauses after a page of text, displaying the word '`more`' and waiting for you to hit a key before continuing. `less` came later as a more advanced form, allowing you to move backwards as well as forwards, move one line at a time, search for text, etc. `less` is generally to be preferred but `more` is similar and these days has many of the extra features included.

Basic usage is simple: `less some_file` will read in the file and allow you to page through it. You can use the cursor keys to navigate, hit `q` to quit, and hit `/` to search forwards for some text. That's all you need to know 95% of time time, but of course the `man` page can tell you all sorts of other useful key presses (eg how to search backwards).

10.4 `head` and `tail`

`head` and `tail` let you examine the start and end of documents: `head foo.txt` displays the first 10 lines of `foo.txt`, and `tail foo.txt` displays the last 10 lines. You can specify the number of lines to read using the `-n` option: `head -n 20 foo.txt` displays the first 20 lines of `foo.txt`, for example.

`tail` has a couple of other useful options:

- If you specify the number of lines with a + in front of it, you can display the file *from that point on*. This is sort of the opposite to `head`, in that if `head -n 10` displays the first 10 lines, `tail -n +10` displays *everything except* the first 10 lines.
- The `-f` or `--follow` option causes `tail` to 'watch' the given file for new lines of text: when a new line appended to the file, `tail` will display it. You can quit by hitting `Ctrl-C`.

  This is most useful for watching log files: you've written a web server, say, and want to watch for hits; you tell the server to log incoming hits to `/myserver.log` and then use `tail --follow /myserver.log` to watch the incoming hits as they occur. Very handy.

10.5 Pattern Matching and `grep`

`grep` is one of the most useful tools you can learn to use in Linux; using it you can filter `stdin` for some text. Even more, you can specify the text to filter on using **regular expressions**, a language for specifying how a line should look: lines which 'match' are kept (or discarded); lines which don't match are discarded (or kept).

If you've used a DOS command prompt before, you've come across pattern matching in the form of `*` and `?` in filenames. You can use these in a shell too:

```
eechris@somepc:~$ ls *.txt
allfiles.txt  filelist.txt  files.txt
eechris@somepc:~$ ls files.tx?
files.txt  files.txx
```

The first example lists all files whose name ends in `.txt` — the `*` character matches any string of characters; in the second example, the `?` character matches any **single** character. Another useful form is `[l-u]` which matches any single character in the range, **l**ower to **u**pper.

10.5.1 grep

`grep` takes a pattern and (optionally) a file on its command line, and searches for lines in that file which contain that pattern; if no pattern is specified, it reads from `stdin` (so it's **very** usable in pipelines).

In fact, you have to specify the `-e` option to `grep` in order to make it use regular expressions as seen above. In the simplest case, just specify the word you want to look for. For example:

```
eechris@somepc:~/sort$ cat grepme.txt
Here is a file which contains a number of lines, each
of which contains a number of words.  We'll use grep
to seek for word so every line which contains the word
'word' should be printed out.  That's the idea,
anyway.

grep's much more powerful than this, of course, but
90% of the time this is all you use it for.  :-)
eechris@somepc:~/sort$ grep word grepme.txt
```

```
of which contains a number of words.  We'll use grep
to seek for word so every line which contains the word
'word' should be printed out.  That's the idea,
```

Some of `grep`'s most useful options:

- `-i` to switch case sensitivity **off**.
- `-r` to search a directory recursively. This doesn't make sense unless you specify a directory on the command-line, of course.
- `-n` to print the line number of each matched line.
- `-v` to invert the match; if we specified this in the above example we would see every line **except** the ones which contain the word 'word'.
- `-c` to print a count of matched lines, rather than printing each matched line.

Regular expressions, the "engine room" of `grep`, is a big, big topic [20] which we will explore further in EG-259. For now, there appears to be a good on-line tutorial to `grep` here:

`http://pegasus.rutgers.edu/~elflord/unix/grep.html`

10.6 `wc`

`wc` (standing for 'word count') counts characters, words, and lines in a text file.

- `wc -c` counts bytes;
- `wc -m` counts characters (why is this different from bytes?);
- `wc -l` counts lines (ie strings separated by newline characters);
- `wc -w` counts words (ie whitespace separated strings);
- `wc` without any option counts characters, words, and lines;
- `wc -L` prints the length of the longest line

[20] The standard reference is Jeffrey E. F. Friedl, **Mastering Regular Expressions**, 2nd Edition, *O'Reilly Media Inc.*, July 2002.

10.7 `sort` and `uniq`

`sort` takes a text file as input and produces as output that same file but with the lines reordered such that they are in lexicographic order.

For instance, if the input looks like this:

```
eechris@somepc:~/sortexample$ cat sortme.txt
thing
Words
345
book
Hello
123
12
eechris@somepc:~/sortexample$ sort sortme.txt
12
123
345
Hello
Words
book
thing
```

`sort` has many options controlling, for instance, whether it is case sensitive or not, but that's its basic usage.

`uniq` looks for *successive* identical lines in its input, and keeps only one such line for each group it finds.

```
eechris@somepc:~/uniqexample$ cat uniqme.txt
thing
word
word
something
Here is a line containing the word 'word'.
word
another
another
another
boogy
```

```
with
with
stu
eechris@somepc:~/uniqexample$ uniq uniqme.txt
thing
word
something
word
another
boogy
with
stu
```

Again, `uniq` has a number of options, of which the most interesting are probably `-d` to only print the repeated lines, and `-s` to allow you to skip the first n characters on each line when making the comparison.

10.8 `cmp` and `diff`

`cmp` and `diff` allow you to compare (and find the differences between) two files.

`cmp` compares two files byte-by-byte and will tell you where the first difference occurs. This is actually just as useful with binary files as with text, in fact. If the two files match, it produces no output.

`diff` is considerably more powerful: it compares two text files and attempts to print the differences, line by line. There are a number of output formats selectable by command-line options.

```
eechris@somepc:~/diffexample$ cat a.txt
thing
book
345
Words
Hello
123
12
eechris@somepc:~/diffexample$ cat b.txt
thing
boing
```

```
345
Words
Hello there
12345
12
eechris@somepc:~/diffexample$ diff a.txt b.txt
2c2
< book
---
> boing
5,6c5,6
< Hello
< 123
---
> Hello there
> 12345
```

10.9 `od` and `xxd`

`od` prints an octal (or hex, etc.) dump of a file; `xxd` does a similar thing but also prints the contents as if they were ASCII on the same line. They can both be useful for debugging purposes, and for seeing **exactly** what's in a file. For instance, can you spot the newline (`\n`) characters in the first three lines of `numbers.c` in the following?

```
eechris@somepc:~$ head -n 3 numbers.c | od
0000000 064443 061556 072554 062544 036040 072163 064544 027157
0000020 037150 005012 064143 071141 020052 072557 063164 066151
0000040 020145 020075 067042 066565 062542 071562 072056 072170
0000060 035442 000012
0000063
eechris@somepc:~$ head -n 3 numbers.c | xxd
0000000: 2369 6e63 6c75 6465 203c 7374 6469 6f2e  #include <stdio.
0000010: 683e 0a0a 6368 6172 2a20 6f75 7466 696c  h>..char* outfil
0000020: 6520 3d20 226e 756d 6265 7273 2e74 7874  e = "numbers.txt
0000030: 223b 0a                                  ";.
```

10.10 `md5sum`

`md5sum` calculates a 'hash value' for a file – a compact string representing some large number with the handy property that two different files will[21] not have the same hash value. It's often used as a checksum when downloading large files from the Internet: you download the file, calculate the `md5sum` value of the file you just downloaded, and compare it with a known good value published on the website: it they match, your file's OK; if they don't match, it's corrupt.

```
eechris@somepc:~$ md5sum files.txt
6665e51fe3f5d483767da2bcb49e8597  files.txt
```

11 Printing Utilities

We should mention some utilities which are useful for printing.

11.1 `enscript`, `nl`, and `fold`

`enscript`[22], `nl`, and `fold` are useful utilities for preprocessing code before printing. The `enscript` utility is the most powerful but we mention the others for completeness.

`enscript` takes a text file as input and 'pretty-formats' it into a Postscript file ready for printing. This includes wrapping long lines sensibly. Its basic usage is `enscript -o output.ps input` – if you don't specify an output file using `-o` it will send the document straight to the printer. Of course, `enscript` has a number of options – see the `man` page – but the most interesting are:

- `-U n` to put `n` pages of input onto a single page of output. This is called 'n-up' printing; for example, these notes have been printed using a 2-up scheme.
- `-b banner` to set the title/banner of the page to `banner`. This includes the ability to include variables such as the name of the document, page number, date, etc.

[21] Or rather, should, within the lifetime of this universe.

[22] The `enscript` utility is not installed on Ubuntu or on the Live CD by default. To install it, issue the command `apt-get install enscript`.

- `-C` to number each line in the output.

A typical usage when formatting a C program for submission as coursework would be:

```
enscript -b "CS-244|\$N (\$% of \$=)|Andy Gimblett 264924" \
   -o cw.ps -C cw.c
```

Exercise: Try this out.

`nl` does line-numbering in plain text files; `fold` wraps input lines to a maximum length.

11.2 psnup

We saw how `enscript` can format its output in 'n-up' format; it's worth knowing that you can do this to any Postscript file using the `psnup` utility:

```
psnup -m20 -d1 -2 input.ps output.ps
```

11.3 gv

`gv` is the Ghostscript Postscript previewer - a graphical previewer for Postscript files. Once you've used `enscript` to generate nicely formatted Postscript versions of your work, use `gv` to check that everything looks sane before printing.

11.4 lpr

Print your work using `lpr`, the Unix 'line printing' utility.

```
lpr cw.ps
```

will, if run on one of the Linux machines in the Computer Science Linux lab, print `cw.ps` on the printer in that room.

- You can specify the number of copies to print using the `-#` option.
- You can specify another printer to print on using the `-P` option – but you have to know its name in order to do this.

12 More On Bash

12.1 Environment Variables

Environment Variables are named variables associated with a particular shell session, which are used for a number of purposes.

12.1.1 Setting an Environment Variable

Environment variables are assigned using the = character as illustrated below. To use a value containing spaces, enclose it in double quotes:

```
MYVAR=Hello
MYVAR="Hello There"
COUNTER=1
```

Note that in all three cases, the value of the variable has type 'string'. Note also that whitespace is significant, ie it is an error to put spaces around the = sign.

To unset/clear a variable:

```
MYVAR=
```

12.1.2 Examining Environment Variables

You can list all your environment variables using the `env` command:

```
eechris@somepc:~$ env
$HOSTNAME=somepc.swan.ac.uk
$TERM=xterm
$SHELL=/bin/bash
$USER=eechris
$PAGER=/usr/bin/less
$PATH=/bin:/usr/bin:/usr/local/bin:/opt/bin:
$EDITOR=emacs -nw
$PS1=\u@\h:\w\$
$HOME=/home/eechris
$LS_OPTIONS=--color=auto
$LESS=-R
$LOGNAME=eechris
$DISPLAY=:0.0
```

(Tip: pipe it to `sort` or `grep` to find something particular.)

If you know the name of the variable you want to examine, use the `echo` command, prefixing the variable name with a `$`:

```
eechris@somepc:~$ echo $USER
eechris@somepc:~$ eechris
```

12.1.3 The `PATH` Environment Variable

One environment variable which we've seen before is `PATH`. This is a colon-separated list of directory paths, denoting the directories searched through by `bash` when you launch a program without specifying its full path.

For instance, if `PATH` is empty or unset, you can't run `cp` – if you try, then `bash` says '`bash: cp: command not found`'. A typical value for `$PATH` is:

```
/bin:/usr/bin:/usr/local/bin:/opt/bin:/usr/X11R6/bin
```

but they can grow much larger; very often when you install new software it requires you to extend your path.

One special case deserves mention: the current directory. If your `PATH` doesn't include an entry for `.`, then you can't run a program in the current directory without prefixing its name with `./` – you can fix this by adding `.` to your `PATH` in the following manner, which also illustrates how to extend your path in general:

```
eechris@somepc:~$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/opt/bin:/usr/X11R6/bin
eechris@somepc:~$ export PATH=$PATH:.
eechris@somepc:~$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/opt/bin:/usr/X11R6/bin:.
```

12.1.4 Setting The Prompt: `$PS1`

You can set the bash prompt in a number of exciting and useful ways using the environment variable `PS1` (and more rarely, `PS2`). There's an excellent HOWTO on this subject at:

```
http://www.tldp.org/HOWTO/Bash-Prompt-HOWTO/
```

The setting used for my prompt is shown in the `env` example, above. It displays my user name, the name of the machine I'm logged into, and the name of the current working directory.

12.2 Aliases

It's possible to set **aliases** for commonly used commands, in order to save typing. For instance, I often wanted a full listing of a directory as provided by `ls -laFi`, so I set up an alias called `ll` in the following manner:

```
eechris@somepc:~$ alias ll="ls -laFi"
```

This effectively gives me a new command to call, `ll`.

You can even over-ride existing names, which can be very handy if you always want to pass the same options. For example, the recommended way to compile C programs (at least on this course) involves passing the `-Wall -pedantic-errors -ansi -O` options to `gcc`, the GNU C compiler. In order to save tediously typing this each time, you can set up an alias for `gcc`, like so:

```
alias gcc="gcc -Wall -pedantic-errors -ansi -O"
```

You can list all the aliases currently set up by entering `alias` without any parameters. Unset aliases using the `unalias` command.

12.3 `.bashrc`

All of the above is useless if you have to re-enter your PATH, prompt, and aliases every time you start a shell. Gladly, you don't – your home directory should contain a file called `.bashrc` to which you can freely add your own code; it's executed every time you start a new `bash` session.

Note that when setting environment variables in `.bashrc` (or any shell script – see below), you should use the `export` command to ensure that the variable endures beyond the execution of the script. So, if you want a prompt like mine[23], you should add the following to your `.bashrc`:

```
export PS1="\u@\h:\w\$ "
```

`.bashrc` is a special case of a **shell script** – see below.

[23] This is actually the Ubuntu default!

12.4 Using `script` to Log a Session

You can use the `script` command to make a typescript/log of a shell session. See the `man` page for more details but the gist is:

- Start a script by calling `script filename`
- This will write the log into the file `filename`
- When you've finished, type `exit` or hit `Ctrl-D`
- You may see odd characters (like `^[[71;139R`) in the output – edit them out using a text editor before submitting any work produced using `script`.

12.5 Shell Programming

`bash` isn't just a shell for working in interactively; you can also write **shell scripts**, effectively small programs which can call all of the commands we've seen so far. Shell scripting is where the power of small programs working together really comes into its own. Shell scripts can contain all of the familiar constructs such as conditional statements, looping, functions, parameters, return values, etc.

Shell scripts are beyond the scope of this course, but if you're interested in finding out more, see the **Advanced Bash Scripting guide**:

`http://www.tldp.org/LDP/abs/html/`

Also, see section 14.2 for alternatives to shell scripting..

13 Networking Tools

13.1 Remote Logins

There are a number of ways to log in to remote Unix/Linux machines. Of these, the most important is `ssh`. A full exploration of these tools and their underlying mechanisms is beyond the scope of this course, but you should be aware of their existence and should endeavour to use `ssh`/`slogin` when performing remote logins.

13.2 telnet/rlogin

telnet and rlogin are insecure and **should not be used** by anybody who cares about having their password stolen[24]. Use ssh instead. We mention telnet and rlogin for historical reasons and because some foolish people still use them.

telnet allows you to connect to a shell session on a remote machine listening on the right port; you are prompted for a user name and password and you get a session just as if you were logged in locally. rlogin is a related protocol which allows remote logins without prompting for user name and password.

The main problem with telnet and rlogin is that everything is sent across the network 'in the clear' – even passwords. Eavesdropping on such communications is trivial with the right tools. Both of these protocols have been superseded by ssh, which should be used instead..

13.3 ssh/slogin

ssh, the **Secure Shell**, provides a mechanism for securely logging on to remote machines. This is achieved by encrypting all stages of the transmission – the data sent during the session **and** the user name/password data used to set the session up.

To log on to a remote machine using ssh:

```
eechris@somepc:~/eechris$ ssh -l eechris eehope
```

This sets up a negotiation between my local ssh client and the remote sshd server, and provided nothing is amiss, I should be prompted for my password and given a session.

The first time you ssh to a machine, you'll tend to see something like this:

```
eechris@somepc:~/eechris$ ssh eehope
The authenticity of host 'eehope (137.44.6.44)' can't be established.
RSA key fingerprint is dc:38:16:95:d7:f2:e3:9e:10:e1:e9:a9:8e:5b:dd:5e.
Are you sure you want to continue connecting (yes/no)?
```

You type in yes and it says:

```
Warning: Permanently added 'eehope' (RSA) to the list of known hosts.
```

What's happening here is that your local ssh client keeps a list of known hosts (in /.ssh/known_hosts), each of which has a fingerprint. The first time you see the machine, it asks if this new fingerprint is acceptable. The truly security conscious would,

[24] This means you.

before `ssh`'ing to a new machine, contact that machine's administrator and find out what the fingerprint should be. This rarely happens in practice, however.

`slogin` is very similar to `ssh`, but can only be used for logging in to a remote session, whereas `ssh` can also be used just to run a command on a remote machine without opening an interactive shell.

13.4 File Transfer

13.5 `ftp`

FTP is the classic **File Transfer Protocol**, one of the most popular methods (before HTTP) for moving files around networks and the Internet. FTP is a client-server protocol: you use an FTP client on your local machine to connect to an FTP server on a remote machine and download or upload files accordingly.

There are many, many FTP clients in existence, but on a Linux system you can expect to have a basic command-line version called, you guessed it, `ftp`. Its `man` page is pretty good, or here's a nice tutorial:

```
http://unix.about.com/library/weekly/aa121800a.htm
```

Note: like `telnet` and `rlogin`, the FTP protocol works entirely in clear text, so it's another good way to give your local hacker your passwords. You should strongly consider using `scp` instead.

13.6 `scp`

`scp`, or **Secure Copy**, is to `ftp` what `ssh` is to `telnet` – it provides basically the same service but everything is encrypted and thus more secure.

It's not exactly like `ftp`, however – in FTP you establish a session within which you issue commands to upload and download files; `scp` works more like standard Unix `cp`, but with the added components of remoteness and security.

Paths on remote machines are specified by preceding them with the hostname followed by a colon. For example:

```
scp /tmp/myfile.txt eechris@eehope:~/coursework/
```

which would copy the local file `/tmp/myfile.txt` to `~/coursework` on the remote machine `eehope`. `scp` is a lot like `cp` in that you can specify multiple files to copy, copy directories (using the `-r` option), etc.

13.7 Beyond passwords

One of the really nice features of the SSH suite of tools is that, like `rlogin`, they provide a mechanism for doing things without having to type in your password each time. Ordinarily when you `ssh` to a box, or use `scp`, or do anything SSH related, you'll be asked for your password, but using public/private keys and a *key chain*, you can arrange things so that you only need to type your password (actually now a *pass phrase* associated with your keys) once, after which everything 'just works'.

If you use SSH a lot, this is a highly recommended technique. An excellent explanation of how to achieve this is contained in three articles by Daniel Robbins, of Gentoo:

1. `http://www-106.ibm.com/developerworks/linux/library/l-keyc.html`
2. `http://www-106.ibm.com/developerworks/library/l-keyc2/`
3. `http://www-106.ibm.com/developerworks/linux/l-keyc3/`

13.8 Web Tools

13.8.1 `w3m`, `lynx` and `links`

`w3m`, `lynx` and `links` are **text-only web browsers** suitable for use in a terminal window. Obviously they don't display graphics but they can be useful in a tight spot. `lynx` is the oldest, the classic; `links` is a remake/update, and `w3m` is similar, and the only one apparently installed on the departmental Linux machines.

Start it up with something like `w3m http://www.google.com/`

13.8.2 `wget` and `curl`

`wget` allows you to non-interactively download files from the web.

```
wget http://www.swan.ac.uk/
```

would download that file and save it to `index.html` locally. `wget` allows you to specify HTTP headers, cookies, proxies, and all sorts of other useful bits and bobs. It's actually really handy if you need to interact with web sites from the command line or via scripts.

`curl` is like `wget` but it also supports a number of other protocols, eg GOPHER, LDAP, and FILE.

14 Power Tools

14.1 Text Editors

Plain text is clearly (and explicitly) very important in Unix and Linux, and we've seen a number of small tools for manipulating text. This brings us to **text editors** – applications dedicated to problem of modifying the contents text files.

There are many, many, many choices of text editor under Linux. If you're using a system which is running a desktop such as KDE or Gnome, you'll probably find at least one, and possibly several, friendly GUI-oriented text editors available.

However, it can be worth knowing about some of the 'classic' text editors which are also available in any self-respecting Unix system. These are less flashy, generally have no GUI as such, and are a lot older. There are two main reasons why these can be useful:

- Ubiquity – knowledge of how to use these classic editors is highly transferable to other systems, because you can nearly always count on them being present, unlike your favourite GUI-based editor. In particular, if you're logged in to a remote machine using `ssh`, you aren't going to be able to run it.
- On a related note, sometimes these non-GUI editors are all you've got. For instance, when Unix systems are used as servers in industry, it's common for them to have no GUI at all.
- Superiority – possibly the most convincing argument is that in many ways they're simply a better tool for the job. These programs have been around for a very long time, have seen a lot of development, tend to be highly stable, and often have many very powerful features.

Having said that, let's briefly examine three such editors:

- **nano**

  Of all the non-GUI editors, `nano` is the easiest to use (and, conversely, the least powerful). If you find yourself unable to find a friendly text editor, try it out. Its usage is obvious and 'intuitive'.

- **gedit**

  Is a simple GUI editor which works under GNOME. It is very easy to use. It's very similar to Windows notepad and is used for similar simple tasks, e.g. for tasks that you'd otherwise use `nano` for under Ubuntu.

- **vi**

  `vi` is one of the classic ubiquitous Unix editors – any self-respecting Unix hacker knows how to use it. Newcomers find it difficult because it works in 'modes': editing mode, command entry mode, etc. It's easy to get lost, confused, and frustrated, which is a shame because if you just learn the basics it's much more powerful than first appearances suggest.

  The best way to learn how to use vi is to launch it then type :help. You'll find a tutorial there that you can work through on-line. You will also find several tutorials on the Internet.

- **emacs**

  The other 'main' editor is `emacs`, considered by some (including the author) to be the best text editor available, with a mind-boggling array of third-party extensions for everything from writing email to playing Tetris.

  `emacs` is less confusing, initially, than `vi`, and there are GUI versions which make it even easier, but to fully reap the benefits you have to learn the emacs way of doing things, which involves arcane key combinations such as `Ctrl-X Ctrl-F` (to open a file) and being able to tell the difference between a buffer, a frame, and a window (whatever they are).

  Once you get there, though, it's superb. Three ways to get there:

  – Built-in tutorial.

    Accessed by launching emacs, then hitting `M-x` (that's either `Alt` and `x` at the same time, or `Esc` **then** `x`, depending on your keyboard), and typing in `help-with-tutorial` when prompted.

  – "Teach Yourself Emacs In 24 Hours":

    `http://home.no.net/skund/emacs/`

  – Lots of other tutorials on the net – google for 'emacs tutorial'.

14.2 Alternatives to Shell Programming

We saw in section 12.5 that Unix shells such as `bash` provide a very high-level way of programming tasks. Whilst shell scripts are undoubtedly powerful and saved a lot of needless C programming, they have certain limitations, such as:

- Typeless variables;
- Limitations to parameter lists (eg in `bash` no function can have more than 10 parameters);

- Slow execution;
- Poor error handling;
- Clunky syntax?

Somewhere between shell scripting and more traditional programming, then, we find a class of languages called **very high level languages**, which we mention now simply because they're cool. Absorb or ignore as you see fit. All of these are interpreted and aim to provide high-level constructs without making the programmer worry about low-level details such as whether memory has been allocated/freed or not. Their intent is to make programming easier, more productive, and more fun.

The two most important very high level languages at this time are:

- `Perl – http://www.perl.org/`

  Perl has been around since 1987 and is currently at around version 5.8.7. It is a very pragmatic language whose emphasis is on 'getting things done' and whose motto is 'There's More Than One Way To Do It'. Perl is very good at text processing, has excellent pattern matching and regular expression features, and is extremely popular amongst Unix system administrators.

  Perl sports a vast array of third party modules and a very powerful, mainly automated mechanism for dealing with them, called CPAN, the Comprehensive Perl Archive Network. If you want to solve some problem on your Unix box, chances are somebody has solved it already and released the solution.

  Critics of Perl point to its weak typing and the possibly inevitable corollary of the 'More Than One Way' philosophy: Perl code (especially but not exclusively other people's code) is very hard to read; because of this Perl has been referred to as a 'write once, read never' language.

- `Python – http://www.python.org/`

  Python has been around since 1991 and is currently at version 2.4.3. It is similar to Perl in a number of ways, not least its emphasis on getting things done, but goes about things very differently.The biggest difference is that Python tries to do things 'properly': rather than providing more than one way to do things and letting evolution take care of which one 'wins', there is a coherent and (hopefully) clean design at work.

  Python programs tend to be extremely readable, and Python positively encourages good documentation and well structured programs[25]. Its object-orientation and exception handling rival such trend-setters as C++ and Java, and it includes a number of idioms found mainly in functional languages (eg list comprehensions).

[25] Python is the recommended programming language for Ubuntu!

Others exist, most notably **Ruby**, which is similar in many ways to Python but takes a more rigidly object-oriented approach and is somewhat younger and thus less mature; it certainly has its proponents.