informationSERVICES

Introduction to Unix

Workbook

Edition 4, December 2016

Document Number:3523-2016

Contents

Chapter 1. Getting started

Chapter outline

In this chapter, we'll cover:

- a basic account of the Unix operating system
- a comparison of Unix with other operating systems
- the architecture of the Unix operating system
- accessing Unix, including passwords and security

Then, we'll move on to:

- the meaning of files in the Unix operating system
- seeing what files exist on your system
- manipulating files using Unix commands
- creating new files using a text editor

What is Unix?

Unix , Linux...what's the difference?

- both are operating systems, Unix is older
- Unix dates from the development of the internet
- Linux has Unix at its core
- both freely available in different distributions
- "*nix" systems used in wide variety of computers

Today, we're going to be accessing a remote machine running **Linux** as its operating system, however this course will use **Unix** and **Linux** interchangeably.

What is Unix?

- multi-user operating system
- acts as interface between user and computer
- allows programs to be run to deal with text and data...
- more complex tasks can be run, for example simulations
- access to resources such as printers and file space

Unix dates from 1969, when it was developed as a multi-user operating system at the AT & T laboratories in the USA. Linux is more modern, appearing in 1991 and stemming from the work of a Finnish student, Linus Torvalds. His Masters thesis was called "Linux: A Portable Operating System."

Both Unix and Linux underpin the operation of a variety of modern-day machines – from small handheld devices to desktops and also large, high performance clusters used in complex scientific computing environments.

- Comparison with Windows and Macintosh operating systems

- Microsoft's Windows has different underlying structure
- a graphical user interface is integral to Windows

  ...What about Apple Macs?

- Macs also have a rich graphical user interface
- Mac OSX has Unix "under the surface"

Today, we're going to access a machine and start learning the Unix operating system.

We'll be using **Unix commands** typed in using a command line interface.

- Why is Unix used? Why learn Unix?
  - widely used
  - good for security, highly configurable
  - can perform simple and highly complex tasks
  - you may require (or only have) access to a "*nix" machine

    ...so, you need to know the basics

It doesn't matter that we're probably doing this from a machine that might be running a different operating system.

- Additional features of Unix and Linux...
  - comes in various "flavours" (e.g. Fedora, Ubuntu, FreeBSD)
  - operating system is open source and free (mostly)
  - possible to try for a while ("live" CDs)
  - some machines come with Linux pre-installed

More on operating systems…

Unix is an operating system, in the same way that Microsoft Windows and Mac OSX are operating systems. Operating systems allow you to perform tasks such as running programs that allow you to work with documents and deal with data.

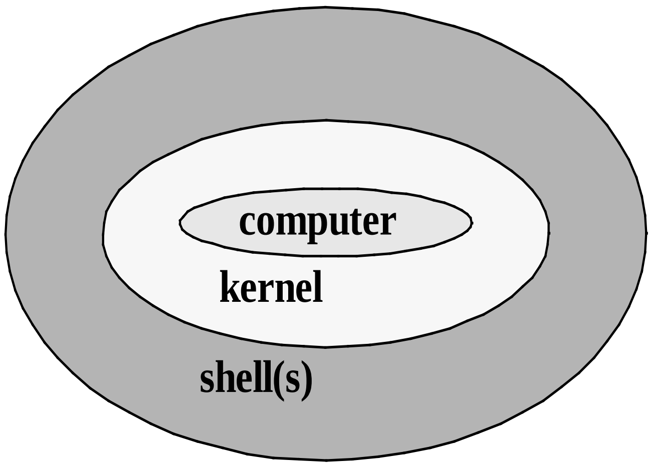
Linux is a "Unix-like" operating system – it was created using many of the features of the earlier Unix operating system. Modern-day Linux systems have many applications available for people to run.

Various "flavours" or "distros" (distributions) of the Unix/Linux operating systems exist – for example Red Hat Linux, Fedora and Ubuntu. Although open-source, some Unix systems are developed and targeted for commercial use (e.g. Red Hat Enterprise Linux). Others, such as Fedora and Ubuntu, are free and provided without dedicated support and technical backup from a manufacturer. Eddie3 uses Scientific Linux on Intel architecture.

It's beyond the scope of this course to cover in any detailed way the features and differences between the various flavours of Unix and Linux. You will be learning basic commands that are common to both.

How is Unix built?

Some terminology…



computer

the physical machine running Unix you're interacting with

kernel

the actual operating system - one large program, always resides in memory

shell

the part of the Unix operating system which interacts with the user

...different shells are available, providing different user interfaces

The Unix shell – also known as the command line – provides a user interface where commands can be typed.

Today we're going to interact with a Unix machine called Eddie3. It uses a shell called "bash".

Interacting with Unix machines

- Single computer – could be anywhere
  ...many people using it at the same time
- User name
  - identify yourself to the computer
  - potentially can bill you for use of resources on some systems
- Password
  - secret to yourself
  - don't leave it lying about in an obvious place
    (e.g. a Post-It note on your monitor!)

The Unix operating system enables a single computer to be used by many people at the same time. Each person is given a user name, and a password.

Your user name is your way of identifying yourself to the computer. Usually user names are based on the user's university user name (UUN).

You are also allocated space on the computer's storage disks and your account is your way of gaining access to this.

Before using the computer you must login using your user name and password combination.

You will be given a password when you are given your user name. This should only be known to you, and should be changed to a new password soon afterwards.

Logging in

- Before you can use the computer
  - quote your user name
  - quote your password
- Unix is case sensitive
  - capital letters are different from lower case letters

Choosing a password – advice

- First defence on your file space
  - change your password regularly
- Choose something nobody can guess
  - include non-alphanumeric characters
  - avoid dictionary words or proper names
  - avoid things like your date and place of birth
  - not the word "password"!!
- DO NOT FORGET YOUR PASSWORD
  - you will have to prove your identity to have it reset

Task 1.1 Logging on to Eddie3

Accessing a Unix machine using Secure Shell Client

➲ On a Windows machine, go to...

Start
In the **Search programs and files** box enter
mobaxterm and click **MobaXterm Personal Edition**

➲ Click the "**Session**" button at top left and then click "**SSH**"

login as: your University username
Password: Enter your EASE password

➲ Enter **Remote host:** eddie3.ecdf.ed.ac.uk and click "**OK**"

➲ **login as:** your University username
Password: Enter your EASE password

(On a Linux machine, look for a program called Xterm, terminal or console. On a Mac, go to Terminal in Application > Utilities).

Logging out of the system

ALWAYS log out before leaving the terminal

- otherwise other people can use your account credentials

Particularly important in public areas

- best not to leave account open even for a short time
- also a security risk

Task 1.2 Logging out

➲ Type:

exit

at the prompt to log off.

The window will now display logout.

...in order to regain access to Eddie3 you'll have to log in again, by quoting the username and password combination you used previously.

Starting to work with Unix: computer files

- Collections of information
    - text files
    - data files
    - configuration files
    - hidden files

- Stored in directories
    - automatically log in to your own home directory when you access eddie3

All computers store information in 'files'. These can be:

Text files

Files containing ordinary text information (such as logs, source code, etc.) are called ‘text files’ and their contents can be displayed in your terminal window. Text files can be created, amended and deleted according to your needs.

Data files

Data files contain information which can only be read by the program which created them, or by other programs which have been designed to handle their particular type of data. You, the user, may create data files during the course of your work with certain programs (e.g. with statistical packages). Data files associated with particular programs often have names similar to the program name.

Configuration files

Configuration files are usually text files containing information about things like fonts, colours, mouse behaviour, etc. There are often separate configuration files for different applications, and they are given names which link them to the appropriate applications.

File names

- Filenames can have special endings
  - text files commonly end in ‘.txt’
  - a script written in Perl would have the extension .pl
  - C program files end in ‘.c’
  - ‘.CPP’ is a C++ language file
  - ...and many more possibilities
- Names of hidden files start with a dot (“dot files”)
  - certain configuration files are often hidden

...there are occasions when you might want to access them though, so we’ll be learning how to view them later on in the course

How Unix views files

Unix command shells do not expect *or use* special endings to file names, they really see everything as a generic file.

Hidden files

In Unix, configuration files usually have names which start with a dot (.login, .profile, .bash_history etc), and their names may end in ‘rc’ (.bashrc, .inputrc, etc).

If a file name begins with a dot and ends with ‘rc’ this means it contains setup information for a package or program; its name will probably tell you which program it is associated with. The letters ‘rc’ stand for ‘run commands’.

Manipulating files – a first foray into Unix commands

- **ls**, ￼￼￼**mvmv**, **mv**, **rm**
    - ￼copy, copy, move, remove
- Unix is case sensitive – type with care
    - capital letters, lower case letters
- LS is different from ls

Unix provides several commands for manipulating files (showing you a list of your files, looking at their contents, changing their names etc). This workbook introduces the following commands:

ls, **more**, **cat**, **cp**, **mv**, **rm**

and includes practical instructions to help you learn about them.

Before looking at the commands mentioned above, it's worth covering a bit more material on Unix commands.

Unix is case sensitive

Unix treats the upper and lower case version of a letter as being different from each other. So, typing 'LS' is different from typing 'ls'.

If at any time a command does not work as you expected it to, check you've not used the wrong case – try using the same command in upper and lower case. Always press return after typing in a command – this is required for the command to be executed.

For example, typing in **ls** as a command works, whereas typing in **LS** and pressing return results in "command not found" as the Unix operating system has no concept of what "LS" means.

Unix commands

- Arguments
    - usually required
    - one or more filenames
    - e.g. **bertbert**

An *argument* following a command (in this example, the **ls** command) is simply another word, or phrase, that is typed in.

- Options
    - useful but not essential
    - several options can be used at the same time
    - options in Unix usually begin with a minus sign

- e.g. **ls –l** (here, the command **ls** has had the option **–l** added)

General information on Unix commands

In the simplest form, Unix commands are executed by simply typing in the command (e.g. **ls**) followed by the <return> key.

Arguments

Other commands need to be told where to find data, or need to be given the names of the files on which the command is to operate. This information is called the ‘argument’ to the command.

Options

Most commands can be made to operate in a different way by typing in ‘options’ as part of the command line. Options are usually single letters, and are prefixed by a minus sign ‘-’.

The only complication with options is that some commands allow several options to be specified after one minus sign (e.g. **ls -al**) while a few commands require each option to be given its own minus sign e.g. **ls –a -l**). If a command fails, and more than one option is included, try both ways of specifying the command!

Unix is flexible

- Shell interprets input
  - the shell processes what you type in to your terminal
  - the command is run
  - separates command, arguments and options
  - any output is displayed on your terminal window
  - the shell is now ready for you to enter the next command
- Using commands
  - command [list of options] [list of arguments]
  - for example **ls –l public_html**
  - most commands will accept either one argument, or several, and either one option or several

In the above example, a web designer wants to examine the contents of a file or folder (we’ll find out later how to tell the difference) called “public_html”. The folder in this case is called public_html. **ls** is the command and has **–l** as an *option*; the *argument* refers to public_html

The shell interprets input

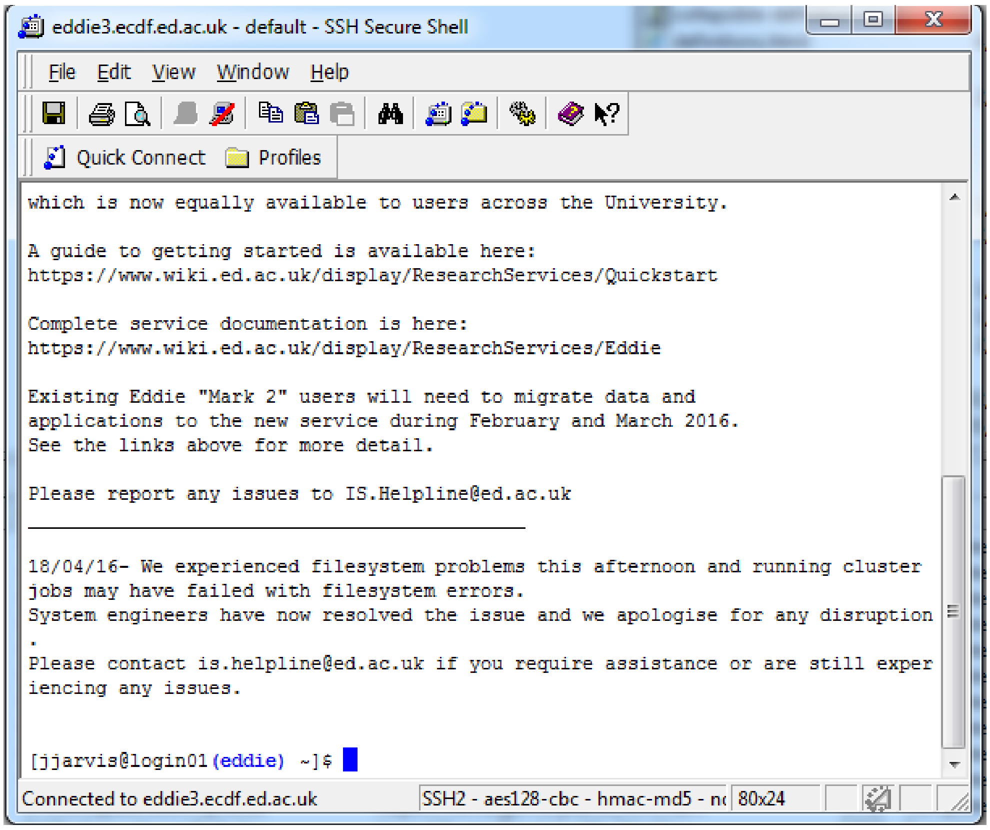
The shell interprets input, and runs the commands having sorted out which part of the input is the command, which is an argument or an option, and how many arguments and options are present.

Changing commands

Most commands will accept either one argument, or several, and either one option or several.

Shells versus Graphical User Interfaces ("GUIs")

- Shells
  - bash shell (although Unix has various possible shells)
  - typically a 80x24 terminal window to accept text commands
- Graphical User Interfaces (GUIs)
  - some Unix machines offer a GUI
  - but we're going to concentrate on learning commands
  - learning the commands allows direct communication with Unix
  - using commands allows greater flexibility once you've learned the basics


eddie3.ecdf.ed.ac.uk - default - SSH Secure Shell
File Edit View Window Help
Quick Connect
Profiles
Connected to eddie3.ecdf.ed.ac.uk
SSH2 - aes128-cbc - hmac-md5 - n
80x24

```
which is now equally available to users across the University.

A guide to getting started is available here:
https://www.wiki.ed.ac.uk/display/ResearchServices/Quickstart

Complete service documentation is here:
https://www.wiki.ed.ac.uk/display/ResearchServices/Eddie

Existing Eddie "Mark 2" users will need to migrate data and
applications to the new service during February and March 2016.
See the links above for more detail.

Please report any issues to IS.Helpline@ed.ac.uk

____________________________________________

18/04/16- We experienced filesystem problems this afternoon and running cluster
jobs may have failed with filesystem errors.
System engineers have now resolved the issue and we apologise for any disruption
.
Please contact is.helpline@ed.ac.uk if you require assistance or are still exper
iencing any issues.

[jjarvis@login01(eddie) ~]$
```

A typical interface – Eddie3 accessed via Secure Shell Client. Terminal windows on other machines (e.g. Macs) look very similar.

The Unix Shell/Command Line interface

The Unix *shell* - or *command line* – is the way we're going to use to interact with Unix on Eddie3 throughout this course. Our input in the form of text commands will be executed by the shell. (An analogous situation in Windows would be to bring up a command prompt by doing Start >Run>cmd to invoke the cmd.exe program).

Although in Unix any program can act as the user's shell, it's most commonly a command line interface the shell is associated with. We're going to use Secure Shell Client as a program to interact with the remote Unix machine called Eddie3. A benefit of Secure Shell Client is that it encrypts passwords, providing good security. (It's also freely available and can be installed on a home PC, details at the end of the book for those interested in this).

The terminology "shell" derives from the fact that we're not directly interacting with the deeper-down kernel level of the operating system. Instead we'll be interacting with the outermost shell, and commands are dealt with by Unix passing them down into the kernel.

Graphical User Interfaces

Some graphical user interfaces are available on Unix machines, for example GNOME and KDE. These are sometimes called *visual shells*. So, some machines running Unix have GUIs that look not too different from a normal Windows desktop, with icons to click and menu options.

Introducing Unix commands

ls (list files)

- List of files in the current directory
  - tells you which files are there
  - additional options available
- Most commonly used options
  - **ls -a**   also show 'dot' files
  - **ls -l**   show details about files
  - **ls -F**   show type of file
  - **ls –al**   is valid syntax for combining options

The command '**ls**' tells the computer to show you a list of the files in your current directory.

If the filename given is itself a directory, **ls** lists the files inside it. If 'filename' is the name of a file, **ls** repeats its name together with any other information requested.

If no filename is specified, files in the current directory are listed. If several filenames are given, they are sorted appropriately.

ls -l

You can ask for the list to give more information about each file by using the **-l** option. The details include file type, access permissions, number of links to the file, the file's owner, its size (given in bytes), the date it was last modified, and finally its name! (Permissions are covered later.)

ls -a

The **-a** option asks for a list of all the files in a directory, including files normally hidden from view. These files have names which begin with a dot (such as .profile) and contain configuration information.

ls -F

This option marks directories with a trailing slash, executable files with an asterisk, and symbolic links with a trailing at-sign (@).

Task 1.3 ls command

***Note:** Your tutor will instruct on setting up the course files. Please do this before starting this exercise.*

Using the ls command

At the command prompt, type in the command in upper case (LS) and then press <return>. What happens?

Now try **ls**, then <return>. A list of your home directory files appears.

Using the ls command options

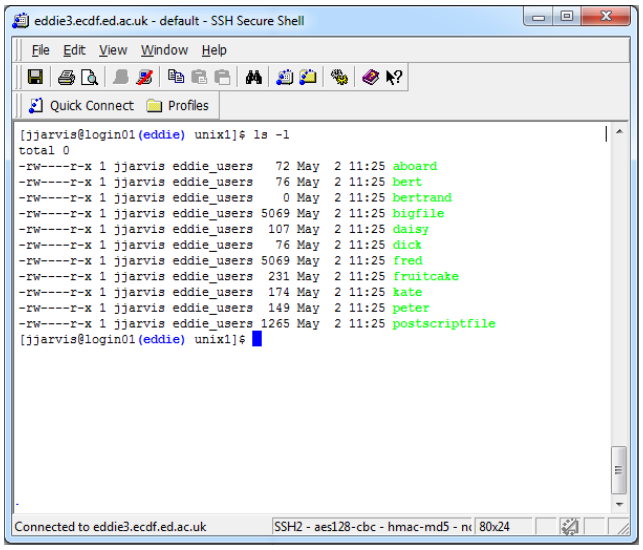
Now type **ls –l** (this is the letter l, not the number 1). You should now see more information about each file.

Now type **ls –a** - output includes normally hidden "dot files**.**"

Then try **ls –F** - using this option, directories are for example marked with a trailing slash (/), whereas executable files have a trailing asterisk (*) .

Also try **ls –alF** to combine the above.

If **ls** is followed by a filename (e.g. **ls fred bert**) the command repeats the filename(s), and gives any other information requested.


eddie3.ecdf.ed.ac.uk - default - SSH Secure Shell
File Edit View Window Help
Quick Connect
Profiles
Connected to eddie3.ecdf.ed.ac.uk
SSH2 - aes128-cbc - hmac-md5 - nc
80x24

```
[jjarvis@login01(eddie) unix1]$ ls -l
total 0
-rw----r-x 1 jjarvis eddie_users   72 May  2 11:25 aboard
-rw----r-x 1 jjarvis eddie_users   76 May  2 11:25 bert
-rw----r-x 1 jjarvis eddie_users    0 May  2 11:25 bertrand
-rw----r-x 1 jjarvis eddie_users 5069 May  2 11:25 bigfile
-rw----r-x 1 jjarvis eddie_users  107 May  2 11:25 daisy
-rw----r-x 1 jjarvis eddie_users   76 May  2 11:25 dick
-rw----r-x 1 jjarvis eddie_users 5069 May  2 11:25 fred
-rw----r-x 1 jjarvis eddie_users  231 May  2 11:25 fruitcake
-rw----r-x 1 jjarvis eddie_users  174 May  2 11:25 kate
-rw----r-x 1 jjarvis eddie_users  149 May  2 11:25 peter
-rw----r-x 1 jjarvis eddie_users 1265 May  2 11:25 postscriptfile
[jjarvis@login01(eddie) unix1]$
```

Typical output from the ls –l command. How to fully interpret the output will be covered later.

more (displaying file contents)

- more is a "pager" program
  - displays text, one page at a time

- more is useful when files are long
  - pressing spacebar permits moving down a page at a time
  - % progress down page displayed
  - command prompt reappears once viewing complete

The command more is a “pager” program. Using more permits you to view the contents of a file one screen at a time.

When a file’s contents exceed what can be shown in one screenful of output, the more command (e.g. more filename) allows you to view the contents of the file one part at a time by toggling through the document. The word “More” (and a percentage figure) appears at the bottom left hand side of the terminal window. Pressing the space bar once will show the next screen (a two line overlap exists for ease of use). Pressing return instead of using the space bar moves the document down one line at a time.

Once the process of moving down the document is complete the command prompt returns – a shorthand way of doing this is to press q at any time if you’ve finished looking at the file.

more options

SPACE	- display another screenful
RETURN	- display next line
Q (or q)	- exit from the more command
b	- move back one screenful

Task 1.4 more command

Using the more command

➲ At the command prompt, type in

more fred

to look at the contents of the file called fred.

The file fred is long enough to need more than one screenful of output.

With shorter files, the benefits of being able to toggle through a document using the space bar aren’t required:

➲ type

more daisy

as an example. All of its contents fit on one screenful of output.

more command options

Try out some of the more command’s options (down a screen, down one line, etc – see previous page).

cat (concatenate and display)

- ‘concatenate and display’
  - looking at a file’s contents
  - join a list of files
  - show the joined files on screen

- redirecting output
  - send output to a file rather than to the screen
    cat daisy kate > fred

The command **cat** stands for concatenate and display.

Uses of the **cat** command include looking at the contents of a single file and joining files together using redirection. In Unix-speak, *redirecting* means directing output not to the screen but to somewhere else instead (this will be covered again later on in this book).

Whereas **more** pauses so that the information within a file can be read, **cat** does not. If the file is longer than a page, the contents will fly past and all you’ll see will be the last screenful! Use **more** if you have to look at different parts of the file.

The **cat** command will display the contents of a single file if that’s all that’s specified – for example **cat peter** would display the contents of a file called peter.

More than one file: redirecting to another file using cat

The default behaviour of most Unix commands is to direct their results to the screen for you to view. However, it can sometimes be useful to send the output elsewhere, for example another file. Here,

cat daisy kate > fred

joins, or concatenates, the two files *daisy* and *kate* and puts them in a new file, called *fred*. If a file called *fred* already existed in t hat location, it would be overwritten. If no file called *fred* existed in that location, Unix would immediately create it (check for its existence by doing **ls –l**).

Appending instead of overwriting - one arrow is different from two

Using the symbols “ >>” means that information will be added to an existing file instead of overwriting it. So,

cat daisy kate >> fred

adds the contents of *daisy* and *kate* to *fred*.

Further cat options

Options that can be added to cat include:

cat –n	- adds line numbers

cat –vt - displays non-printing characters (not always visible)

Caution!

cat a b > a

and

cat a b > b

would both destroy the input files (a and b) before reading them.

In the earlier example (cat daisy kate > fred) the input files aren't destroyed.

also...

If you use the cat command with a binary file (e.g. myfile.zip) you'll probably end up with a lot meaningless output, as binary files need to be decoded. If this happens, exiting Eddie3 and then going back in is the best thing to do.

Task 1.5 cat command

Using the cat command

➲ Look at the contents of the files **daisy** and **kate** in your home directory:

more daisy

more kate

➲ Now use the cat command to join the files:

cat daisy kate

Redirecting output using the cat command

➲ Now join the files daisy and kate and put the results into a third file called fred:

cat daisy kate > fred

➲ Check the contents of fred:

cat fred and/or **more fred**

➲ Now append the file daisy to fred:

cat daisy >> fred

➲ After this, the contents of fred should have changed. Check this has happened:

cat fred and/or **more fred**

cp and mv (copying and renaming files)

- Copying files
  - the **cp** command creates a copy of a file

  - two versions of the file
- Moving/renaming files
  - the **mv** command moves a file to another location
  - only one version of the file

command allows you to make a copy of a file. For example, command allows you to make a copy of a file. For example, command allows you to make a copy of a file. For example,

cp oldfile newfile

would copy the file *oldfile* and create a new one - *newfile* - with the same contents.

Adding the –i flag prompts you before an existing file is overwritten - using **cp –i oldfile newfile** would mean the system would ask you whether you wanted any existing file called newfile to be overwritten. The –i stands for "interactive".

Similarly the **mv** command allows you to move a file from one place to another. The command

mv file1 file2

moves the current contents of *file1* into a new one, *file2*. As with **cp** above, **-i** can be added to give **mv –i file1 file2**

Only one copy of the file exists and in using the **mv** command the file has ultimately moved from one part of the filesystem to another, acquiring a new name on the way. We'll cover absolute and relative pathnames later, but in Unix-speak what has happened here is that the file's absolute path has changed.

The same syntax can be used to move entire directories – for example **mv workdir oldworkdir**.

Task 1.6 cp and mv commands

Using the cp command

➲ Create a new file called **buttercup** by copying the current contents of daisy:

```
cp daisy buttercup
```

Check the files are the same using an appropriate command. You could examine the contents of the two files using **more**.

Remember too that a variant of the **ls** command gives information on file size.

Using the mv command

➲ Rename the file called buttercup using the **mv** command. Change the file's name to bindweed:

```
mv buttercup bindweed
```

➲ Check that the move and renaming has worked by using the following sequence of commands:

```
ls bindweed
```

ls buttercup
more bindweed

rm (deleting files)

- **rm** stands for ‘remove’
  - a command to be wary of!!
  - adding **–i** option a good idea – allows extra confirmation step

command is for removing a file. Unix doesn’t have a Recycle Bin or a Trash area. So, removing files means really removing them! command is for removing a file. Unix doesn’t have a Recycle Bin or a Trash area. So, removing files means really removing them! command is for removing a file. Unix doesn’t have a Recycle Bin or a Trash area. So, removing files means really removing them!

First of all, do **ls –l** to verify you’re in the part of the file system you think you’re in. The command

rm byebye

would immediately delete the file *byebye*.

Using the command

rm –i byebye asks you to confirm before the file actually goes! Press **y** (*yes*) or **n** (*no*).

Now use **ls** to confirm that the file has gone.

Task 1.7 rm command

Using the rm command

➲ Look at the files in your home directory using **ls**. You should see a file called **bindweed**.

➲ Remove this file:

rm bindweed or (**rm –i bindweed** if you wish to add the interactive prompt)

Has the file gone? Use **ls** again to check.

Editing text in Unix: an introduction

- What are text editors?
  - allow modifications to text files
  - many different uses (e.g. writing scripts)

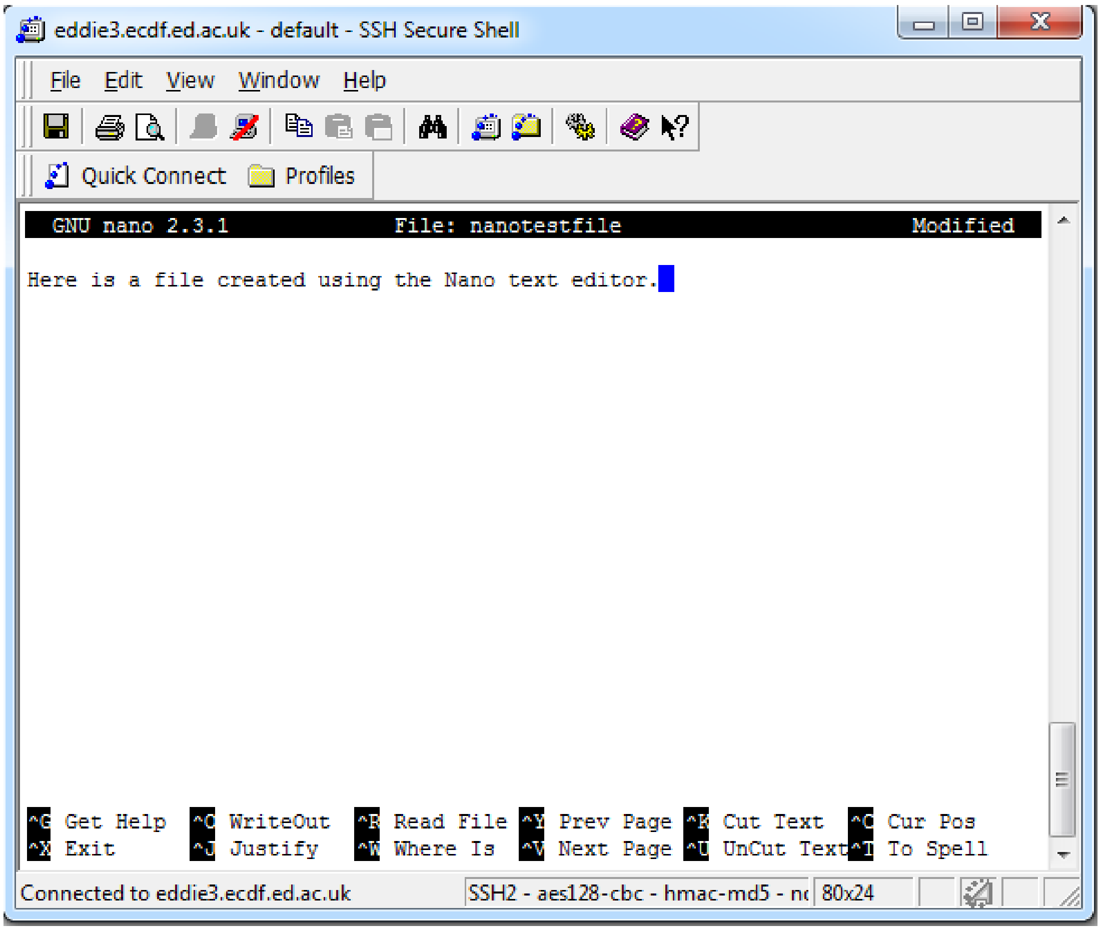
On UNIX systems, text editors are used for creating, modifying and saving text files (as are Notepad in Windows and TextEdit on a Mac).

So far in this course, we've looked at simple text files such as "daisy" and "kate". It's likely that in the future you'll want to create new files, and modify existing ones.

For example, you might want to alter an aspect of your UNIX setup such as your terminal type by modifying an environment variable (see Chapter 3). Another possibility is editing HTML files held on a UNIX server to update the details of a web page you have written. These tasks, and the creation of normal text files, can all be done using a text editor.

A number of text editors exist, and many UNIX systems have more than one. Choosing one depends on what's available, any past experience and how much time you would like to invest in learning a new editor from scratch.

For this course, we'll be using the text editor called **nano**. This text editor is menu-driven and, as a result, does not involve as steep a learning curve as other editors such as vi or microemacs (ue).


eddie3.ecdf.ed.ac.uk - default - SSH Secure Shell
File
Edit
View
Window
Help
Quick Connect
Profiles
Connected to eddie3.ecdf.ed.ac.uk
SSH2 - aes128-cbc - hmac-md5 - nc
80x24

```
  GNU nano 2.3.1              File: nanotestfile                     Modified

Here is a file created using the Nano text editor.

^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

A terminal window showing a text file that has been opened in the nano text editor.

nano – the basics

From the command prompt, typing

nano nanotestfile

does two things – the nano text editor opens up, and a file called (in this case) *nanotestfile* is created. (If there was already a file called *nanotestfile* it would be opened).

Instead of the command prompt reappearing, instead the cursor is positioned at the top of your window and you're ready to type your first file!

Note the menu options that show at the bottom of the window when nano is opened (see screenshot above).

nano – useful keystrokes

Saving

Once you are in to nano, you can just start typing. At the bottom of your screen you'll see menu items such as **^O**, and the **^** denotes that you need to use the **control key (ctrl)** in combination with a letter to execute the command**.** As it happens**, ^O** (pressing down the control key then the letter **O** in quick succession) is the command for **saving** a file. You should use it often!

In nano, the terminology "WriteOut" (see menu at bottom of the screen in nano) means saving.

When you do **^O**, simply pressing return completes the save. If you wish to rename the file, type in a new name and then press return.

Quick document navigation

←	move cursor back, delete
→	move cursor forward
↑	move cursor up
↓	move cursor down

Cutting and pasting

Go to start of the text you wish to cut. Then press **^^** (this is most easily done by pressing down the control and shift keys with one hand, and pressing **6** with the other). The words "Mark Set" appear to show that your selection has begun. Next, use the → key to choose how much text you wish to **cut**. Once you've done this, **^K** will remove your selection.

Finally, after moving to where you want the text to be **pasted**, do **^U** ("UnCut text") and it will appear. You can move to another place in the document and by selecting **^U** the same text can be pasted.

More keystrokes

^f	moves the cursor forward one character
^b	moves the cursor back one character
^a	moves the cursor to start of current line
^e	moves the cursor to the end of the current line
^n	moves the cursor to the next line
^p	moves cursor to previous line

^w	searches for text string (this isn't case-dependent)
^w^r	for search and replace
^t	starts spell-check

For more keystrokes, do **^g** when you're in nano to get some pointers.

Task 1.8 Using the nano text editor

Creating and modifying text documents

➲ Use **nano** to edit the file called *fruitcake* in your home directory.

Familiarise yourself with moving around the document using the arrow keys.

Experiment with some of the keystrokes given on the previous page – as before, they involve using the control key and a letter.

➲ Add a couple of new sentences. Change their placement in the document by cutting and pasting selected words.

In the final chapter of this workbook, you'll have further opportunities to use nano to modify files.

Summary

In this chapter, you've been introduced to the Unix operating system and have started to learn some Unix commands.

Other points worth considering include...

What are the advantages and disadvantages of having a remote machine running a multi-user operating system such as Unix?

Can you try accessing Eddie3 from an operating system other than the one you're using now? (For example, in your research lab?)

Commands covered so far:

ls	nano (to open the nano text editor)
more	
cat	
cp	
mv	exit
rm	

Next....Chapter 2

Chapter 2. The Unix file system

Chapter outline

In this chapter, we'll cover:

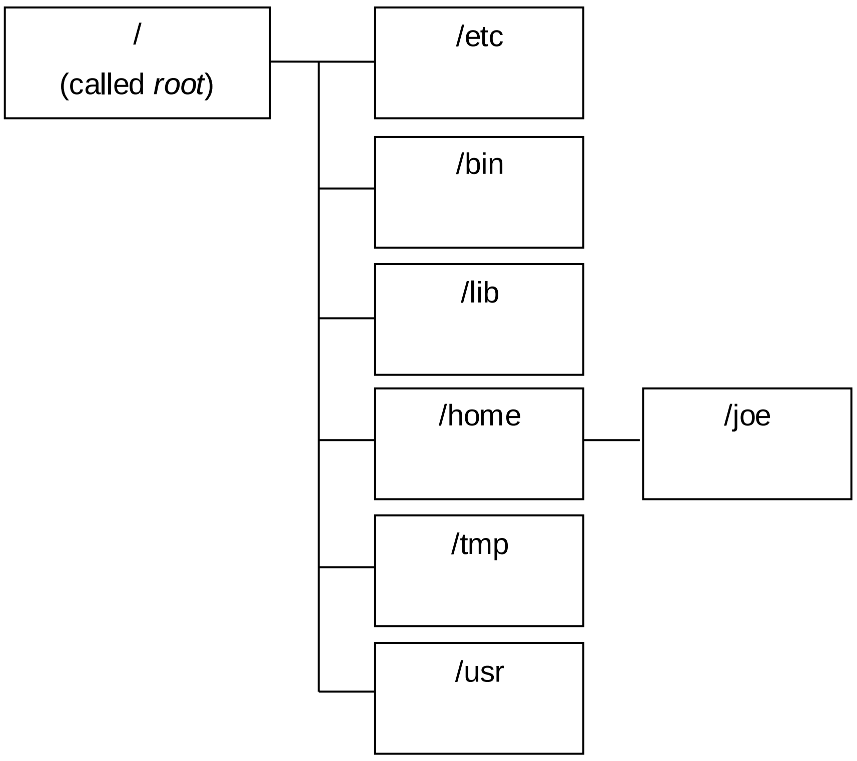
- the structure of the Unix file system
- navigating the Unix file system
- features of your home directory
- how to work out your location in the file system
- creating and deleting directories and files
- moving files around

Then, we'll move on to:

- file and directory permissions
- changing permissions

The Unix file system

- The organisation of Unix: the file system
  - lots of directories, organised structure
  - it's possible to look around without breaking things
  - the root directory lies at the top
  - all other directories and files are nested within the root directory



The Unix file system has a root directory at the top (also denoted /). Below that, nested within the root directory, is everything else.

Shown above are five of the most common Unix directories.

Others, such as /var and /sbin, are likely to be present too.

- Unix directories
  - root access is for system administrators, *not* normal users
  - normal user accounts have restricted rights (although users can still browse to areas they don't have rights to)
  - /home is the area of the file system we're most interested in – by default when you log on you are taken to your home directory
  - you can navigate elsewhere from your home directory
- "Moving up and down"

  You'll frequently be "moving up" and "moving down" directories.

  - "*moving up*" takes you closer to the root directory and into the directory containing the current directory
  - "*moving down*" takes you further away from the root directory, and involves moving into subdirectories contained by the directory you're currently in

  It is beyond the scope of this course to cover what is contained within all the common Unix directories. However, here's a short guide to what some of them contain -

/	etc	system configuration files
/	bin	the home of many essential commands
/	lib	many program libraries
/	tmp	temporary files
/	usr	where many local programs and system data files reside
/	var	system log files (can be used for troubleshooting faults)
/	sbin	files needed for booting the system

Where am I? – file navigation

pwd command

- The **pwd** command ("print working directory") tells you where you are in the file system
  - a relative measure...
  - if you move down into subdirectories, output of **pwd** will differ
  - the same applies if you move up, towards the root directory

  The **pwd** command is always useful if you are moving up and down the filesystem and forget where you are.

  By default, you are taken directly to your home directory when you log on to Eddie3.

The **pwd** command issued at this point will give the full path of your home directory.

If you then create a new directory called “work” (we’ll find out the command for creating a new directory soon) and move down into this directory from your home directory, the output of **pwd** will be as before but with /work at the end of the output.

For example:

pwd	could give	/home/fred	as output
ls – l	would then list the contents of /home/fred		

If a new directory (a subdirectory of fred) was then created, called betty, you could do

cd betty	to move into this directory.
pwd	at this point would give /home/fred/betty

Task 2.1 pwd command

Using the pwd command

➲ Display your current directory using **pwd**. Unless you have moved elsewhere in the file system after starting today, this will be the directory that was set up for the course.

The output, starting with a “/”, provides the full path down from the root directory (denoted “/” in Unix systems) to your present directory.

➲ Do

ls –l

to see which files you have there.

Later on, we’ll also be creating directories. When you have one or more directories in your home directory, the output of **pwd** will change if you move down into a new directory (or up, closer to root).

From any location, the command **cd** will always return you to your home directory.

Exceptions to the rule

On occasion it’s necessary for system administrators to shift files around, due to space restrictions or hardware upgrades for example.

This might include where in the file system users’ home directories are placed. Normally even if this does occur you’ll notice no functional difference – you’ll still automatically be taken to your home directory.

Sometimes though “full” (or absolute) pathnames for files will no longer work for files within users’ filespace. The way to resolve this is to use pathnames relative to the present position of each user’s home directory as these don’t change. Relative (and absolute) pathnames are covered soon.

More on file navigation

- Paths are assumed to start in the current directory
  - unless they start with "/" (/ denotes root level)
- Some useful shortcuts exist...
  - the current directory is called "**.**" (yes, just a dot!)
  - the directory above is called "**..**" (two dots)
  - **cd** *foldername* to change directory
  - **cd** alone always returns you to your home directory

Further pointers on file navigation

Specifying the full path to a file doesn't have to be in the form of an absolute file path (e.g. as you get as the output when doing **pwd**). Instead, file and/or directory locations can be specified with reference to the current location. This would be a relative path - for example /trnskl04/work instead of an absolute path such as /disk/home/scifac/trnskl04/work

The current directory is denoted "." and the one above is "..". So, **ls ..** lists the contents of the directory above your current one.

"**.**" and "**..**" can be used anywhere, the output depends of course on where you are relative to your home directory.

The notation **~** (the tilde sign) is sometimes seen – this also denotes home. So,

/home/bill

is the same as

~bill

cd ~ also returns you to your home directory.

Task 2.2 Further file navigation

Where am I now?

➲ Display your current directory using **pwd**.

The output is an absolute path to your current directory location. Note the text box below.

Moving around

➲ Create a new directory, called workdir, then move into it.

mkdir workdir (more on this command on next page)

cd workdir

to access the directory called workdir.

Doing a **pwd** will give you a different output now.

➲ It's also possible to look around elsewhere in the file system – try one of the system directories for example:

cd /usr/local

➲ Do **pwd** to check your location and then **ls –l** to see what is there. The output from ls –l from this location should show you lots of files owned by "root" and "other". You can see that these files exist, but can't change them (more on file permissions soon).

➲ **cd** here (or from anywhere) will bring you back to your home directory.

mkdir command

Course Files Directory

When you set up the course files earlier the command automatically changed your directory location from your home directory to the directory with your course files.

Important Note: Now you will need to use the **cd** command to return to the course files directory for future exercises. The command will typically be:

cd ~/IS_Courses/unix1/

- New directories can be created below present directory
  - **mkdir** command
  - allows better organisation of your own files

Task 2.3 mkdir command

Making new directories

➲ From your course files directory, create another new directory with any name you wish, e.g.

mkdir datafolder

➲ Do an **ls**. Is the new folder listed?

➲ Use **cd** to move down into this folder ("down" because it's a subdirectory of your home directory), e.g.

cd datafolder

An aside...

An **ls –al** on a recently created directory, with no files in it yet, actually reveals two files when you do an **ls –al**. The entry for "." signifies the directory itself, and ".." the link between the current directory and the one above it. The command **ls –l** will however give an output of total = 0 until a file is actually saved there.

More on commands and pathnames

Course Files Directory

When you set up the course files earlier the command automatically changed your directory location from your home directory to the directory with your course files.

Important Note: Now you will need to use the **cd** command to return to the course files directory for future exercises. The command will typically be:

```
cd ~/IS_Courses/unix1/
```

- What if no path is specified when issuing a command?
    - system assumes commands (**ls**, **cp**, **mv**...) refer to files in the present directory
    - current directory is "."
    - many commands can use *absolute* or *relative* pathnames

Examples...

Copying

cp file1 file2

contents of "file1" copied to another one "file2", all in same directory

cp file1 datafiles

file "file1" is being copied to the "datafiles" directory

Moving

mv file1 file2

contents of "file1" moved to another one - "file2" - all in the same directory

mv file1 datafiles/expt1

"file1" is being moved to the "expt1" directory, itself a subdirectory of "datafiles"

Important point:

Remember that **cp** will overwrite an existing file without warning (possible to get round this by doing **cp –i)**. **mv** is similar, except it involves copying the file and deleting the original.

Task 2.4 More navigation

Copying files

➲ Copy the file “daisy” from your course files directory into a new directory called “alfred”:

mkdir alfred

ls –l (check that the directory alfred now exists)

cp daisy ./alfred (moving file daisy to new directory, alfred)

In the command above, the dot followed by the slash denotes the current directory and the name of the new location (the directory alfred).

➲ **cd alfred** (move into the new directory)

ls-l

more daisy (to examine contents of daisy copy in alfred directory, same as original daisy file)

➲ The **touch** command can also be used to create a new (and empty) file in the new directory:

touch flintstone

Moving a new file

➲ To copy the new file “flintstone” back up into your course files directory, do

cp flintstone ..

then

cd .. (to move back up to your course files directory)

Shared areas and files owned by other people

- Possible with appropriate rights to see information about, and perhaps the contents of, files owned or created by other people
  - for example, in shared areas used by the same group
  - files that the system administrators want to give users access to, etc

- What about the security implications?
  - usually it's not possible to view the contents of files not owned by you
  - not possible to change system files!

...if you get lost, **cd** will always bring you back home

Unix is a multi-user operating system, and many users can keep their files on the same machine. Logging on to the machine with a unique username, and your password, takes you into your home directory; and it's here you will normally be saving files.

Looking at other user's home directories

In theory if you know another user's username, and if they have set the rights to allow it, you can use the **cd ~username** command to move into their home directory. This was sometimes used in the past and you may see references to it in other documents but it is not common nor recommended on modern systems to change the permissions on your home directory. Use shared areas instead.

Moving up the file system

Checking the contents of system directories can be useful on occasion; for example, you may wish to see if a particular package is installed. Alternatively, you might want to look at some configuration ("dot") files.

Security considerations

It's important to ensure that, even if others can view files owned by you, they can't actually change their contents (unless you want to grant this permission to them). System administrators ensure that system files are locked down and can't be modified by normal users.

Task 2.5 Looking at a shared area

Where am I now?

➲ Check your current location using **pwd**. If you're not in your home directory, go there using **cd**.

Moving into the shared area

➲ The area is in a special "scratch" area of the server; scratch is designed for temporary usage or for "ad-hoc" sharing of data

cd /exports/eddie/scratch/unix1

➲ Check where you are using **pwd**, then **ls** to see what's there.

What can you see? You may be able more information about those files, for example who actually owns them, by doing **ls –l** – take note of one filename for use later.

Soon, we'll see how Unix file and directory permissions become important when dealing with files that do (and do not) belong to us.

Moving higher up

➲ Move to the root directory by doing **cd /**

➲ Have a look at root's files using **ls -l**

➲ Try looking around – for example **cd /etc**, followed by **ls –l**

➲ **cd** back home when you're finished.

Shared resources: groups

- Possible to access shared resources
  - shared server group space (e.g. lab group data files)
  - communal printers
  - under Unix, each user belongs to at least one group

Working collaboratively

A number of users may have accounts on a Unix system, each one using a unique username and password combination to access their files held in their own home directory.

A shared group space might however be used for a number of people to access the same set of data files, such as experimental results within a lab group. Creating an appropriate group and granting access to those entitled to it is really the only recommended way to share data with multiple people

Other resources

Groups can be created to control access to shared hardware, such as printers (although this is uncommon).

Task 2.6 Groups

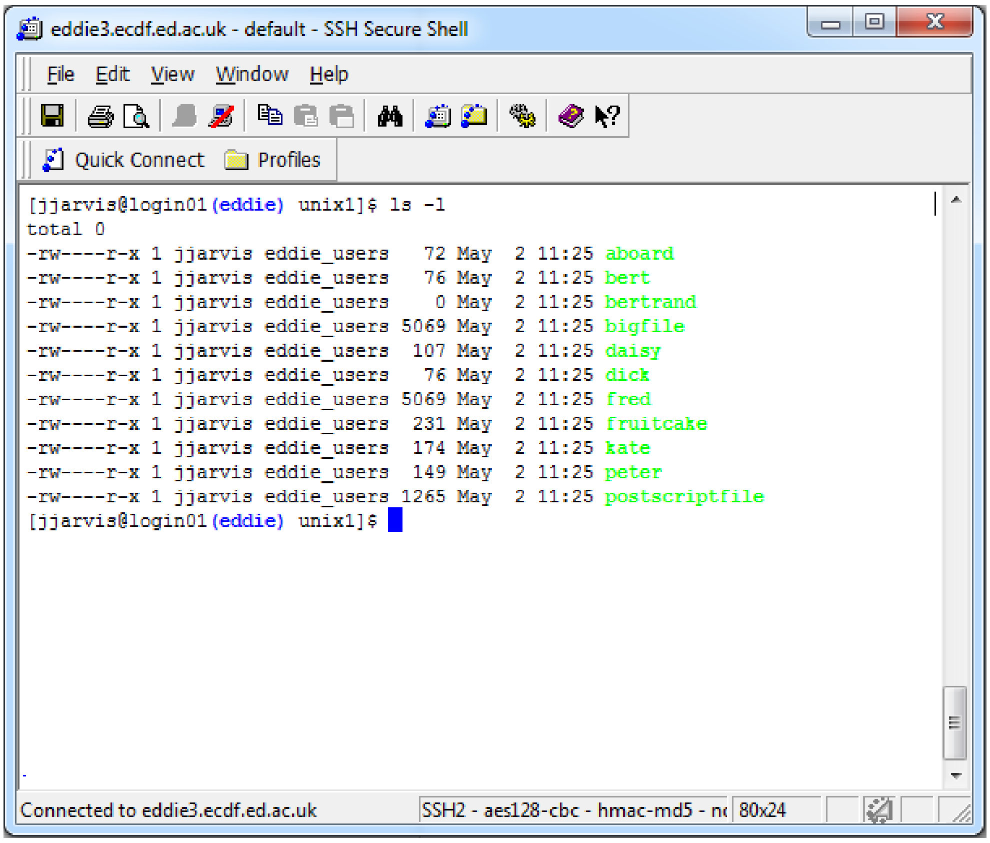
Which group(s) am I in?

➲ Check your group membership simply by using the **groups** command.

  o Note that this may take some time and produces a lot of groups, many of which will look unfamiliar – this is due to integration with other University systems.

Deeper Unix: information on files

- File ownership and permissions
  - check the output of **ls -l**
  - look at the ownership of the files and directories
  - look at the permissions of the files and directories


eddie3.ecdf.ed.ac.uk - default - SSH Secure Shell
File Edit View Window Help
Quick Connect
Profiles
Connected to eddie3.ecdf.ed.ac.uk
SSH2 - aes128-cbc - hmac-md5 - n
80x24

```
[jjarvis@login01(eddie) unix1]$ ls -l
total 0
-rw----r-x 1 jjarvis eddie_users   72 May  2 11:25 aboard
-rw----r-x 1 jjarvis eddie_users   76 May  2 11:25 bert
-rw----r-x 1 jjarvis eddie_users    0 May  2 11:25 bertrand
-rw----r-x 1 jjarvis eddie_users 5069 May  2 11:25 bigfile
-rw----r-x 1 jjarvis eddie_users  107 May  2 11:25 daisy
-rw----r-x 1 jjarvis eddie_users   76 May  2 11:25 dick
-rw----r-x 1 jjarvis eddie_users 5069 May  2 11:25 fred
-rw----r-x 1 jjarvis eddie_users  231 May  2 11:25 fruitcake
-rw----r-x 1 jjarvis eddie_users  174 May  2 11:25 kate
-rw----r-x 1 jjarvis eddie_users  149 May  2 11:25 peter
-rw----r-x 1 jjarvis eddie_users 1265 May  2 11:25 postscriptfile
[jjarvis@login01(eddie) unix1]$
```

The output of the command **ls –l** on a user's home directory. In the left-hand column "d" denotes a directory, and "-" denotes a file. The letters r, w and x relate to the permissions on the file or directory. The other columns relate to the user's login ID, group membership, information regarding when the file was created and, in the last column, the name of the file or directory.

Unix file ownership

- Three levels of Unix file ownership exist, relating to...
  - user (denoted "u")
  - group (denoted "g")
  - other (or "rest of world", "o")

File ownership and permissions

When doing **ls –l** in your home directory (or any other directory) you get as part of the output information on file ownership and permissions.

Understanding file permissions is important – if you can't access or modify a file, it's probably because of a permissions issue.

User (u)

This refers to your identity as the currently logged-in user (e.g. trnskl08). Any files created by a user are automatically owned by that user.

Group (g)

All files (and directories) are also members of a group (remember that in Unix every user has to belong to at least one group).

The owner of a file cannot change the group ownership of that file to that of another group (although a system administrator could).

Other (o)

Refers to anyone else, namely other users with no association to the file or the group the file is in.

An example of where this level of permission is important is mounting web pages on a Unix server, or in writing programs for others to use. Without granting access to “other”, the intended audience wouldn’t be able to see the web page or run the program!

Unix file permissions

- Three levels of Unix file permission exist
  - read (denoted “r”)
  - write (denoted “w”)
  - execute (denoted “x”)

...and it’s possible to modify each of these.

File permissions apply to each of the three levels of file ownership just covered (user, group, other).

Read (r)

This grants permission to read the contents of the file or directory.

Read permission is denoted by the letter “r”.

Write (w)

This grants permission to write to a file or directory (in other words, to amend it).

Write permission is denoted by the letter “w”.

Execute (x)

This grants permission to execute a file (for example a shell script you wish to run).

Execute permission is denoted by the letter “x”.

Some further points to note

File permissions and directory permissions are different, and both must be considered.

The table below shows some possibilities for directory and file permission.

File permission level	**Permission on directory containing file**
	x
none	can't do anything with file and directory
r	file can be read or copied
w	file contents modifiable
x	program in file can be run
	r + x
none	**ls** can be used on all files
r	file can be read and copied
w	file contents modifiable
x	program in file can be run
	w + x
none	files in directory can be deleted or renamed
r	file can be read or copied
w	file contents modifiable
x	program in file can be run

Note that...

- -execute permission to a directory allows the user to access files in the directory (needed for access to subdirectories too)
- -read permission to a directory allows any user (u, g, o) to list the files in the directory (read permission always needed at the file level too)
- -directories are useless without execute permission!

Task 2.7 File ownership and permissions

What are the permissions of the files in my home directory?

➲ Check the permissions of the files in your own home directory...

ls -l

➲ Now try those in the root directory

cd / (check you're there using **pwd** if you wish)

ls –l (you should see a lot of files, see the different permissions)

➲ Now look at some less frequently seen system files...

cd /bin

ls –l

➲ Now head back home

cd

What are my directory permissions?

➲ Look at your home directory’s permissions, and those of other directories.

cd followed by **pwd**	**-** your home directory
cd ..	- to change to the directory above your home directory
ls –l	- see what’s there
cd ..	- move up yet another level
ls –l	- see what files are there
cd	**-** head back home

Changing file permissions

- File permissions can be changed
  - for the files and directories that you own
  - the **chmod** command is used
- Various permission levels can be specified
  - for user (**u**), group (**g**), other (**o**), default is all (**a**)
  - can add (**+**), remove (**-**), assign (**=**)
  - possible to have read (**r**), write (**w**), execute (**x)** access

The **chmod** command is derived from changing the access mode of a file. Permissions are for **u**, **g**, **o** and **a** (**a** = all, which is essentially an abbreviation for **ugo**).

Adding and subtracting permission

Permission is added using the plus sign (**+**), removed using the minus sign (**-**), or assigned explicitly using the equals sign (**=**).

Removing all access

All access to a file can be removed (so it can’t be read, written to or executed) by using the minus sign on its own.

Command syntax

It’s important that no spaces exist between the user type, the equals sign and the settings when using the **chmod** command. If there are, the command won’t work!

If you want to use **chmod** to change permissions for multiple files, do so by separating the file names with commas (and no spaces).

Some examples of the chmod command

Imagine you have the following output from **ls**:

```
-     rwxr-xr-x       donald   science    home.sh
-     rw-r—r--        donald   science    feedback.html
-     rw-r-----       donald   science    imasecret
drwxr-xr-x            donald   science    stories
```

The above output shows one directory ("stories", denoted "d") and three files.

At the moment, the file "home.sh" can be executed by anyone. To prevent outsiders from doing so, the command

chmod o=r home.sh (making it –rwxr-xr-- now)

stops "home.sh" being executable by anyone.

Donald's web page "feedback.html" is currently only readable by Donald, the owner. To make it readable and writeable by anyone, the command

chmod og=rw would work (making it –rw-rw-rw- now).

To remove all group access to the file imasecret,

chmod g= imasecret would work (making it –rw-------).

To make the "stories" directory have write access for the group, do
chmod g+w stories (now it's drwxrwxr-x).

Eddie3 and DataStore File Permissions

Although we are teaching you about **chmod**, be aware that on many University systems the underlying permissions are being set in a more complex way.

Please do not change permissions using **chmod** *except* in the course files directory.

Task 2.8 chmod - changing permissions

Changing some permissions

➲ Check your own permissions again using **ls –l** in your course files directory.

➲ The file *fred* lacks write permission for the group, add it by using

chmod g+w fred

➲ Do **ls –l** again, have the permissions changed?

Removing some permissions

➲ Look at the permissions for the file *daisy*. Remove all its permissions (even for you) by doing

chmod = daisy

➲ Again, verify the new permissions using **ls –l**. (After this, you'd have to add your own permissions back to the file to do anything with it.)

Granting full permission

➲ The file *kate* can have full permissions granted to it – do

chmod a+rwx kate

ls -l

Restricting a file to yourself

➲ The file *kate* can be read by others. To make it only readable by you, remove all permissions using the = option on its own

chmod = kate

followed by adding the desired permission

chmod u=r kate

Check the results using **ls –l**

Copying files from elsewhere

- Files can be copied from directories other than your own.
  - read access to the file is required
  - the copy of the file then belongs to you
  - execute permission is needed on all directories from root level down, up to and including the directory containing the file to be copied

It is possible to copy files owned by someone else that you have been given permissions to see. As long as you have read rights to the file (either as a member of a group or as a user of the system if permissions have been granted to "other") and execute permission to both the directory it is in and all directories above it in the path to the file then you can

The other person has to grant access permission to people in specific groups, or to everyone on the system.

Note: If you are using a shared area set up for a specific group of which you are a member you usually do not need to think about this; the permissions should be correct for your group by default. Indeed, you should normally not be changing permissions with chmod as this can cause unintended effects due to integration with other systems.

Further points to note

The **cp** command doesn't produce any output (unless there is a mistake in the syntax of your command).

Also, be careful as **cp** will readily overwrite an existing file without any warning! Using **cp –i** gets round this, as it adds the interactive step of asking you if you are sure you want to overwrite any already existing file.

When copying files, relative or absolute paths can be used. Using absolute paths (such as /home/bob/work) has the advantage over using relative paths (such as bob/work) in that directories and files can be manipulated from *anywhere* in the file system, as with relative paths files can only be manipulated with reference to your *current* location.

Task 2.9 cp – copying files

Using cp to copy a file owned by someone else

➲ Remember the shared scratch area used earlier? We're going to copy a file straight from that shared area. You should be in your IS_Courses/unix1 area for this – **cd ~/IS_Courses/unix1**

➲ To actually copy a file to which you do have access, use the command

cp /exports/eddie/scratch/theirfile myfile

where;

/	theirfile the name of the to-be-copied file (located in the shared area and that you noted the name of earlier)
myfile	a name for the copied file (now yours, and you can call it anything)

➲ Check the ownership of the copied file...

ls –l myfile

Removing files and directories

- Files can be removed
  - use the **rm** command (write permission needed)
  - adding the **–i** option to give **rm –i** gives a safety net...
- Directories can be removed (need write permission)
  - remove all files from the directory first
  - then use the **rmdir** command
- Be careful when using **rm !**

- it's possible, in one step, to delete a directory including all of its files and the contents of any of its subdirectories
- adding **–i** option gives a safety net once again

Removing files from directories

To remove a file from your current directory (after doing **ls –l** to check its contents), do

rm –i byebyefile	adding **–i** ensures you're asked whether you're really sure you wish to delete the file, using **rm** followed by a space and the filename is fine, but won't offer a second chance!

Multiple files can be deleted by doing e.g. **rm –i file1 file2 file3**

Removing directories

To remove a directory, remove the files in it first. Then

rmdir –i byebye	is the safest way to delete the directory, as again you'll be prompted to see if you are sure.
rm –r byebye	deletes the directory "byebye" and ALL of its contents, including any subdirectories – be careful!

Task 2.10 rm and rmdir - removing files and directories

Using rmdir to remove a directory

➲ Check your current location –

pwd

(If not in your course files directory, please follow instruction in box below).

ls –l

to then list the files you have there.

➲ Remove the directory called "alfred". First of all, remove any files contained in the directory. The command

rmdir alfred

will then remove the directory.

Other options

➲ If you're absolutely sure you wish to delete the directory and everything in it, in one step, use

rm –r alfred

Summary

In this chapter, you've been introduced to the directory and file structure underlying the Unix operating system. You've also covered navigation up and down the file system, and the creation of new directories and files in your own home directory. We've also looked at directory and file permissions, and how to change them.

Another point worth considering...

Somebody has given you a shell script to run to automate a tedious task you frequently have to perform on some data from an experiment. (A shell script is a list of commands saved in a file, that file can then be run when you need to perform the task, saving you from issuing many separate commands.)

You try to run the shell script file and it won't work. What would you do to make the shell script work?

Bear in mind that, by default, Unix doesn't directly execute a file as a command (and files that you make are similarly created with only read and write permission for you).

What security implications do these features of Unix have?

Course Files Directory

When you set up the course files earlier the command automatically changed your directory location from your home directory to the directory with your course files.

Important Note: Now you will need to use the **cd** command to return to the course files directory for future exercises. The command will typically be:

```
cd ~/IS_Courses/unix1/
```

New commands covered:

pwd	chmod	groups
cd	rm	rmdir
mkdir		

Next....chapter 3

Chapter 3. The Unix bash shell and command line editing

Chapter outline

In this chapter, we'll cover:

- using some time-saving features of the Unix bash shell
- reviewing previous commands using the history command
- command-line editing
- extracting information by using wildcards

Then, we'll move on to:

- environment variables
- looking at some Unix configuration files

Saving time using the shell

- Typing shortcuts can be used
    - to complete file names without typing the full name
    - to review previously issued commands
    - to edit the command line
    - ...and in more complicated command syntax (pipes etc.)

Using typing shortcuts

The bash shell has various shortcuts available, and knowing some can save you time when you're issuing Unix commands.

One of the most useful is the ability to complete file names without having to type out the full file name (also known as filename completion or auto-completion).

The bash shell also allows you to review previously-issued commands (useful when you wish to avoid typing out a long command again (and when you can't remember the correct syntax!).

It's also possible to edit the command line so that instead of making a mistake and hitting return (leading to an error message or, worse, doing something you didn't intend to do to a file) you can simply edit what you have already typed and correct the mistake before executing the command.

Filename completion

- Type the first few characters of a file's name
    - then get the system to do the rest of the work for you by pressing <tab>

Filename completion works by you typing in the first few characters of a file name, and then pressing the <tab> key on your keyboard. The more characters you have typed in, the narrower the search becomes.

If after pressing the tab key once nothing happens, press it again. The system will then return the names of all the files with names beginning with that combination of characters (if any such files exist). This can be useful if you can't quite remember the exact name of a file you're looking for (we'll be looking at other mechanisms for searching later.)

Task 3.1 Filename completion

Using filename completion

➲ Use **ls** to look at the names of the files in your course files directory.

➲ Choose a file, then type in the first few characters followed by the <tab> key. For example,

ls –l bert <tab> would find the file called "bertrand".

Unix history mechanism

- Possible to review previously issued commands
  - saves you typing them again
  - easy navigation using arrow keys (up and down)
- Keystrokes available to save time
  - control key (^) in combination with others
  - moving backwards and forwards

Using previously issued commands

The bash shell allows previously issued commands to be reviewed, and used again without having to be retyped. This can save time, and is also useful when the syntax of a command used before has been forgotten!

Some useful keystrokes can be used:

^p	- show the previous command used
	...and **^p** again shows the next previous command
^n	- after doing **^p**, this moves you forward one command
^u	- completely clears the command line

The history mechanism

It's also possible to list the commands you've used before simply by issuing the command

history

The history file typically lives at .bash_history (which should be visible when you do **ls –al** in your home directory).

Task 3.2 Using previously issued commands

Looking at commands issued previously

➲ Go through some previously issued commands using the arrow keys to move up and down

➲ Also try **^p**, **^n** and **^u**

Using the history mechanism

➲ Use the **history** command

The information in history is usually located in a file in your home directory called

.bash_history

➲ You can, if you are interested, look at its contents using **more**

Editing the command line

- The command line can be edited
  - another time saving feature
  - for example, to change the named file in a command
  - the command will execute from anywhere

Editing the command line

When working back through previous commands using the arrow keys, or other shortcuts, you may need to alter a small part of the command's syntax, such as the name of the file. Consequently, editing the command line is a useful Unix feature as it means that you can alter mistakes made when typing in commands if or when you spot them. Alternatively, having the ability to edit the command line prevents error messages being returned to you after executing a command with incorrect syntax.

Once you have amended the syntax of your command simply press return to execute it. The command will execute irrespective of the position of the cursor in the command line.

Useful keystrokes

Keystrokes can be used to edit the command(s) you type in, including:

^b	- move back one character (left arrow key may also work)
^f	- move forward one character (right arrow may also work)

^a	- move to start of line
^e	- move to end of line
^u	- clear the line completely

Task 3.3 Editing the command line

Modifying the contents of the command line

The last practical emphasised the time-saving feature of being able to locate previously issued commands. Often, however, the exact same command is not the one you will want to execute.

For example, you may need to change the name of the file to be modified.

➲ Familiarise yourself with some of the useful keystrokes outlined in the preceding section.

The keystroke **^u** will clear the line, so you can start again.

Note that the command you type in will execute regardless of where the cursor lies.

Searching in Unix: wildcards

- Wildcards are useful searching tools in Unix
  - asterisk (*****)
  - question mark (**?**)

Wildcards in Unix

Using wildcards can be useful when you have incomplete information – for example, to find files with similar names or in cases where you can’t quite remember their names.

Wildcards can be used in a number of situations, and with many commands, but are most commonly put to use when searching for particular files or directories.

The wildcard symbol can be placed at any point in a name; so, for example, fr*d and *red are both valid terms.

The asterisk

The asterisk symbol (*) can be used as a wildcard and it substitutes for any number of characters. So,

ls –l pic*

would return information on files called “picture”, “pickup” and “pickpocket” in a directory. (Also any file called “pic”, as the asterisk wildcard substitutes for zero or more characters).

The question mark

A question mark (?) substitutes for one character. For example, the command

ls fre?

would list details of files called “fred” and “free” in a directory, but not “freddy”.

Task 3.4 Wildcards

Using wildcards to search – the asterisk

➲ Look for all the files in your course files directory beginning with the letter “p”

ls p*

➲ and a list of the files ending with the three letters “and”

ls *and

Using wildcards to search – the question mark

➲ Which files in your course files directory have four letters, the last of which is “d”?

The command

ls ???d

should tell you.

➲ Which files contain the letter “b” as the second letter?

ls ?b*

Redirecting information

- Standard input and standard output
  - by default, *standard input* is what is typed in as a command at the terminal
  - *standard output*, the result of a command, is usually sent back to the terminal (the screen)
- It’s possible to redirect output
  - using “**>**” will overwrite an existing file
  - using “**>>**” will append information to an existing file
- What about when things go wrong?
  - *standard error* is where many commands will write messages if errors are encountered

Standard input and standard output

In Unix-speak, commands are normally read from what you type in to the terminal (standard input). You type a command, press return, and by default the standard output - the result of the command you entered - goes back to your terminal (the screen).

Redirection

The “>” symbol is used to redirect output into a file. We saw an example of this in Chapter 1 with the command

cat daisy kate > fred

which was used to join together the files “daisy” and “kate” and place them in a new file called “fred”. If a file “fred” already exists its contents are overwritten by the command above; otherwise a new file – “fred” – is created.

Using “**>>**” is the syntax to use when you wish to append material to a file which already exists.

Standard error

Typing in an invalid command (e.g. **ls file_1 > file_2** when no “file_1” exists) results in an error. In this example, the error on your screen would be **ls: file_1: No such file or directory**.

Standard error has to be considered by for example programmers who sometimes need to capture error messages, but this is beyond the scope of this course.

Task 3.5 Redirection

Using “>” and “>>”

➲ Use the command **date** – you’ll see that when this command is issued the system returns details of today’s date and time.

Imagine you want to write this output to a file and not just the screen; try

date > datefile

and the output of **date** will be sent to a new file, *datefile*. You can check the contents of the new file by doing

more datefile

➲ Using **date** again, append a new date and time to the end of *datefile*

date >> datefile

Look again at the contents of *datefile* –

more datefile

Pipes

- The output from one command can be used as input to another...
  - string several commands together in a pipeline
  - the “ **|** “ symbol denotes the use of a pipe in a command
- One useful application of this is long output from **ls**
  - **ls** can be “piped” into more
  - output of **ls** can then be easily viewed a page at a time

Using pipes

In Unix, commands containing the vertical bar symbol (**|**) show that a pipe has been used.

For example, the command

ls –l | more

means that the command ls is being “piped” into more.

Pipes allow the standard output from one command to be used as the input to another command.

Specifically, what a pipe does is take the standard output from the command to the left of the pipe symbol and uses that as standard input to the command situated to the right of the pipe.

*Pipes differ from simple redirection (**>**) in that pipes pass output from a command not to another file, but to another command.*

Task 3.6 Pipes

Pipes: a simple example

An often-used example of where a pipe can be handy is in situations where piping **ls** into **more** allows tracking, a page at a time, what might be a long list of files in a directory (otherwise, the output would simply scroll by rapidly to leave just the final page visible).

➲ Do

ls –l | more

and **more** will list the files you have in the directory one screen at a time.

If the directory you are using **ls** on doesn’t contain many files, there is no need to pipe **ls** into **more** as the results will fit onto one page of output anyway.

Grep

- The **grep** command is used for searching
  - lines within a file that contain a pattern

- one file or multiple files

The command **grep** searches for patterns in text, and can also be used in combination with a pipe. For example,

ls –l | grep –i fred

combines the command to list files with an additional instruction, passed on via a pipe, to print the names of files containing “fred”.

Options when using grep

-i	ignore case sensitivity (e.g. would find “bob” and “Bob”)
-n	displays the matched lines and their line numbers

See the manual page (**man grep**) for more options.

The **grep** command can be used in combination with wildcards; for example

grep work *.html

would search for instances of the word “work” in all .html files in the current directory (note, one space exists after the search term of “work”, but no space exists between the asterisk and the dot).

Task 3.7 grep

Using grep and pipes

➲ Search through a system directory for all files containing the letters “ls”

ls /usr/bin | grep ls

Looking for others

➲ Find out the username of the person sitting beside you. Then try a combination of the **who** and **grep** commands to search for them. (The **who** command provides information on who is logged on to the system)

If their login name was for example “mike” you could do

who | grep mike

If “mike” was logged on, the output would show for how long they had been logged on to the system.

You can use very short patterns that may match many things. Unix 2 will cover more of these but a simple example is

who | grep mi

would return output on for example “mike”, “michael”, “micky”, “michaela”... in fact any line with “mi” anywhere in it

Commands start processes

- Commands are associated with *processes*
  - a process is an instance of a program being run
  - a process starts when the return key is hit to execute a command
  - processes are identified by PIDs (Process IDentification numbers)
- Can investigate your own processes (and those of others)
  - use the **ps** command
- Processes as troubleshooting and diagnostic tools
  - you can use **ps** to identify problems
  - you can terminate processes belonging to you

Processes in Unix

When a command is issued in Unix, it is associated with a process. In a multi-user system, one person running one program would be associated with one process; four users logged on running that same program would mean four separate processes.

Each process is identified with its own PID. The **ps** command displays information about the processes you are currently running.

The **ps** command gives output on the command being run (the PID), and the time taken for this to execute ("TIME"). Alternatively, **ps –f** provides a fuller listing including the time at which the command was issued ("STIME").

An alternative form, **ps –fu username** , shows the active processes of another user (where "username" is someone else's login name).

Processes belonging to you can be terminated using the **kill** command:

kill PID or **kill –KILL PID**

The **kill** command sends signals to running processes such that they (should) be terminated by the operating system.

kill –KILL is a stronger variant of **kill** that will remove particularly stubborn processes; it should however be used as a last resort as it is not a "clean" way of killing off processes (for example, its use may prevent the writing of log files by a program that might later allow fault diagnosis to occur).

Task 3.8 Investigating processes

Experimenting with processes

➲ See which processes are active just now:

ps

As well as the process associated with this command, the output will show a PID for bash.

As mentioned in the first chapter, bash is the shell we are using.

Terminating processes

Processes can be killed using the **kill** command.

Don’t try to terminate the bash process as without a functional shell (in our case, the bash shell) there is no way to type in commands or run programs!

➲ In the meantime, you could open a new shell of a different kind such as the Bourne **sh** shell:

sh

Note that the terminal prompt changes.

➲ Do **ps** again – it should show that sh is active.

➲ Use the **kill –KILL** command to kill the sh process and return to the bash shell; note the prompt reverting back to the bash shell.

➲ Verify that **sh** has gone by issuing the **ps** command once more.

Remember, it’s not possible to kill off other users’ processes!

Foreground and background

- Processes can be
  - in the foreground
  - in the background
  - suspended
- Commands in the foreground
  - command executing now
  - shell returns prompt once current command finished
- Commands in the background
  - jobs can be sent to the background
  - prompt returns for a new command while background task is still running
- Suspended jobs
  - can be restarted, either in foreground or background

Unix job control

Unix job control allow several tasks to be performed simultaneously.

The default situation is for a command to be typed in, followed by the return key for the command to be executed. When this happens, the shell will not provide the prompt for a new command to be typed in until the current command has finished executing. This normal situation is called running a *foreground* job.

If however you type in a command followed by a space and then an ampersand (&) the shell will return you to a prompt for the next command, and execute the command just typed as a *background* job.

Stopping jobs

Processes can be stopped by typing <ctrl> <z> (the control key, followed by the z key).

If you wish to start a suspended job again, but have it running in the background, type

bg

A background job can be brought back to the foreground if it was started during the current login session. (Otherwise, for jobs that commenced during another login session, the kill command is needed. We'll just be dealing with commands in a single session in this course.)

To restart a suspended job, so that it will again run in the foreground, type

fg

When <ctrl> <z>has been used to suspend a process, either **fg** or **bg** will reactivate it.

All processes (foreground and background) have their own PID and can be monitored using **ps**.

Remember the earlier **kill – KILL PID** syntax (substituting the actual number of the relevant process) to kill a process off.

Why does this matter?

It depends on the Unix environment you are in. Large jobs can make heavy demands on a system's processing power. Programs such as large simulations may use a lot of RAM and can consume a lot of the system's resources (available memory, disk space). In multi-user environments users will often be competing for the available resources with many others.

On many multi-user systems, such as a cluster environment, jobs are frequently prioritised. This could be on the basis of, for example, some of the system being reserved and research groups paying for a certain amount of available compute time.

Potential users of the Edinburgh University ECDF cluster should visit the site

www.is.ed.ac.uk/ecdf/

where information can be found on courses available for users to learn about this system.

Task 3.9 Foreground and background jobs

➲ Start off a process, for example use more to examine a file's contents:

more bigfile

As covered earlier in the course, **more** is a program that allows you to examine a file's contents a page at a time.

After executing the **more** command above, you will get the first page of output (in this case, of "bigfile")

➲ Now do:

^z (the <Ctrl> and <z> keys in quick succession)

to stop the **more** process.

Note that the command prompt returns immediately, ready for the next command to be typed.

➲ Type:

ps

and you'll see **more** is in the output.

➲ The **more** process can be restarted by typing:

fg

and now you're back looking at "bigfile" at the place you left it.

Unix environment variables

- Environment variables
  - many environment variables are set at login
  - examples: **USER** (your login name), **PRINTER** (your default printer)

Variables

The function of variables is to pass information from the shell to programs that are running. Programs "look" for variables that have been set and, where available, will use them.

Your Unix environment is set up in a particular way because, at login, the shell finds information held in initialisation files that are always read. (We'll look at these files soon.)

Programs you may run also have access to the values stored in environment variables.

Many environment variables are set at login. These last for the duration of a session.

Some common environment variables are listed below:

USER	- this is your login name
HOME	- file path of your home directory location
OSTYPE	version of operating system in use
PRINTER	- your default printer
PATH	- denotes which directories the shell should search to find a command

Note...

Environment variables are, by convention, written in UPPER CASE.

Viewing and modifying environment variables

To view all current environment variables set, type:

printenv

Individual environment variables can be checked by using e.g.

echo $HOME

(**echo** tells Unix to display a line of text, and the **$** character is for displaying the content of the variable)

To set the value of an environment variable, simply type its name, then an equal sign, then the new value. For example, if your current default printer was called pr1 and you wanted to change this to be pr2, you would do:

PRINTER=pr2

...note that there are *no spaces* in this command. You would then normally follow this by doing:

export PRINTER

The reasons why you need to use **export** are beyond the scope of this particular course; you will learn more about them if you go on to do Unix 2 and Unix 3. At this stage you should take it that if you set a variable you should almost always export it.

For the current session, the default printer would now be pr2. (Longer-term changes can also be made, see the following section on configuration files.)

The **printenv** command could then be used again to check the new details.

Task 3.10 Environment variables

Looking at environment variables

➲ Examine some environment variables using echo:

echo $HOME

echo $PATH

echo $PRINTER

Changing the content of an environment variables

Earlier, we used the text editor **nano** to create and modify text files.

➲ Use **printenv** to check your current environment variables – you may see an entry for EDITOR, but most likely will not. You may also find **grep** useful.

```
...
EDITOR=something
...
```

If not, make nano your default editor:

```
EDITOR=nano
export EDITOR
```

➲ Do **printenv** again – the EDITOR entry should now be EDITOR=nano (or at least, it will be for the rest of the current session. We'll see in the next section how to make changes that last across sessions.)

Configuration (dot) files

- Configurations files can be edited, allowing modifications to the Unix environment
- Files are read at login time
  - shell startup files are found (some of the "dot" files)
  - bash_profile is used by the bash login shell
  - .bashrc is another configuration file for the bash shell
  - information also available to scripts and other programs run in bash

Looking at Unix configuration (dot) files

In Chapter 1, we saw that the **ls –al** command issued in the home directory gives a list not only of regular files (such as text files you've created) but also some configuration files. These files have names preceded with a dot, hence the alternative term of "dot files."

The file .bashrc for example contains environment variables that are available to the user running bash as their shell. The .bashrc file is specific to each user, and is found in each home directory. (There is a system-wide file too - at /etc/profile – but only a system administrator can alter this.)

Task 3.11 Configuration files

Looking at configuration files

➲ Examine some configuration files, for example .bashrc in your home directory:

more .bashrc

In general, changing configuration files is not something that a regular user would often (or possibly ever) have to do. Furthermore, any higher level administrative tasks would be performed by a systems administrator.

Anyone doing this should make a copy of the file being modified so that a known working configuration file still exists if a changed file later causes problems.

(The Unix2 course covers configuration files in greater depth, and covers the export of shell variables to other programs you may wish to run.)

Compressing Files

- It has often been necessary to reduce the amount of space that a file takes up on disk or the amount of bandwidth it takes to transfer across a network.
- Whilst this is less of an issue with ever larger hard drives and ever faster networks it is often still useful to make files smaller.
- There are a great many commands for doing this on and across all operating systems; the one we'll use today is called "GNU Zip" or "gzip"
- It has a great many options, but we'll be using it in its simplest form;

  To compress a file,

  gzip *filename*

  To decompress a file,

  gunzip *filename.gz*

- Note that when a file is "gzipped" the filename has the suffix ".gz" added to it. When the file is gunzipped this is removed.

Task 3.12 Compressing Files

➲ Look at the size of a file:

ls -l bigfile

Take note of the size of the file

➲ Now type:

gzip bigfile

to compress the file.

➲ Check the new size of the file;:

ls –l bigfile.gz

and note that it’s much smaller (text compresses very well).

➲ Now uncompress the file;:

gunzip bigfile.gz

to get everything back as it was.

Unix and networks

- Unix networking is mature
- A large numbers of web servers run Unix/Linux “under the hood”
- Many cluster and grid computing environments have Unix/Linux as their underlying operating system

Unix and networking

Unix is well suited to being used with computer networks, given the high level of configuration possible and strong underlying security. Consequently, many cluster and grid-style computing environments have a version of Unix as their operating system. Environments such as these permit users to communicate with other computers.

In a cluster for example a researcher might submit “jobs” such as large simulations to a powerful remote machine with the capacity to run the simulation much more quickly.

In the final chapter of this book we’ll be using a simple example of file transfer to allow you to retrieve files for the final practical from a remote machine. This will be done using a method called FTP (file transfer protocol). Using FTP is one way to transfer files across the network from another server to your own machine.

For transferring date where security needs to be rigorous you may also encounter methods such as SFTP (secure file transfer protocol).

Summary

This chapter has covered using some features of the bash shell to make efficient use of Unix. These include the ability to use shortcuts such as filename completion to type commands more quickly, and the history mechanism for reviewing previous commands. Editing the command line and using wildcards can also save time.

You have also seen how Unix processes can be explored and that by moving jobs to the foreground or background you can allow Unix to perform different tasks simultaneously.

Finally, you have had a brief look at some underlying configuration files and the use of environment variables.

New commands covered:

history	grep	who	ps	kill	
fg	bg	echo	printenv	gzip	/gunzip

Next....chapter 4

Chapter 4. Practical

Murder at McGumption Mansion

An extended practical

In this chapter, you have the opportunity to use your recently acquired Unix skills in a longer practical. At each part of the practical you can make progress with one or more carefully chosen Unix commands. In parts there is no one “correct” way to move forwards, sometimes the same result can be achieved using more than one command or variations of a single command. Your tutor and any demonstrators will be happy to help.

The practical is self-contained. All the files you’ll need are made available for you to work with.

In the event of not reaching the end, you can still use the information to work on the practical after today’s course (for as long as the Unix account you’re using remains active – your tutor can advise on this).

Some tips are provided at the end of this chapter to help you. In this chapter, the names of files and directories are in **bold type**, as are actual Unix commands.

Before the end of the course a printed solution will also be provided.

Instructions for getting files for practical

The files for the McGumption Mansion practical can be downloaded from the University’s of Edinburgh’s anonymous FTP site. To do this we will use a command line web browser called **curl**.

From the Eddie3 prompt, do the following two commands:

cd

curl –O ftp://ftp.ed.ac.uk/pub/Unix1/murder.tar

murder.tar will now be in your home directory (do **ls –l** to check; also, that is “).

Tar files are compressed. The command:

tar xpof murder.tar

will uncompress **murder.tar**. Check for a new directory by doing **ls –l** again. This new directory (**case_notes**) contains the files you need.

The background

Major Duncan McGumption threw a small dinner party at his country mansion just outside Penicuik. His wife and son were at table, plus two guests who were to stay the night. The meal was not as pleasant as it might have been - the Major was loud and rude throughout.

The meal began at 9pm and finished late. Guests and family retired to bed. The night seemed uneventful - but the Major was discovered on his bedroom floor in the morning, dead. A bloodstained brass candlestick lay nearby.

It's murder. And there are only four suspects....

How to solve the mystery

On the following pages are three sets of tasks which you will need to complete to gather information about the murder. There is also a short biography of each suspect. As you proceed you can make deductions about what happened.

Once you have completed all three sets of tasks you should have enough clues to eliminate all but one suspect. You can then check your answer to see if you have caught the murderer!

All the Unix knowledge you need has already been covered in the course. However if you get stuck there are hints at the back of this book to point you in the right direction.

Major Duncan McGumption

The Major volunteered for the battlefields of France and Italy during World War 2. A cool head and more than a little luck eased him up the ranks and kept him alive until 1945. He stayed with the Army and was posted to various colonies around the globe, including a lengthy tour in India. His career became unremarkable and in 1972 he was honourably discharged with failing eyesight.

Throughout his travels the Major acquired a small collection of native artefacts. With these and a considerable inheritance he settled into the twin pursuits of business and anthropology. For a time, his shrewd instincts and uncompromising attitude served him well and he was able to buy outright a country mansion near Penicuik.

Sadly, in his last few years the Major became erratic, rude and obsessive, a sour conversationalist and an embarrassment to his family. He was found murdered in his bedroom on the morning after the dinner party.

Part one

A map of the McGumption Mansion

You need to consult a plan of the first floor of the McGumption mansion. Unfortunately, the only one available is in four pieces.

You will find three of the pieces in the directory:

map

The other one is in a subdirectory of the directory:

greek

It has only one subdirectory.

Collect the four map pieces together in one directory. When you have them all, concatenate them together starting with piece one.

Look at the following information to read about the four suspects before going on to part two.

The four suspects (1)

Mrs Maria McGumption

The daughter of a minor staffer in the Diplomatic Service, Maria cut free from her conservative upbringing and lived the wild life after graduating from Oxford. She met the Major in India, at an embassy party she had gatecrashed with friends.

Maria fell in love with his energy and devil-may-care attitude and they were married in 1969. Only a few years later she realised that he was entrenched in the same stuffy values she had rejected, and he was actually a boring old trout.

Their marriage has not been a happy one for some years and she would leave the Major were it not for the fact that she has grown to like his money.

The four suspects (2)

Martin McGumption

Martin's upbringing was a haze of mixed messages. His father hammered on about duty and making one's own way in the world, while his mother smoked Turkish cigarettes and told him to grab what he could while he was still young.

He has responded by becoming a waster, with a dismal education and dull friends. Although he lives in Edinburgh, he survives on his parents' money and is a great disappointment to both.

The Major keeps Martin on a short leash by constantly threatening to cut him out of the will. Thus he is often at the mansion for "family time".

The four suspects (3)

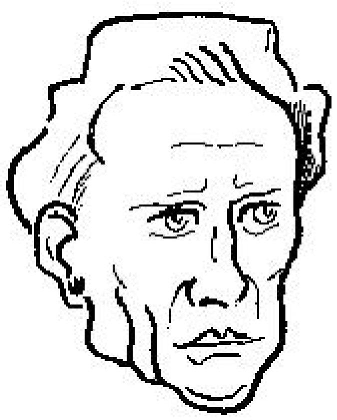
Miss Daphne Postlethwaite

Orphaned by the Blitz on London, Daphne applied herself to study with a single-mindedness and determination which she retained through her spell at the London School of Economics and into commerce.

A woman trader was a curiosity in the sixties when she began; nevertheless, Daphne's tireless, hard-nosed approach and instinct for the right price won respect for her. She met the Major in 1991 and they found it easy to cooperate. Thus began a profitable partnership which ended around Christmas 1994 when Daphne abruptly retired from the market to live in a quiet London suburb. She now plays tennis seriously and has won two amateur championships.

The four suspects (4)

Mr. Sidney Bus



Sidney is the curator of a small museum in Derbyshire. The Major wrote to him occasionally after leaving the Army, with specific questions about various artefacts he had acquired. Sidney was happy to help initially and the letters continued, hinting that the Major might donate some of his discoveries to the museum.

However, no donations were ever made and the Major received a lot of free advice from Sidney. Over the last year the Major had become increasingly scornful of Sidney and what he regarded as a small, uninteresting museum which nobody visited anyway.

Part two

The facts

You now know a little about each suspect. You also need to learn some more information about them and some background to the evening of the murder.

Look in the directory named:

facts

There are four files in that directory containing facts about the murder and the suspects. However, it will not be straightforward to read any of them.

You need to do three search-and-replace operations to decode the file **facts1**. There is a hidden file in the same directory which will tell you which substitutions to make.

You may not have permission to read the file **facts2** and therefore will need to change the file permissions.

The full name of the file which begins **facts3** will be very difficult to type. You will need to find a way to refer to the file without typing the entire filename.

Getting the information from the file **facts4** is likely to involve two steps. However, the file itself should help you with one of those.

Tip – to execute a file as if it were a command, you need to specify the path; however, if it is in the same directory as you, typing

```
./file
```

(note the ./ prefix) should be sufficient.

Part three

Statements by the suspects

Each suspect was interviewed the morning after the murder. If you can obtain the statements of all four, you will have enough information to solve the mystery.

Look in the directory named:

statements

The four statements are available there.

Sidney's statement is filed in the directory **sidney**. It is a compressed file.

Martin's statement is in the file **martin**. However, some other text has got mixed up with it. To obtain the correct statement you will have to find a way to extract only lines containing the capital letter **I**.

Daphne's statement is in the file **daphne** but it has been encoded. You must execute the file **./filter1** using **daphne** as standard input, then pipe the result through the script **./filter2** in the same directory.

(See the tip for **./facts4** for why we use ./ to execute a file).

You can read Maria's statement by executing the file named **maria**.

Checking your solution…

Once you have made your deductions and are ready to accuse one of the suspects, there are only a few short steps to let you check your solution.

Change into the directory

check

Type the following command *exactly*; it begins with a dot and a space.

. go

Now follow the instructions carefully and you will find out whether you are correct!

See the next page for some further hints and tips that may help you if you haven't managed to complete all the steps...

Hints and tips

Part one

Having trouble finding the fourth piece of the map?

Certain options to the **ls** command will show you which files are subdirectories.

Are you having difficulty putting the four map pieces together?

There are at least two ways you could do this. Either you could append pieces two, three and four to piece one, or you could create a new directory to put all the pieces in. You will probably have to know about redirection of standard output.

Part two

Can't locate the hidden file?

An option to the **ls** command will make it show hidden files.

No idea how to do a search-and-replace?

Look back at the section on how to use the text editor called **nano**.

Don't know how to execute a file?

Just type its pathname (for example: ./file)

Can’t get the file to execute?

Perhaps the file’s permissions are not set correctly.

Part three

Can't find Sidney’s statement?

The command **gunzip** looks for a file with a particular extension to its filename which labels it as compressed.

Martin’s statement looks like gibberish?

There is a command called **grep** which searches a file for a text string and only prints the lines containing that string.

Unsure how to handle the pipe for Daphne's statement?

You must first specify what you want to execute, then what file to use as standard input, then where to pipe the result to. There are particular symbols to indicate standard input and pipe.

Having trouble deleting a directory to get Maria's statement?

The command **rmdir** deletes an empty directory. However, you perhaps need to use an option to the **rm** command along the way. There are a few different ways to complete this section.

You will also be given a printed solution before you leave today's class.