CS&IT
UNIX SURVIVAL GUIDE

RMIT School of Computer Science and Information Technology

Table of Contents

Quick Start Guide for new students

Now that you have enrolled in your degree program in Computer Science and Information Technology what will you need to be thinking about? The next few pages are to point you in the right direction and to help you feel like you know what is going on. There is much information on the web and in your enrolment pack about courses, administration and support services. Become familiar with where to find things now, then when you need them you will be ready.

The words in bold are terms used by the University if you do not understand them, please ask someone. The page references are for this manual.

Your Program

This is the degree you are doing, it is managed by CS&IT and any questions you have about what you should do will be answered by a **Program Advisor,** reading the **Program Summary** and meeting with a **Teaching and Learning Advisor.** see page 23

Your Program Summary has the rules you need to follow to complete your degree and graduate. It is used with **Enrolment Online** to select courses each year, and to make any changes at the beginning of the **semester**. You must complete the specified **core courses** (these are compulsory), courses from your **specialisation** or **major** study and **electives**. The order you do things is determined by the availability of courses and the **prerequisites** you have already completed. If you have **advanced standing** you will need to know the courses from which you have been **exempted**. Your Program Advisor has this information.

Your Courses

Each semester you will typically study four courses. They must be ones that contribute to your program and you choose them after you have had advice.

A course has timetabled teaching activities and learning resources both online and in print. The **Course Guide** details everything that is available to you and all that is expected from you, including how you will be assessed. You will find the course guide in the **Learning Hub**, or by searching for it. See pages 24, 25.

The Course Guide has two parts, the **Course Overview** (part A) helps you select the course, and the **Course Detail** (part B) which gives specific information for this semester including the lecturer, textbook, topics and assessment due dates. You must read these.

Many people are involved in supporting your learning in each course.
Lecturer – manages the course and the teaching team. Delivers lectures.
Tutors – facilitate tutorial sessions which are small classes where you discuss exercises and questions that you have prepared in advance. Tutors also mark assignments.
Head Tutor – If a large number of students are enrolled in the course, a head tutor will assist the lecturer in managing the course. The Head Tutor is often the main contact person for students.
Lab Assistants – Practical classes held in computer labs have assistants who will help you with the given task and mark assessments in the lab as required.
Mentors – some programming courses have student volunteers who you can meet with to talk about that programming language.
Teaching and Learning Advisors – will help you with general study skills and specific issues that are affecting your study.

Keeping in touch

Communication is the key to university study, you will do lots of listening and reading, but it is also important for you to talk and write.

As a student you are expected to be active in your learning and also to take responsibility for administrative matters. You will therefore need to be aware of deadlines, changes and opportunities. It is assumed by the School and the University that you are attending to your study and keeping in touch.

Email

The primary communication tool used by staff and students at RMIT and in CS&IT is **student email**. See page 27 You will receive lots of messages; read them and archive them as you may need the information later.
There are different types of messages:

Broadcast messages from RMIT are sent to all students in the form of announcements and bulletins. You may not be expecting anything as you have not asked a question, but it is important you scan through each message, some of the information will be relevant to you.

Broadcast messages are also sent from CS&IT to students enrolled in our programs. Again you didn't ask for anything but the information will be relevant to you because you are a student in the program. Read, make note of deadlines and activities, and act on the information.

Email from your lecturer or head tutor may be sent to everyone enrolled in the course. This is usually when something urgent needs to be said to all students.

Personal email will be sent to you as replies to questions or comments that you have sent to an administrator or lecturer. If you do not get a reply to an email, think of another way of communicating, perhaps talking after a lecture or attending consultation time. You might see a Teaching and Learning Advisor if you have problems like this.

Personal email between students is an important learning tool, especially if you are working on a group assignment. If you do not hear anything from a group member you should try another way of getting in touch. If this fails alert your lecturer, head tutor or a Teaching and Learning Advisor.

It is your responsibility to check your email, and clear your mailbox.

Online Discussion

Each course will have at least one area (usually in the **Blackboard** area of the Learning Hub) where students can post and read messages about course work. All other students can then read and learn from the discussion. Many students are a bit shy about using this tool and only read. This is ok, but it is much better to ask a question early rather than wait for someone else. The discussion area is usually managed by the Head Tutor, but other tutors and the lecturer will add comments. You will need to learn their names so you know who the answer is from.

Newsgroups are discussion areas where you are free to read and post items relevant to that group. A few courses use newsgroups, but the main ones of interest to CS&IT students are related to IT. You can read online or through UNIX. See pages 30, 31

Face-to-Face Communication

While you are on campus talking with someone will often be necessary, there are a number of ways to do this.

Course Consultation

Time is set aside by each lecturer or head tutor to meet with students outside class time. It can be that you need to clarify part of a topic that you don't understand, or you want to check up on something related to assessment. The time and location for consultation is given in the first lecture and will be in Blackboard in the staff information area.

Meeting a Program Advisor

Anything related to your program can be discussed with your advisor. At busy times of the semester you might be asked to make an appointment via email, or wait to meet outside their office at the time they have allocated for meetings. Your advisor has probably put a notice on their door regarding meeting times.
Teaching and Learning Advisors are available to meet students most days. You can email for an appointment, make a booking using the staff booking system or just knock on the door.

Administrative Questions

The **Main Office** for CS&IT is 10.10.07 and has an enquiry counter. Staff here will either answer your question or refer you to the Hub or an Advisor.

Duty Programmers at 10.10.13 will answer technical questions. see page 7

The Hub located on Level 4 of Building 12 is the place for all questions about being a student. The staff will either give you information or refer you to the correct service. The Hub also has lots of information about **Student Services** offered by RMIT.

Student Staff Consultative Committees

You are invited to become a part of the process for improving the learning experience for students in CS&IT. These **SSCC** meetings provide direct communication between students and staff of the school. The minutes of these meetings are available online and include issues and actions taken. All programs and year levels need representatives for these committees, the meetings are friendly and assistance and training for representatives is available. http://www.rmit.edu.au/csit/sscc

Group Meetings

Students like to work together and finding a space can be a challenge.
The Swanston Library has eight discussion rooms equipped with computer and white board which can be booked for up to two hours per day for group assignment work or informal study groups. http://www.rmit.edu.au/library

The Student Lounge on Level 9 of Building 14 has tables and chairs and is useful for informal discussion.

Learning Resources

Here is a quick look at the tools and activities you will engage with while you are studying in CS&IT.

Time

The **Academic Calendar** has the semester and exam dates and deadlines for making changes to your enrolment. These deadlines are final.

Your **Timetable** is your personal schedule of classes during the week. see pages 23,25 Once you have filled in lectures, labs and tutorials you will need to schedule your personal study and coding practice time. Allow a minimum of **12 hours for every course** you are doing. Full time study is a commitment to 48 hours per week; it is recommended that you make this your priority. Paid work can then fit around your study, perhaps a day on the weekend or two afternoons during the week. The semester break is a good time to do extra paid work and get money in the bank.

A **Study Planner** is the whole semester's work at a glance. You fill in the weeks with course topics, assessment starting and due dates as well as progressive submission dates. The Student Union produces a free planner or you can draw up your own. Some lectures make a planner for their course with week by week links to lecture, tutorial and lab tasks. Your job is to combine them all so you know what you are doing each week, and also planning for busy times when a lot of assignments are due. Teaching and Learning Advisors can help with this.

Classes

You are encouraged to attend all the classes for each course. Your lecturer assumes that this is what you are doing, and that you come to class prepared for that topic. This means reading the **lecture notes** and text book chapters before the lecture. The tutorial will challenge you with questions related to the topic from the previous week. This gives you time to prepare your answers, to think about the ideas, and to try the programming examples. Lab classes often relate to assignment work. Key concepts and skills that you need for the assignment are presented in exercises. Once you have completed them successfully you should be able to work on that part of the assignment.

If you miss classes because of something unavoidable such as ill health, you should meet your lecturer or head tutor during consultation time to get advice on how to quickly catch up. Even if you are ill, you can often keep in touch with the online elements of your course. However serious illness requires advice regarding **Special Consideration**. Teaching and Learning Advisors and Student Services can help.

If you get behind with topics and assessment, tell your lecturer, tutor or Teaching and Learning Advisor as soon as you can. Don't just try and catch up weeks and weeks of work without advice.

Study Skills

The RMIT Study and Learning Centre at the Hub provides free assistance in English language, study skills, academic writing, and Maths. There are online study skills tutorials too. http://www.rmit.edu.au/studyandlearningcentre

Basic Unix

While the Windows PCs and terminal servers are available for you to use, most Computer Science assignments need to be completed on the Unix servers (Yallara and Numbat).

What is Unix?

There are actually many operating systems that are "Unix-like". Both Yallara and Numbat run Sun Solaris. You may have heard of some others: Linux, FreeBSD, Mac OS X, SCO, and so on. While internally these operating systems are very different, they all share a common interface, meaning the skills you learn here are generally applicable to all Unix-like operating systems.

If you have used Linux or another Unix-like operating system before you may already be familiar with many of the commands introduced here; be aware, however, that there are some commands that are either Solaris or RMIT specific.

Using the terminal

When you first log into Yallara you are presented with two *xterms*. Remember you can always open another terminal from the start menu if you need to; it is often convenient to have a few open when doing several tasks at once.

After the welcome message is the *prompt*, which lets you know that the terminal is ready for input:

```
yallara.cs.rmit.edu.au%
```

In this manual we usually don't show the prompt; just the commands you need to type. For example, try typing "date" and pressing enter:

```
date
```

This runs the program "date" and displays the output to the terminal. Most commands accept *arguments*, or *parameters*, which modify their behaviour. For example, if you type:

```
date -u
Sat Feb 11 14:17:57 GMT 2006
```

the "date" program is run with the argument "-u", which then displays the time at GMT (a different timezone). This time we have also shown the result, in a smaller, italicised font.

When you need to repeat a command with the same or similar arguments, you can save typing by pressing the up-arrow on the keyboard. This shows the last command you entered. You can continue pressing the up and down keys to move back and forth through the history of commands you have typed in the current session.

Getting help

When you know the name of a command, but can't remember what arguments it takes, you can look it up in the *man* pages (short for manual pages). For example, to find out all the options for the *date* program:

```
man date
```

You can scroll through the page with the spacebar. To quit viewing the manual page and return to the prompt, press **q**.

If you don't remember the name of the command, *man* won't help. You will probably need to ask a friend, lab assistant, use this manual or Google.

Files and Directories

Most people are familiar with Windows and DOS drive letters where disks are accessed as C:, D:, and so on. In Unix there is no such distinction. All files, regardless of where they are stored, are accessible through the *root directory* or "/". Whereas on Windows directories (sometimes called folders) are separated in a path with a backslash ("\"), on Unix you must use a forward slash ("/"). Here are some example directory paths:

```
/
/home
/home/j
/home/j/joebloggs
/usr/local/bin
```

Paths are read from left to right. The first slash ("/") means to start at the root directory (you can start in some other places as well, as we shall see shortly).

You can use the **ls** command to list the contents of a directory:

```
ls /
```

This will print out all files and directories directly below the root directory. Some of these directories and their purpose is explained below:

`/bin`	Standard Unix programs
`/etc`	Configuration files
`/home`	Students' home directories
`/public`	Academic material for specific courses
`/tmp`	Temporary files

Look inside the /**home** directory:

```
ls /home
```

This directory contains every student's personal *home directory* organised by the first letter of their username. For example, if your username is **joebloggs**, your home directory is /**home/j/joebloggs**. Try to list the contents of your own home directory now. Your home directory is the place where you should store any assignments you are working on, email, and is where applications can store your preferences.

As a shortcut to writing out your whole home directory path, you can simply write ~ (tilde character, in the top-left corner of the keyboard). Assuming your username is **joebloggs**, the following two commands are equivalent:

```
ls /home/j/joebloggs
ls ~
```

What happens when you try to list another student's directory?

```
ls /home/a/aholkner
/home/a/aholkner: Permission denied
```

You are sharing the computer Yallara with all other Computer Science staff and students, but there are systems in place that prevent you from reading or modifying other people's data, as well as data that could interfere with the upkeep of the system. Permissions are discussed in detail later.

Test yourself

- What is the path to your home directory?
- What files and directories exist in your home directory? Can you guess what they are?
- Log into Numbat now. Can you see any differences in either your home directory or the root directory? Why do you think that might be? Return to Yallara when you are done.

Working directory

Every terminal window you have open has a *current working directory*. This is the directory that applications will load and save their data to or from by default. You can print the entire path to the working directory with the **pwd** command:

```
pwd
/home/j/joebloggs
```

If the directory or file in which you are interested is in the current working directory, you don't need to specify a complete path to it.
For example, you can list the contents of the current working directory by typing **ls** without any arguments:

```
ls
```

Creating directories

There is not much interesting in your home directory yet, so let's create some directories. To create a directory, use the **mkdir** command. Let's say you want to create a directory called "courses" within your home directory:

```
mkdir courses
```

Remember that since your current working directory is your home directory you didn't need to type in the whole path. Of course, you could have if you wanted to; the above is equivalent to:

```
mkdir /home/j/joebloggs/courses
```

List the contents of your home directory now and make sure you can see *courses* as one of the items.

Let's assume you are taking 3 courses: "maths", "programming" and "databases", and create a directory under *courses* for each one:

```
mkdir courses/maths
mkdir courses/programming
mkdir courses/databases
```

Now list the contents of *courses* and make sure you can see all of these directories.

```
ls courses
databases  maths   programming
```

Changing the working directory

Earlier we saw that the current working directory was your home directory. Let's now *change directory* to the "courses" directory:

```
cd courses
```

Now that the working directory has changed, what do **pwd** and **ls** do?

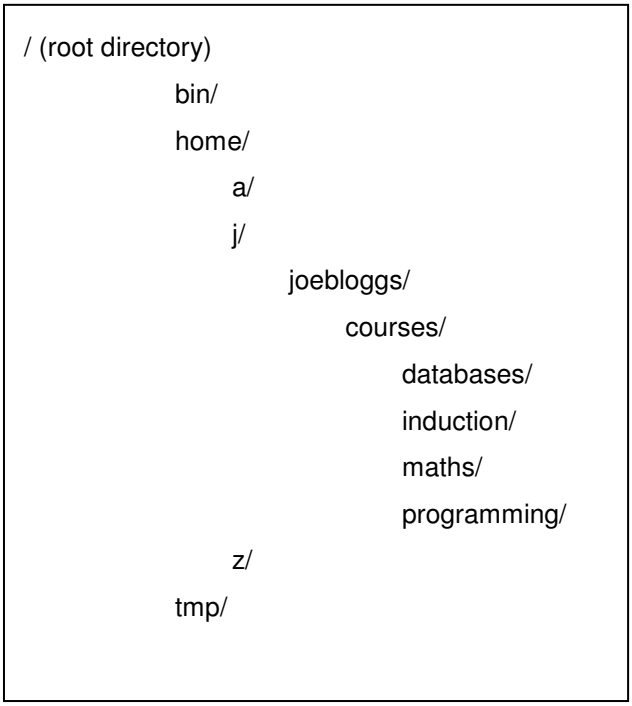
```
pwd
/home/j/joebloggs/courses
ls
databases  maths   programming
```

Create one more directory under *courses* named "induction":

```
mkdir induction
```

Note that we didn't write *courses/induction* this time, as we are creating the directory directly within the current working directory. List the contents of the current directory and make sure you now see all 4 directories.

The layout of directories is often referred to as a *directory tree*, and is displayed like this:



In this diagram you can see that the *joebloggs* directory is the parent of *courses*, which in turn is the parent of the four subject directories you created. We changed directory from *joebloggs* to *courses* by typing

```
cd courses
```

You can't, however, change back to *joebloggs* from *courses* by typing **cd joebloggs**; this is because *joebloggs* is not visible from *courses.* You can change back to *joebloggs* by typing its full path:

```
cd /home/j/joebloggs
```

or, you can change to the parent directory with "**..**" (two full-stops, commonly pronounced "dot-dot"):

```
cd ..
```

The "dot-dot" can appear anywhere in a regular path to signify "the parent":

```
cd /home/j/..
pwd
/home
cd /home/j/../a
pwd
/home/a
cd /home/j/joebloggs/../../..
pwd
```

Test yourself Change directory to the *courses* directory

- Write down the *absolute path* of the following paths (an absolute path is one that begins at the root directory):
  - `courses/`
  - `courses/..`
  - `../`
  - `../..`
  - `courses/../courses/../courses`
- Create two further directories under *courses/maths* named "calculus" and "algebra".
- Change directory to *algebra*, from there list the contents of *courses*.

Tab completion

When typing the names of files or directories on the command-line, you can often type just the first few characters, then press the **tab** key to fill in the rest automatically. For example, you can save a lot of typing when changing to the *courses/maths* directory by just typing:

```
cd c(tab)m(tab)
```

which is expanded as you type to:

```
cd courses/maths/
```

Hidden files and directories

Change to your home directory and list all the files. Now add the **-a** option to **ls**:

```
ls –a
```

You should see about 20 extra files and directories that weren't in the standard listing. These are hidden files, and have a full-stop (".") as the first character in their name. Typically they are used to store application preferences and caches.

Create a hidden directory in *courses* named ".hidden":

```
mkdir courses/.hidden
```

Make sure you can see it only when you use the **–a** option with **ls**.

At the beginning of the listing of hidden files in each directory are the two special directories "**.**" and "**..**". We already know that "**..**" refers to the parent directory. The "." ("dot") directory refers to the current directory. Ordinarily this is not needed on the command line, but you may find you need it at some stage.

Check now that the following three commands are equivalent:

```
ls courses
ls ./courses
ls ./courses/.
```

Checkpoint directory

There is one more special directory that exists on the file server used by Yallara and Numbat. It is named "**.ckpt**" and does not even show in a **ls -a** listing. You can, however, see inside it:

```
ls .ckpt
```

The "checkpoint" directory contains backups of the current directory that are made automatically at regular intervals. For example, you might see a directory called **.ckpt/2006_02_05_13.15.01_GMT**, which is the backup that was made on the 5th of February at 1:15 PM, GMT. If you list the contents of that directory you will see the files that existed in your home directory at that exact time.

This automatic backup is a feature of the RMIT Computer Science servers and is generally not available on standard Unix systems. Backups are not kept forever, so you should make a habit of continuing to backup your work both at home and university onto CDs and storing them away from your computer.

Note that you cannot modify files or directories in the checkpoint directory, but you can view them and make copies back into your regular home directory.

Creating and editing text files

Many files you work on in your Computer Science subjects will be plain text files (they have no formatting or fonts like a Microsoft Word file, for example). There are many programs you can use to edit these files. One very powerful and flexible editor is **vim**, which is introduced in the next section of this manual. Regular practise is recommended so that you become proficient editing using vim.

For now, however, we will use a much simpler editor called **nedit** which only runs in xterm. Another editor you might use is pico.

Change into the *courses/maths* directory and start editing a new file called "assignment1":

```
cd courses/maths
nedit assignment1
```

nedit behaves much like any Windows text editor, though it does have some helpful features when programming. Type a few lines of text and save it with the File,Save menu or by pressing Control+S. Quit nedit with the File,Exit menu or by pressing Control+Q.

Renaming, moving, copying and deleting files

You should now have a file *assignment1* in the *maths* directory. Let's say you wanted to rename it to "assignment1.txt":

```
mv assignment1 assignment1.txt
```

The first argument to **mv** is the original file name, the second is the name you would like it renamed to. **mv** won't actually output anything; you will need to list the current directory contents with **ls** to check that the result is what you expected.

You can use the same command to move a file to another directory:

```
mv assignment1.txt ../induction
```

This moves the file *assignment1.txt* to the directory *../induction*. Remember that the double-dots ("..") indicate to start from the parent directory, which in this case is *courses*.

Similarly, you can move entire directories with the same command:

```
mv ../induction ~/
```

This moves the *induction* directory to your home directory (remember that the tilde ~ is short-hand for your home directory).

The **cp** command works similarly, except that instead of moving or renaming a file it makes a copy:

```
cp ~/induction/assignment1.txt ./
```

This makes a copy of *assignment1.txt* and places it in the current working directory (remember that "." means to start from the current directory), which if you have not changed it is the *maths* directory.

If you want to copy an entire directory you need to specify the **–r** argument to **cp**:

```
cp -r ./ ../maths-backup
```

This copies the current working directory (*maths*) to the directory *~/courses/maths-backup*.

To delete a file, use the **rm** command:

```
rm ~/induction/assignment1.txt
```

To delete an entire directory, add the **-r** argument to **rm**:

```
rm -r ../databases
```

Careful! The automatic backups only take place every few hours; if you delete something you need there may be no way to get it back if it's too recent. The **mv** and **cp** commands won't give you any warning if you overwrite another existing file.

Test yourself

- Rename the *maths-backup* directory to make it hidden. Hint: how does the name need to change to make it not show up in a normal directory listing? Check with **ls** that it doesn't show up, then show that is there when you supply the appropriate argument to **ls**.
- Create a text file in the *induction* directory listing all the commands you have learnt so far and call it *notes.txt*.
- Make a copy of this file (*notes.txt*) in the same directory and call it *commands.txt*.
- Find a file in the checkpoint directory (the automatic backup directory) from two days ago and copy it into the *induction* directory.
- Delete the original *notes.txt* file.

- In the space below, write out the whole directory tree starting from your home directory and including all the files and directories you have added.

A note about Unix filename conventions

You will have noticed that most of the directory and filenames given in this manual are composed entirely of lower-case letters. This is entirely optional: Unix systems allow filenames to have any form of letter, including most punctuation marks and characters from non-English character sets. Unlike Windows and DOS, however, Unix treats upper- and lower-case letters as different. In other words, you can have a directory containing the files **test.txt**, **TEST.TXT** and **Test.TXT**, and they would all represent different files. As you can imagine, this can get quite confusing.

For the sake of simplicity and clarity, most users elect to name their files entirely in lowercase English letters, with the addition of the hyphen ("-"), full-stop (".") and underscore ("_") punctuation marks. Some programs may not work correctly with other punctuation marks in the filename, as they have special meaning to the command environment, as you will see.

You will have also noticed that unlike Windows, many files do not have an extension (like **.txt**). Again, these are optional, however they can help you to organise your files and they let programs know what kind of data to expect.

Using Windows files on Unix

Often you will want to work on an assignment at home on Windows, then copy it back to Yallara for submission (see page 15 for tips on copying files from home).

There is a sample Windows file at */public/induction/windows.txt*. Copy this file to your *induction* directory, and change to that directory:

```
cp /public/induction/windows.txt ~/induction
cd ~/induction
```

If you open this file with **nedit** it will look fine. Now try opening it in **gvim**, another text editor that you will learn about in the next session:

```
gvim windows.txt
```

What's wrong with the text? Can you see the extra **^M** characters at the end of each line?

In Windows and DOS, each line of a text file is terminated by two characters: a carriage return (CR, code 13) followed by a line-feed (LF, code 10). In Unix, lines in a text file are terminated with just a line-feed. Some Unix programs (like nedit) can handle both types of file, but most will not work correctly with the Windows style line-endings.

In gvim, the carriage returns appear as **^M** characters, which is harmless but annoying. In Perl (a programming language), however, a program written with these characters will just fail to work. This can be quite a shock to a student who has worked all weekend at home on an assignment only to find it does not work at all at uni.

Luckily, the solution is simple: simply remove the carriage returns from the file. You can use the **dos2unix** program to do this:

```
dos2unix windows.txt unix.txt
```

This converts the file *windows.txt* and saves it as *unix.txt* (It will print out a 2-line warning about an unknown keyboard layout which you can safely ignore). Open this file in gvim and check that the lines now look correct.

Filename globbing

List the contents of the */public/induction* directory. There are a series of files ending in *.ant* and *.syn*. These are lists of words that are antonyms or synonyms of the filename, respectively.

If you were to copy all of the synonym files (those ending in *.syn*) into your current directory, you would need to type the **cp** command 8 times. Actually, there is an easier way:

```
cp /public/induction/*.syn ./
```

The "*" character (an asterisk, commonly called a "star") is a placeholder, or "globbing operator" for any sequence of characters. You can use it in place of part of a filename where you want to list all the files that match the pattern.

It works on all commands, not just **cp**:

```
ls /public/induction/*.ant
ls ./d*
```

Warning: using a glob is equivalent to typing out all the filenames it matches in sequence. This is not always intuitively what you want. Consider what happens when you type:

```
dos2unix *.txt
```

This is equivalent to typing (assuming there are two files in the directory):

```
dos2unix file1.txt file2.txt
```

Instead of converting all the files in the directory, you have overwritten *file2.txt* with *file1.txt*! Later in this manual we will write a script that can convert all the files in a directory.

Finding text within files

You should now be in a directory with several *.syn* synonym files. If you look at these files with nedit you can see that they are just simple text files with one word per line.

You can search a file for a word or phrase with the **grep** command:

```
grep perplexed *.syn
```

This searches all files matching the glob **.syn* for the word "perplexed". If any are found they will be printed to the terminal. Specifying the **-n** option also causes the line number that the word was found on to be printed:

```
grep -n perplexed *.syn
```

This is a particularly useful feature when you start dealing with large amounts of source code, and need to find where a particular variable or routine is being used.

Finding files within directories

grep is very good for searching within files, but it doesn't help when you know the name of a file but can't remember which directory it is in. Do you remember where the *assignment1.txt* file is? You can use the **find** command here:

```
find ~/ -name assignment1.txt
```

Note that the arguments are quite different from grep. First, you specify a directory where the search will start. You could specify the root directory ("/"), but that would take a long time; here we start from the home directory. The **-name** option instructs **find** to show only files with the given filename.

You can use a glob with **find** to locate files with just part of the filename, but you must then surround it in quotation marks:

```
find ~/ -name "*.txt"
```

Test yourself

- Use the **unix2dos** program to add carriage-returns to *assignment1.txt*. Check that you can see the **^M** symbols in gvim. Now convert it back to Unix format and check the output.
- Copy all the antonym files (those ending in **.ant**) from /**public**/**induction** into your ~/**induction** directory.
- What files contain the word "satisfied"?
- What is "addlepated" a synonym for?
- Where under the /**usr**/**include** directory is there a file named **scf.h**?

Submitting assignments

Every course lecturer has their own preference for how you submit assignments. It is important that you follow their directions carefully – simply emailing an assignment will normally result in you failing that hurdle. Some courses use WebLearn in the Learning Hub for assignment submission.

Read the assignment documentation carefully and follow the given instructions for submitting your work. Check the discussion area if you have any doubts. If you are required to format the files in a particular way make sure you do, if you are not sure how the Duty Programmers can help.

A system that is commonly used by computer science courses is turnin.

To submit an assignment with turnin, use the **turnin** command. There is a pretend course set up called *induct2006* which you can practice submitting assignments to. To submit all the *.syn* files:

```
turnin -c induct2006 *.syn
```

If this were a real assignment, you would substitute *induct2006* with the course code provided by your lecturer. You can get a list of all the courses being used by turnin at the moment with the **-l** (lower-case L) option:

```
turnin -l
```

You can submit an assignment as many times as you like; each submission overwrites the previous one. It is a very good idea to submit an assignment every day that you work on it, even before it is finished. Firstly, this lets the lecturer know how often and when you worked on it, and can help prove that your work is your own if a plagiarism case arises. Secondly, it is a good backup in the event that you accidentally lose your own copy, or forget to complete the assignment by the due date – at least you will have submitted something, getting partial marks.

Most importantly, remember to submit all the files in the assignment at once. If you submit them one at a time, each will overwrite the last, and only one file will be submitted for marking! You can see a listing of all the files you submitted with the **-v** option:

```
turnin -c induct2006 -v
```

Disk usage

You have a limited amount of disk space on the file servers. You can check your current usage with the **quota** command:

```
quota -v
Disk quotas for joebloggs (uid 1234):
Filesystem  usage   quota   limit
/home       5912    20000   20000
```

The "usage" column shows the amount of disk space you are currently using, in kilobytes. The "limit" column shows how much disk space you are permitted before limits are enforced (also in kilobytes). In this case, the user has used 5.9 megabytes of their 20 megabyte limit. The amount of space you are allocated is dependent on how many courses you are taking.

When you are approaching your limit (or are over it!) you will need to delete files so you can continue working. It can be helpful to see which files or directories are taking up the most room. You can use the **du** command for this:

```
du -ks ~/*
```

This shows the size of each file and directory (directories recursively include the size of all files and directories within them) in kilobytes in your home directory. Don't forget to check for hidden files:

```
du -ks ~/.*
```

Note that this also lists the total size of the parent directory (".."), which you can of course ignore. You may find that the ".mozilla" directory is using a lot of space; this is due to Firefox's browser cache. You can easily clear this from within Firefox.

Compressing, archiving and extracting files

When you are running out of disk space, an easy way to reclaim some space is to compress files you don't use on a day-to-day basis, but don't want to delete (such as old assignments).

The most common way for compressing files on Unix is using the programs tar and gzip. tar creates one file that contains many files, and gzip, reduces the file size of that file. Thankfully **gtar** can do this all in one step:

```
gtar -czf backup.tar.gz *.syn
```

We pass three options to **gtar**: **-c** means to create a new archive, **-z** means to compress it with gzip, and **-f** is used directly before the name of the file to create. Finally, ***.syn** is the list of files to backup. **.tar.gz** is the standard extension for files made this way, though you may also see .tgz sometimes.

We can see a list of the files in an archive with the **-t** option:

```
gtar -tzf backup.tar.gz
```

To extract the contents of the archive into the current directory, use the **-x** option:

```
gtar -xzf backup.tar.gz
```

Besides gzip, some people are starting to use bzip2 compression on their files instead; this almost always makes files smaller. Archives created this way typically have the extension .tar.bz2, and you can work with them by using the **-j** option instead of **-z**:

```
gtar -cjf backup.tar.bz2 *.syn
gtar -tjf backup.tar.bz2
gtar -xjf backup.tar.bz2
```

Note that there is also a **tar** command which is commonly used on Linux and other operating systems, however the version installed on Yallara and Numbat is different and generally not useful; you should always use gtar instead.

On Windows it is more common to use **zip** files. You can work with them on Unix as well with the **zip** and **unzip** commands:

```
zip backup.zip *.syn
unzip backup.zip
```

Note that none of these commands deletes the original file(s); typically if you are trying to save space you would delete the files after creating the backup and checking that its contents are correct.

File permissions

Earlier we saw that certain files and directories (such as those belonging to other students) cannot be read and gave the error message “Permission denied”. Unix file permissions are quite complicated but it is essential you understand the basics of them so you know how to protect and share your files appropriately.

First, let’s look in more detail at the files in your home directory, by adding the **-l** (lower-case “L”) flag to **ls**:

```
ls -l ~
drwx------   2 joebloggs students    80 Jan 24 10:59 Mail
drwx------   2 joebloggs students    80 Jan 24 10:59 News
drwxr-xr-x   4 joebloggs students  1024 Feb  2 14:05 WINDOWS
drwxr-xr-x   5 joebloggs students  1024 Feb 10 18:05 courses
drwxr-xr-x   5 joebloggs students  1024 Feb 10 19:21 induction
-rw-r--r--   1 joebloggs students    12 Feb 10 21:31 unix.txt
```

Instead of just listing the names of the files, we now have a detailed listing of the files. The columns, from left to right, are:

drwx------	The permission bits for this file or directory. These are explained in great detail below.
2	The number of hard links to the file or directory. You can probably ignore this number for your entire career.
joebloggs	The owner of the file. That's you.
students	The group that the file belongs to. Groups are described below.
80	The size of the file, in bytes. Note that for a directory this does not include the files within it, merely the amount of space the directory itself is taking.
Jan 24 10:59	The date and time the file was last modified.
Mail	The name of the file or directory.

When you access a file, you are classified into one of three categories with respect to the file:

- You are the owner of the file.
- You belong to the group that the file belongs to.
- You are someone else.

To see what groups you belong to, use the **groups** command:

```
groups
students
```

You may be added to more groups for certain courses, or to access a particular resource such as a CD burner. While you can belong to many groups, a file can only belong to one group. By default, all the files you create will belong to the *students* group.

Now look closely at the permission bits for the last file in the earlier list:

```
-rw-r--r--
```

Ignoring the first hyphen (it is a **d** for directories), you can divide the remaining characters into three sets of three characters:

```
rw-  r--  r--
```

Each of these sets corresponds to the rules to apply for a user falling into the respective category listed above. The first set is for the owner of the file, the second for a user belonging to the group that the file belongs to, and the third set is for everybody else.

Each set can have the letters **r** (read), **w** (write), or **x** (execute) set. If a letter is not set, a hyphen ("-") is displayed in its place. The meaning of these letters depends on whether the file is a directory or a regular file:

	File	Directory
r	The contents of the file can be read	The contents of the directory can be listed
w	The file can be written to or replaced	Files can be added to and deleted from this directory
x	The file is a script or program and can be run	The user can change to this directory, and can access files within this directory

So, for the permission bits:

```
-rw-r--r--
```

The owner of the file can read and write it, members of its group can read it, and everyone else can also read it.

The three categories that the permission bits address are called *user* (for the owner), *group* (for members of the group) and *world* or *other* (for everyone else). So we would say the above file is world-readable and user-writeable.

You can use the **chmod** command to change the permissions of a file or directory that you own:

```
chmod g+w unix.txt
```

The **g** refers to the *group* category of users, **+** means to add permission, and **w** refers to the *write* permission bit. In other words, it gives write permission to members of the group. The resultant permission bits will be:

```
-rw-rw-r--
```

Another example:

```
chmod ug+x unix.txt
```

This adds the execute permission bit (**x**) for the owner (**u**) and group (**g**). You can remove permissions with a minus sign:

```
chmod ugo-x unix.txt
```

This removes the execute permission bit (**x**) for the owner (**u**), group (**g**) and others (**o**).

So **chmod** alters the permission bits for files and directories. The left-hand side selects which users to apply the changes to (**u**, **g** or **o**)**,** the right hand side selects what permission bits to change (**r**, **w** or **x**)**,** and they are separated by either a **+** sign, to add the permission, or a **-** sign, to remove the permission.

Note that Windows does not have the same security model as Unix, so all permissions on files will be lost when you transfer them to a Windows computer. When copying files from a Windows computer to Unix, you will often find that all files have all the permission bits set for all users.

Test yourself

- Submit all the antonym files and all the synonym files to the *induct2006* course using turnin. Check that what you submitted is what you expected.
- How much disk space do you have left, in megabytes?
- Which directory or file is taking up most of the space in your home directory?
- Compress that file or directory with tar/gzip, tar/bzip2 and zip. Which method gives the best result (Hint: you could use either **du** or **ls -l** to check the file sizes)?
- Who owns your home directory? Does this mean you can change the permission bits on it?
- Create a new text file in your home directory. What permission bits does it have? Can another student read it? Ask your friend to try and open the file. Can they? Why or why not? (Hint: you may need to look at the permission bits for the directory it is in, as well as the file itself).
- Change the permission bits on the file so your friend cannot read it, if they could; or change them so they can, if they could not.
- Set appropriate permissions on your *courses* directory so that other students cannot read the files within it, or even see what files are in it. Get your friend to check for you that they cannot access it.

Creating a personal homepage

You can use your account on Yallara to set up a small personal website, for showing off previous projects, keeping a weblog or just telling the world a little bit about yourself.

First of all, create a directory under your home directory named ".HTMLinfo". Note that the capitalisation is important, as is the full stop (making it a hidden directory). Make this directory world-readable and executable (that is, set the **r** and **x** permission bits for all users **u**, **g** and **o**).

Now, make sure your home-directory is world-executable (it should not be readable, however, except for yourself).

Using **nedit** or another editor, create a file in that directory called "index.html". Make sure it's world-readable. Write a couple of lines of text introducing yourself and save the file.

If all has gone well, you should now be able to access your home page at http://yallara.cs.rmit.edu.au/~joebloggs

Please remember that your website is viewable by anyone, including people outside RMIT, and reread the appropriate sections of the Acceptable Usage Policy for guidelines on what you may and may not publish.

Printing

Unlike in Windows, where to print you simply select a printer from a list, in Unix you need to specify a complete program to print to, which in turn takes care of printing the document. For example, start Firefox and go to the File,Print menu.

When you click on the "Printer Properties" button you will see the command that gets executed. To print the web page normally, you would type in the **lpr** command here (type it in Firefox's print dialog, not in a terminal):

```
lpr -Pstudent1010
```

Note that you must use a capital **-P** in the option, and there must be no space between the **-P** and the name of the printer. In this case we have chosen the printer *student1010*, which is located in building 10, level 10, in room 10.10.24.

You can see a list of all the printers' names with the **lpstat** command:

```
lpstat -v
```

You can also use **lpstat** to see how busy a printer is (the more students printing to a printer, the longer your job will have to wait in the queue):

```
lpstat student1010
```

Note that most printers are unavailable to undergraduate students, for example those named "staff" or "research". Check with the duty programmers' desk if you need access to a particular type of printer.

You should try to save paper; both for the sake of the environment, and also because you have a limited quota of pages per semester. Using the **mpage** command you can shrink pages and fit 2 or more pages on each sheet. You would use this command in place of the **lpr** command in Firefox's print dialog.

```
mpage -4 -Pstudent1010
```

Specifying **-4** instructs mpage to fit 4 pages per sheet (they will be laid out in a grid). For most web pages or course notes, this is still quite legible.

You can cancel a print job with the **cancel** command:

```
cancel
```

This cancels all of your pending print jobs.

PDF, PostScript and image files

Many lecturers publish lecture notes and assignments as PDF files. On Windows you use Adobe Acrobat to read these files. On Unix, this is called **acroread**.

```
acroread /public/induction/coversheet.pdf
```

You can print PDF files from acroread through the File,Print menu and typing one of the **lpr** or **mpage** commands that you used in Firefox (see above).

Although they are less common nowadays, some documents are distributed in PostScript format (they typically have the extension **.ps**). The easiest way to deal with these files is to convert them to PDF files and then open them with acroread.

```
ps2pdf document.ps document.pdf
```

This would convert *document.ps* to *document.pdf*.

To view an image file (for example, in JPEG, GIF, PNG, BMP, EPS, TIFF, etc format), use the **xv** program:

```
xv image.tif
```

Right-click on the image to bring up a menu. Follow the same procedure for printing an image as with acroread and Firefox.

Starting programs in the background

Start **gvim** from a terminal. While gvim is still open, switch back to the terminal window and notice what happens when you type. You will find that you are unable to return to the prompt until gvim is closed. The same will happen with any program that opens its own window, such as Firefox, Acrobat Reader, xv, and so on.

You can stop this happening by starting the program in the "background". All this means is that as soon as the program has started, the terminal will return to the prompt, allowing you to continue using it while the windowed application is still open. Use an ampersand ("&") after a command to run it in the background:

```
acroread /public/induction/coversheet.pdf &
```

Try this and note how it is now very easy to open windows for many programs all from the same terminal. You will find this especially useful when you start programming, as you will often need to edit several files at once

End of Unix Survival Skills Part 1!

As mentioned earlier you need to learn how to edit using **vim.** You will find the Introduction to Vim on page 73 and now is a good time to get started.

More Unix

In the preceding section “Basic Unix,” you learnt the basics of how to deal with files and directories, how to open, edit and print documents, how to submit assignments or archive them, and some of the basics of Unix file permissions.

If you feel that you have a good grasp of the concepts in the previous section and wish to learn more, this section is for you. Some of the technicalities of Unix are explained in detail, a set of small programs that work well together are introduced and you will learn how to write simple scripts to automate repetitive tasks.

The real story

In the previous section we referred to the “terminal” as the program through which you enter commands. This is a bit of a simplification. In reality there are several programs interacting that you should be aware of.

The program which responds to keyboard input and displays the result in a window is **xterm**.

The actual program that does the work in interpreting your commands and acting upon them is called a *shell*. By default, your student account is set up to use a shell called **tcsh**. There are many other shells around; the other common ones are **bash**, **ksh** and **zsh**. All these shells have the same basic operation, in that you type the name of a program followed by its arguments and hit enter to run it. But a shell can do a lot more than that, as we will see.

If you login to Yallara or Numbat remotely via ssh you will be interacting with the same shell, but xterm will not be running (if you are using Linux then it probably is, but on Windows you are probably using Putty, and Mac OS X users typically use Terminal.app to perform the same function).

Each xterm you run starts another instance of the shell. Each of these instances is called a *process*. These shells have effectively nothing to do with one another once they are started. This is why changing your working directory in one window doesn’t affect any other windows.

Processes

Every program you start launches a new process. Typing **ps** will show you all the processes you have launched from the current terminal:

```
ps
   PID TTY          TIME CMD
 12388 pts/73       0:00 tcsh
 12392 pts/73       0:00 ps
```

The columns are:

PID	A unique *process identifier*.
TTY	The name of the terminal the process was started from.
TIME	How long the process has been using the CPU. Note that this does not include time spent waiting for user input.
CMD	The name of the program.

To see all the processes you are running on all terminals, use the **-u** option with your username:

```
ps –u joebloggs
```

To see all the processes that all users are running, use the **-e** option. Combining this with the **-f** option shows more details, including the name of the user running the process, when it was started, and the exact command line used to start the process.

```
ps –ef
```

A more useful way to see this information is with the **top** program.

```
top
```

This shows a constantly-updating list of the processes using the most CPU time, and who is running them. It also provides some statistics on the average CPU load and memory usage.

Signals and dealing with crashed applications

When you start working on programming assignments, there will come a time when you write a program that crashes, or hangs, and becomes completely unresponsive. How do you deal with it?

You can send a *signal* to a program while it's running with the **kill** command.

```
kill 12388
```

The number you specify is the program's process identifier, or PID, which you obtained using **ps**. By default **kill** will send the *TERM* signal, which requests that an application exit immediately. Normally this will close the application immediately, but it's possible for an application to crash and become unresponsive to even the TERM signal. In this case you can send it the KILL signal, which instructs the operating system to terminate it immediately:

```
kill –KILL 12388
```

If the program is running in the current terminal (as most of your own programs will), you can simply press Control+C. This actually sends the *SIGINT* signal, which is usually enough to terminate a crashed program. It can also be used to stop a program that is working fine, but is producing too much output for you to read. For example:

```
ls -R /
```

will recursively list everything in the file system. Running this to completion would take several minutes and almost all of it will scroll off screen. Hitting Control+C is the easiest way to stop the process and rethink your strategy.

For programs running in their own window, there is an even easier way to kill them:

```
xkill
```

xkill waits for you to click on a window with the mouse, then kills the process that owns that window.

Symbolic links

Did you get sick of typing */public/induction/* all the time? You can create a "shortcut" to this directory in your home directory with a *symbolic link*:

```
ln -s /public/induction ~/induct
```

List the contents of your home directory now. *induct* appears to be a regular directory, but if you **cd** into it and **pwd**, you will find yourself in */public/induction*.

List the contents of your home directory again, this time with the **–l** option:

```
ls -l
```

```
lrwxrwxrwx   1 joebloggs students        15 Feb 11 15:33
     /home/j/joebloggs/induct -> /public/induction/
```

Note the first letter in the permission bits is "l" for *link*. Also note how instead of just printing the filename, the link is shown, including its destination.

The **ln** command can create symbolic links (also called *symlinks* or *soft links*) to directories or files. The link only points to the destination path; if the target directory or file is moved somewhere else, the symbolic link will be broken and no longer work. Removing a symbolic link is done just as with any regular file:

```
rm ~/induct
```

This has no effect on the target directory.

Hard links

The **ln** command can also create *hard links*, which point to an actual file (they cannot be made on directories) instead of just the name. This has several consequences:

- If the target file is moved, the hard link will still work.
- If the target is deleted, the hard link will still work – the file won't actually be truly deleted until there are no more hard links pointing to it.

While this might sound like a great solution, in general hard links are not used as shortcuts. They cannot be used between different file systems (for example, between your home directory and the /tmp directory), and they can cause unpredictable results as it is easy to forget that a file is linked in two or more places.

To create a hard link, use **ln** without the **-s** argument. Their use is strongly discouraged however, and is presented here only so their consequences are known.

Shell quoting and escaping

When the shell (tcsh) interprets your command, one of the things it does is separate the command into the name of the program and a list of arguments. For example, the command

```
turnin -c induct2006 *.syn
```

is split into the program name "turnin", followed by the arguments "-c", "induct2006", "angry.syn", "confused.syn", and so on.

What if you have a filename with a space in it? For example, list the contents of /public/induction and look for the file named "My Assignment". Now try and copy it to your home directory.

```
cp /public/induction/My Assignment ~/
cp: cannot access /public/induction/My
cp: cannot access Assignment
```

The shell thinks you are trying to copy two separate files. You can put the name of the file in quotation marks to show that it is to be treated as one argument:

```
cp "/public/induction/My Assignment" ~/
```

Alternatively, you can *escape* the space with a backslash ("\"), which tells the shell not to treat it as an argument separator:

```
cp /public/induction/My\ Assignment ~/
```

You can also escape special characters such as *****, **?**, and even a backslash, so that they can be included within an argument without being interpreted by the shell.

Introduction to the scripting utilities

Several programs will now be introduced that appear to be quite simple and possibly useless on their own. These programs are present on every Unix-like system and can be used together to build up more complex programs, as you will see shortly. For now, we will run them directly from the shell so we can see what they do.

The **echo** command simply prints its arguments out:

```
echo Hello there
Hello there
```

What if you wanted echo to print the line "2*3=6"?

```
echo 2*3=6
echo: No match.
```

Remember that the shell expands the asterisk ("*") as a globbing operator, and tries to find files in the current directory that match. We must either quote the argument or escape the asterisk:

```
echo 2\*3=6
2*3=6
```

The **cat** command reads one or more text files and prints out the contents:

```
cat angry.syn
furious
indignant
irate
ireful
mad
wrathful
```

When run with more than one filename, cat will print their contents one after the other:

```
cat angry.syn happy.syn
furious
indignant
irate
ireful
mad
wrathful
fortunate
lucky
providential
```

Redirecting output to a file

Let's say you wanted a single file that contained all of the data in the synonym files. You could open each one and copy/paste the lines into a new document, but this could take some time.

```
cat *.syn > all.syn
```

The **>** operator instructs the shell to take the output of the command and write it to a file, instead of printing it to the terminal. Remember that **cat** will read all of the files in its argument list, which in this case is all the synonym files.

The output from **echo** can be treated the same way:

```
echo Hello there > greeting.txt
cat greeting.txt
Hello there
```

You can *append* (add to) a file by using two angle brackets:

```
echo Nice weather today >> greeting.txt
cat greeting.txt
Hello there
Nice weather today
```

Redirecting output to another program

The **sort** program reads lines of text and sorts them alphabetically. You can give it the name of a file to sort, or you can pass it lines directly from another program:

```
cat *.syn | sort
addled
addlepated
confounded
confusional
debilitated
decrepit
defeated
desolate
...
```

The *pipe* operator ("|", located above the enter key) is used to transfer the output from one program to the input of another program.

Remember the **grep** program for searching for text in files? If you don't give it the names of any files, it will search through the input you pass to it:

```
cat *.syn | grep ad
mad
addled
addlepated
muddle-headed
```

The real power in this is that you can then pass the output through another pipe to another program, building arbitrarily long pipes of commands:

```
cat *.syn | grep ad | sort
addled
addlepated
mad
muddle-headed
```

You can think of each command as a filter that processes data and sends it on to the next filter. When there are no more filters, the result is printed out to the terminal (or written to a file, if you choose to redirect it with **>**).

More scripting utilities

The **head** and **tail** commands show only the first or last few lines of a file, respectively.

```
cat angry.syn
furious
indignant
irate
ireful
mad
wrathful
cat angry.syn | head -3
furious
indignant
irate
cat angry.syn | tail -1
wrathful
```

uniq reads a file line-by-line and discards adjacent lines that are the same. For example, given a file containing:

```
one
two
two
three
three
three
```

Piping it through **uniq** would result in:

```
one
two
three
```

This is often used in conjunction with **sort** (so that identical lines are moved next to each other) to remove duplicate lines from a file.

wc counts lines, words and characters. By default it will show all three, respectively:

```
cat angry.syn | wc
      6       6      44
```

You can also show just the wordcount (**-w**):

```
cat angry.syn | wc -w
6
```

cut works with text data laid out in columns, and extracts one or more columns from the text. Consider the output of **ps**:

```
ps
   PID TTY          TIME CMD
 24618 pts/115    0:00 tcsh
 25773 pts/115    0:00 ps
```

First we note that the delimiter (separator) between the columns is a space character. Now see what happens when we extract the 2nd column:

```
ps | cut -d ' ' -f 2
25803
24618
25802
```

The **-d** option specifies the delimiter to use, in this case a space character. The **-f** option instructs **cut** to save just the 2nd column (it is the second in this case, not the first, because there are some spaces before the first column).

bc is a simple calculator. It evaluates any expressions that are passed into its input:

```
echo "2 + 3" | bc
5
```

Note how we used **echo** to create a line of input that we could pass to bc.

xargs takes lines of input and turns them into arguments for a command. For example, we can use **find** to return a list of files that match some criteria, and then pass that list to **grep** to search within those files:

```
find ~ -name \*.txt | xargs grep Hello
```

If you did not use xargs in this pipe, grep would have only been searching the filenames, rather than through the files themselves.

The **file** command tries to identify what sort of file a given filename is:

```
file /public/induction/coversheet.pdf
/public/induction/coversheet.pdf:  Adobe Portable
Document Format (PDF)  v1.3
```

Test yourself

Note that there are several ways to accomplish each of the following tasks. If you can't think of a solution immediately, try to break down the tasks into small pieces that can each be solved by one of the programs listed above.

- How many files are in the */usr/bin* directory?
- How many of the files in */usr/bin* are described by **file** as script files?
- Output just the last two lines of *happy.ant* to a file called *unhappy.txt*.
- Make a symbolic link to the file */public/induction/My Assignment* in your home directory called "assignment.txt". Make sure you can now display the contents of the file using the link.
- How many words appear in more than one antonym file?
- (Difficult) How many words in the synonym files also appear in the antonym files?

The backtick operator

We have seen how you can pass the output of one program to the input of another with a pipe. You can also pass the output directly as an argument to a program, without using xargs:

```
grep Hello `find ~ -name \*.txt`
```

The *backticks* ("`", located in the top-left corner of the keyboard) surround the **find** program and its arguments, then pass the output of that (a list of text files) to grep as its last argument.

You can even have pipes inside the backticks, which can make for some pretty complex expressions:

```
echo `cat *.ant | wc -l` - `cat *.syn | wc -l` | bc
```

To understand this, you need to take it piece by piece, starting with the backticks. The first set of backticks surround the command:

```
cat *.ant | wc -l
```

This reads all the antonym files and counts the number of lines in them. If you run this by itself the result is 53. Similarly, the second backtick expression does the same except with the synonym files. So after the backticks have been removed the remaining command is:

```
echo 53 - 39 | bc
```

which just passes the expression to **bc**, which in turn calculates the result. If you managed to solve the last "Test yourself" problem, see if you can now do it in just one line.

Writing shell scripts

A shell script is just a text file that contains a set of commands. When you run the shell script, all the commands in the file get executed almost exactly as if you had run them yourself.

Using **nedit** or another editor, create a file called "hello.sh" containing the two lines:

```
#!/bin/bash
echo Hello there.
```

The first line of the script is called the *hash-bang*, and it names the shell you would like to interpret your script. Although we use **tcsh** as our interactive shell, most people consider **bash** to be a better scripting shell, which is why we will use it here (although tcsh would work equally well in this case).

To run the script, you first need to make it executable. Do you remember how to set this permission bit?

```
chmod u+x hello.sh
```

We are choosing to set the execute bit (**x**) only for the owner (**u**) of the file, but you could also set it for group and others if you wanted to.

To run the script:

```
./hello.sh
Hello there.
```

Why did we need to use the "dot-slash" (**./**) at the start? To explain that, we need to delve into the world of *environment variables*...

Environment variables

Type the following command:

```
echo $PWD
/home/j/jbloggs
```

What has happened? Why didn't it print out "$PWD"? The dollar sign ("$") is interpreted by the shell as a *variable*. PWD is a variable that is always defined and always has the current working directory. You can set your own variables in tcsh with the **set** command:

```
set colour=red
echo $colour
red
```

You can see all the variables, and their current values, by typing just **set**:

```
set
...
autologout 60
colour     red
cwd        /home/j/jbloggs
...
```

These variables are used only by tcsh. However, some of these variables are mirrors of the *environment variables*, and these are used by all processes. To list all the environmnent variables:

```
setenv
USER=joebloggs
LOGNAME=joebloggs
HOME=/home/j/joebloggs
PATH=/usr/bin:/bin:/usr/local/bin
...
```

Programs such as Firefox use these variables to find your home directory, for example. The *PATH* environment variable is particularly important. This is a list of directories that contain programs. When you type a command like:

```
ls
```

the shell searches all the directories in PATH for the program named *ls*. You can find out where ls is actually located with the **which** program:

```
which ls
/usr/bin/ls
```

Going back to the script we wrote earlier, since *hello.sh* is not in one of the *PATH* directories, we need to specify its exact location:

```
/home/j/joebloggs/hello.sh
```

or just

```
./hello.sh
```

You might be thinking now that it would be a good idea to add "**.**" to the PATH environment variable, to save having to type the "./" in front of scripts you write. In fact, this is a bad idea, and there are good security reasons why "**.**" is specifically *not* in the PATH.

Back to writing shell scripts

Remember the **ps** command? Without any arguments it lists all the processes you are running on from current terminal. But to show the processes from all your terminals you needed to specify your username:

```
ps –u joebloggs
```

Wouldn't it be nice to have a shell script that does this for us? We'll start by creating a new file, "processes.sh", with the following contents:

```
#!/bin/bash
ps –u joebloggs
```

Set the execute permission bit and try and check that it works.

Now, what if your friend wanted a copy of your script to use for themselves? Obviously you would need to change the username, but that would mean having two or more copies of the same script, that do exactly the same thing. Here is a good candidate for using an environment variable we know about:

```
#!/bin/bash
ps –u $USER
```

Echo the value of $USER first to convince yourself it has the right username, then rewrite the script to make it useful for anybody.

Arguments to shell scripts

Now we'll write something a little more complicated. Remember the **kill** command? You have to specify a process identifier, or PID, to kill a program. Wouldn't it be nice to be able to specify just the name of the process?

We'll construct this step by step. First, we'll use the **ps** command to retrieve a list of all the running processes we own:

```
ps –u $USER
  PID TTY          TIME CMD
24618 pts/115     0:00 tcsh
25773 pts/115     0:00 ps
26110 pts/115     0:15 firefox
```

Next, we want to filter the list so it contains just the name of the program we're interested in. We can use **grep** to do this:

```
ps –u $USER | grep firefox
26110 pts/115     0:15 firefox
```

Now we need to extract just the process ID from this line. We have already seen how the **cut** command can do this:

```
ps –u $USER | grep firefox | cut –d ' ' –f 2
26110
```

Finally we need to pass this number to **kill**. Since **kill** only takes arguments, not standard input, we need to use either backticks or xargs. Either of the following would suffice:

```
ps –u $USER | grep firefox | cut –d ' ' –f 2 | xargs kill
kill `ps –u $USER | grep firefox | cut –d ' ' –f 2`
```

We are almost ready to put it in a script. We would like to make it general enough to be able to kill any process, not just Firefox. In a shell script we have access to the special variables **$1**, **$2**, and so on, which correspond to the arguments passed in on the command line.

In this case we only need the first argument, which takes the place of the word "firefox" in the grep command. The final shell script, which we will call "killall.sh", is:

```
#/bin/bash
kill `ps –u $USER | grep $1 | cut –d ' ' –f 2`
```

After you save and set the execute permission bits, you can test it out:

```
./killall.sh firefox
```

A teaser of scripting possibilities

The following script loops over all the arguments passed in and calls **dos2unix** on each one. The input and output filenames to dos2unix are the same, so it will do the conversion in-place. The -437 argument passed to dos2unix specifies that the files use US ASCII encoding, which suppresses the warning message about keyboard layout (see the dos2unix man page for details).

```
#!/bin/bash
for f in $*; do
    dos2unix -437 $f $f
done
```

If you save this as *d2u.sh* you can then convert a whole lot of DOS files to Unix format at once:

```
./d2u.sh *.txt
```

Where to go for more information

This manual represents just a sampler of the programs installed on Yallara and Numbat. Every single program presented has a myriad of options for customising how data is processed and formatted. Shell scripting in bash can be extremely flexible and goes far beyond the material presented here.

Any time you spend learning more about Unix will easily repay itself when you come to do assignments – especially using shell scripts to automate repetitive tasks. The following resources are good starting points for investigating the programs here that interest you.

Advanced Bash-Scripting Guide http://www.tldp.org/LDP/abs/html/

Easily the best online tutorial and reference for doing anything and everything with Bash.

Dave Raggett's Introduction to HTML http://www.w3.org/MarkUp/Guide/

If you would like to continue writing your web page, you will need to learn a little HTML.

Getting Started with awk http://www.cs.hmc.edu/qref/awk.html

Awk is a unique programming language for processing text files as collections of data records, making it easy to manipulate the data, create summaries and perform numerical computations on columns of text.

Sed – An Introduction and Tutorial http://www.grymoire.com/Unix/Sed.html

Sed is a program installed on all Unix systems. It is extremely useful for modifying text files with simple rules, for example, doing search-and-replace.

A Tutorial Introduction to GNU Emacs

http://www.lib.uchicago.edu/keith/tcl-course/emacs-tutorial.html

If you *really* don't like Vim (introduced in the next section), you may find Emacs and xemacs a better choice for you. They have a similar feature-set to Vim and gvim, respectively, but a completely different interface. There are not many people who can comfortably edit a file in *both* Emacs and Vim!

Unix man pages

As mentioned in the first section, the manual pages (accessible by typing "man <command>" are the definitive reference for everything currently installed. Some man pages are particularly comprehensive and worth at least skimming through:

tcsh: explains the syntax, configuration files and options.

ls: ls has many options built-in for formatting and sorting directory listings.

sort: sort can perform ascending and descending, alphanumeric and numeric, case sensitive or insensitive sorting on multiple columns of text.

fvwm: although not covered at all in this manual, there are literally hundreds of ways to customise the window manager, including setting up mouse and keyboard shortcuts, creating startup files, changing the desktop background, and so on.

Vim documentation

http://vimdoc.sourceforge.net/htmldoc/usr_toc.html
http://www.vim.org/tips/index.php

Vim and gvim are extremely flexible text editors used by the majority of computer science students at RMIT. The next section of this manual will give a very brief introduction, but to really take advantage of Vim you need to find the features that work best for you. The first URL above is the complete Vim documentation; the second is a series of over 1000 tips submitted by users which are particularly useful for seeing just what Vim can do.

Index of Unix Commands

pwd	Print the current working directory	37
quota	Shows how much of your allocated disk space you are using	47
rm	Remove (delete) a file or directory	42
sed	Stream editor, a program for manipulating lines in a text file	68
set	Set a variable in tcsh	64
setenv	Set an environment variable in tcsh	65
sort	Sort the lines in a text file	60
tail	Show just the specified number of lines from the end of a text file	61
tar	A program similar to gtar (always use gtar instead)	47
tcsh	The default shell used by students	
top	Shows and updates a list of processes that are the most active	56
turnin	An RMIT Computer Science program for submitting assignments	46
uniq	Remove duplicate adjacent lines from a text file	61
unix2dos	Convert Unix line endings to DOS line endings in a text file	46
unzip	Extract the contents of a zip archive file	48
vim	A powerful text editor	73
wc	Count the number of characters, words and lines in a text file	62
which	Determine which program will be run from the PATH	65
xargs	Executes a program with the standard input as its arguments	62
xkill	Lets you click on a window to kill its process	57
xterm	A simple terminal application	55
xv	View an image	53
zip	Create a zip archive file	48

Equivalent DOS and Unix commands

If you have used the Windows command prompt before, or a DOS shell, you may find the following table useful. Naturally the commands do not behave identically; you should check the Unix man pages for details on usage.

DOS	Unix / tcsh	Description
append	setenv PATH	Change the location of where to look for programs
at	at	Schedule a program for later execution
attrib	chmod	Change file attributes
call	source	Execute a list of commands in a file
cd, chdir	cd	Change working directory
cls	clear	Clear the screen
compact	gtar, zip	Compress files
copy	cp	Copy files and directories
date	date	Print current date and time
del	rm	Delete a file
deltree	rm -r	Delete a directory and everything in it
dir	ls	List the contents of a directory
diskcomp	diff	Compare files and directories
diskcopy	dd	Low-level file copy
echo	echo	Display a message
edit, edlin	vim, pico, emacs	Edit a file
exit	exit, logout	Exit the current shell
expand	gtar, unzip	Expand a compressed archive
fc	diff	Compare files
find	grep	Search for text within a file
ftp	ftp	Download files from an FTP server
help	man	View manual pages
md	mkdir	Create a new directory
mem	top	Display system memory usage
mkdir	mkdir	Create a new directory
more	more	Show a text file one page at a t
move	mv	Move a file or directory
path	setenv PATH	Change the PATH environment variable
popd	popd	Pop a directory off the pushd stack
print	lpr	Print a text file
prompt	setenv PROMPT	Set the shell prompt
pushd	pushd	Push a directory onto the pushd stack
rd	rmdir	Remove an empty directory
rename	mv	Rename (move) a file or directory
set	set, setenv	Change the value of a shell or environment variable

DOS	Unix / tcsh	Description
sort	sort	Sort lines of a text file
telnet	telnet	Log into another computer with encryption
time	date	Show current time of day
type	cat	Print the contents of a file
xcopy	cp -r	Copy a directory and its contents

Introduction to Vim

Vim is a powerful and flexible text editor used by users of Unix around the world. It can be quite tricky to learn at first, but once mastered it is an invaluable tool when editing configuration files, programming and writing reports. There are so many features of Vim that there is probably not a single person alive who knows them all. Everybody has a set of commands that they use themselves though, and you will need to find out which commands suit you the best.

Vim is an enhanced version of an older text editor called Vi. Both Vim and Vi are text-only; they run directly in the terminal window (also making the suitable for use over an ssh connection).

We recommend using the graphical Vim, **gvim**, while in the labs. It is exactly the same program, except it runs in a separate window and has mouse support. You can follow this manual using either Vim, Vi, or gvim. Note that gvim is also available for Windows and Mac OS X (see http://www.vim.org), and Vim is already available on every installation of Mac OS X.

Command mode editing

Unlike other text editors you have used, Vim has two modes: *command* and *create*. You can only type text while in a *create* mode. While in *command* mode you can load and save files, do searches and replacements, import other text files, and so on.

To start Vim:

```
vim file1.txt
```

If you are in a computer science lab, you will probably prefer gvim:

```
gvim file1.txt &
```

Remember from the Unix tutorial that the ampersand ("&") is used so the terminal is still available after starting gvim.

When Vim or gvim starts you will be in command mode with a blank file. To start typing, press **i** (for "insert"). Type a few sentences of nonsense, then return to command mode by pressing ESC (the escape key). In summary:

- Type **i** to enter *insert* mode, where you can type text.
- Press ESC to leave any create mode and enter command mode.

Saving and exiting

To save the file you are working on, in command mode type **:w** (that's a colon, followed by a lower-case **w** for "write") and hit enter.

To quit, in command mode type **:q**. You can actually save and quit in one smooth move by typing **:wq**.

If you try quitting without first saving, Vim will show a warning that the file is not saved. You can override this with **:q!**, which means "quit, don't save, I know what I'm doing."

Help me, what's going on?

Sometimes a simple typo can make you feel hopelessly lost in Vim as it activates a feature you have never heard of. In almost all cases, you can simply return to command mode by hitting ESC two or three times. Similarly, you can undo the last command or insertion by hitting **u** (for "undo").

More creation modes

You have seen **i**, which starts insertion mode wherever the cursor is. There are a couple of shortcut keys for starting insertion in different places relative to the cursor:

- **a** begins inserting text just after the cursor.
- **A** begins inserting text at the end of the line
- **I** (capital i) begins inserting text at the beginning of the current line.
- **o** "opens" a new paragraph below the current line.
- **O** (capital o) "opens" a new paragraph above the current line.

Try each of these now to get a feel for how they work; they can save a lot of time that in any other editor would be spent moving the cursor.

Moving the cursor

Speaking of moving the cursor, Vim has a hundred ways of letting you move the cursor around. Here are just a few (note that with the exception of the first two you need to be in command mode):

- If you are using gvim, you can of course point and click where you want the cursor.
- You can use the arrow keys as with any other text editor to move the cursor around.
- The **w** and **b** keys move forward and back one word, respectively.
- The **(** and **)** keys move to the previous or next sentence.
- The **{** and **}** keys move to the previous or next paragraph.
- The **0** (zero) and **$** keys move the the start and end of the current line.
- Press **Control+U** and **Control+D** to scroll up and down half a page at a time.
- The **Page-Up** and **Page-Down** keys scroll up and down a whole page at a time.
- You can move to a specific line-number by typing the number in and pressing **G**. For example, to go directly to line 48 you would type **4 8 G**. This becomes very useful as you start programming as most errors are reported with the line number on which the error was found.

Moving more

All of the above movement commands can be extended by specifying a number of times to repeat the command. For example, typing **5 w** moves the cursor forward 5 words. Typing **3** and then pressing the down arrow moves down three lines.

Deleting text

While typing text in a create mode you can delete straight away as usual with the delete and backspace keys. In command mode you have a few more options:

- Press **x** to delete the character the cursor is highlighting.
- Press **d d** (you press it twice) to delete the current line.
- Press **d** and one of the movement keys listed above to delete that amount. For example, typing **d w** deletes one word. Typing **d 3)** deletes three sentences.
- Make a selection with the mouse and press **d**.

Pasting text

Vim does not have separate “delete” and “cut” commands. After you delete something, it is immediately available to be pasted.

- Press **p** to paste text just after the cursor.
- Press **P** to paste text just before the cursor.
- An easy way to fix those typos where you swap two letters around (e.g. in “teh”) is to position the cursor at the first of the pair of letters and press **x p** in succession.
- An easy way to swap two lines of text is to press **d d p** in succession.

Copying text

Vim calls copying “yanking”. After you have “yanked” some text you can paste it with one of the commands above.

- To yank (copy) the current line, press **y y** (press **y** twice).
- As with the delete command, you can use **y** with a movement. For example, **y** and the down key copies two lines; **y 2 }** copies two paragraphs.
- Make a selection with the mouse and press **y** to copy.

Replacing text

These are shortcuts to deleting text and then inserting new text.

- Press **r** to replace a single character. For example, pressing **r b** replaces the character under the cursor with a “b”, and then returns to command mode.
- Press **R** to replace a lot of text. This enters a create mode where every letter you type overwrites the existing text instead of inserting.
- Press **c** and a movement to change some text. For example, **c w** deletes one word and then enters insert mode, perfecting for changing just that word.
- Make a selection with the mouse and press **c** to delete the whole selection and enter insert mode.

Indenting text

When you start programming you will find that having well-indented source code is invaluable (both for readability and getting reasonable marks on your assignments!).

- To indent the current line, press **> >** (the right angle bracket twice).
- To unindent the current line, press **< <**.
- You can use **>** or **<** with a movement. For example **< }** will unindent the whole paragraph.
- Make a selection with the mouse and press **<** or **>**.

Searching text

Check that you are in command mode first (press ESC).

- Press / (forward-slash) and type the word or phrase to search for and press enter. For example to search for "checker" you would type /**checker** and hit enter.
- To move to the next search result press **n**.
- To move to the previous search result press **N**.
- Note that searches can contain complex regular-expressions. If you don't know what a regular expression is yet, make some time to find a tutorial. You will love them!
- You can search for occurences of the word the cursor is in simply by pressing * (asterisk).

Replacing text

Usually you will want to do a text replacement over an entire file. In command mode, type:

```
:%s/original/replacement/g
```

and hit enter. An explanation of this command follows:

- The colon is used before most complex commands consisting of more than one or two characters.
- The percent sign ("%") signifies that the command is to act over the entire file. There are ways (not discussed here) of restricting it to just a section of the file.
- The **s** stands for "substitute" and is borrowed from **sed**'s language (see the references at the end of the Unix section of this manual).
- The / (forward slashes) separate the original text that you are searching for with the replacement text.
- The **g** option at the end stands for "global", and instructs Vim to make the replacement for every occurrence of the original text on each line. Without this flag, only the first occurrence on each line is be changed.
- The original text can be a regular expression, and the replacement text can contain back-references into that expression, making for a very flexible substitution scheme. See the Vim manual for examples on usage.

Getting help

Vim's online help is very comprehensive, sometimes a little *too* comprehensive. You can access the table of contents by typing:

```
:help
```

or search for a particular feature (say, the substitute command described above):

```
:help :s
```

Close the help window with:

```
:q
```

Vim configuration

You can create a Vim configuration file to store your preferences in your home directory. The file should be called **.vimrc** (note the full-stop at the start, making it a hidden file). Create this file now with the following contents:

```
syntax on
set autoindent
set ts=4 sw=4 et si
set whichwrap+=<,>,[,]
set backspace=indent,eol,start
set hlsearch
```

These options turn on some nice user-interface features, such as allowing the cursor to move anywhere, highlighting search results, replacing tabs with spaces, automatically indenting source code sensibly, and highlighting source code according to its programming language.

Congratulations!

If you have made it this far you are well on your way to becoming a Vim grand master. Remember that the commands introduced here represent less than 1% of Vim's functionality. Browse through the tips at http://www.vim.org for ways other people are using Vim.

Some offences that will result in the suspension of your account include, but are not limited to:

- Eating or Drinking within the computer laboratories.
- Having a drink bottle on the desk whilst at a terminal/workstation.
- Locking of terminal/workstation (even whilst on a toilet break).
- Use of a non-RMIT web based email service.
- Bypassing of Internet proxies.
- Changing the hardware or software configuration of the terminal/workstation.
- Use of an instant messaging/chat program.
- Use of a peer-to-peer file sharing application.
- Broadcasting of messages to computers and/or system.
- Account sharing.
- Noisy or disruptive behaviour within the computer laboratories.
- Use of mobile phones in computer laboratories.
- Failing to follow RMIT Staff instructions or directions.
- Print queue jumping.
- Viewing, Displaying or Downloading of offensive or illegal material.
- Streaming of Web Radio or Video that is not course related.
- Playing of Games (unless required by course).
- Allowing people into computer laboratories after hours that do not have a valid access card.
- Being in the computer laboratory after hours and not having a valid access card.
- Sleeping in computer laboratories.
- Making inappropriate posts to course/program newsgroups (i.e. for sale items).
- Changing headers in posts to newsgroups to hide identity.

Some offences or repeat offenders will have the matter referred to the Technical Services Manager and/or Program Leader for further action.