Introduction To Unix Shell Accounts at Lancaster University

Tabitha Tipper
unixintro@lancaster.ac.uk

March 13, 2017

WARNING: While the following document makes every effort to be factually correct some errors may be present, this document was only written by humans. Be sure to understand what you're typing before you press enter as no responsibility will be taken for incorrect information or loss of data caused by use or reference to this document.

Contents

1 Welcome to the Unix System at Lancaster (unix.lancs.ac.uk)

1.1 What this document tells you

This document is designed to help a new student, of any level of computer literacy, get familiar enough with the Unix operating system to be productive with it. This includes file management, communication (primarily through the use of email) and also some background information to just familiarise themselves with other interesting and helpful parts of the system. It is not designed as a complete reference, or a command reference, and parts of it are specific to Lancaster University's central Unix server as opposed to being generic.

While there are Graphical User Interfaces (GUIs) available for Unix, and available for use at Lancaster this document concentrates entirely on "shell accounts" or in other words logins that are text-only, this is because these will be far the more common method of accessing the Unix systems and can be done from anywhere.

1.2 What is the point of the Unix systems

The Unix system, also known as cent1, is provided by the University of Lancaster to help its students get their work done. It is a powerful, fast, centralised computer system that's always available. This means that wherever you are, and whatever time it is you can always get to cent1 to run programs on it, or to get your files from it. It's never switched off, or locked away, and it doesn't matter if you're in a lab, or a campus room or a room off campus, once connected to cent1 then you can do whatever you could from anywhere else.

The range of uses for the Unix systems are wide, you can use them for email, word-processing, software development, web browsing, reading usenet, the maths students use it to run matlab and the computer science department use it to teach Scheme and Java.

1.3 Primer on the concepts involved

Before I progress any further in this document I'm going to write a quick section giving an outline of the differences between using a shell account and using a Windows machine in a lab or your room.

When you connect to the Unix systems at Lancaster you will get a load of text appear in the window, then you'll see a prompt, essentially you type a command and press enter and that command is run. Once it finishes you see your prompt again and the system is ready to accept another typed command. Anyone who has used MS DOS, or the command prompt under Windows should be familiar with this method of interaction.

The above paragraph shows you how simple using a shell account is, however there are a few concepts to think about, primarily you must remember that although the window that has all the text in appears on your machine all the programs you're running are being run on another computer. This means that you cannot directly access or affect the files on the machine that you are sat in front of with the commands typed into a shell.

While this may seem counter productive to be unable to directly affect those files, later on its shown how easy it is to move files between the Unix server and your machine (see section 4.10) also your shell on the Unix server will have its own files and directories to affect.

Also its worth remembering that while the machine you are logged into the Unix systems is generally only being used by you the Unix servers are being used by many users just like you, all at the same time, all with their shells running commands for them.

1.4 About this document

This document is a guide, and is designed to cover as many different topics as possible, in as much detail as is required to give the reader a fairly good grounding in the topic, as well as some pointers on where to find more information. If you skim over the index it should be fairly clear what each area covers and you should be able to jump to the correct part of the document. While it would help to read the thing from end-to-end I know that most people will blanch at this, the topics are fairly self-contained but reading some of the later ones without having at least skimmed the earlier topics will probably leave you fairly confused.

It should be available either via the web (in which case you're reading this in your browser) or as a giant PDF file (in which case you're reading this in either xpdf or Adobe acrobat).

Its usually available from:
http://www.lancs.ac.uk/ tipper/luui/unix-intro.pdf

Or in HTML form from:
http://www.lancs.ac.uk/ tipper/luui/html/

More information about it can be obtained from:
http://www.lanc.ac.uk/ tipper/luui/

Suggestions and corrections should be mailed to:
unixintro at lancaster.ac.uk

It was designed to be written in a fairly conversational and relaxed style and some of the terms which might puzzle you can be found in:
http://www.catb.org/jargon/

However it is also a technical reference and so precision is important. It should be noted that example commands for you to type in are shown like this:

```
% ls
```

Here the `%` (percent) symbol represents the prompt where you type, you don't type `%` before your command. "`ls`" is the command being used and that is all you type, then you press enter to run the command.

It should be noted that you'll need to be careful about typing your commands. They are all case sensitive, which means the commands "`ls`" and "`LS`" would be two different commands.

Since most programs have names all in lowercase, and this case is important, for clarity these programs will *always* be written in lowercase. Even if the name of the program starts a sentence or title, it will be written all in lowercase.

As a last point about programs is that all program names can be verbed, for example you may find such sentences as "frank grepped for X" instead of "frank searched the output using the grep program for string X.".

The other thing is that you will see a few sections that include notation like:

Then press `^d`

What this means is that you need to hold down the control key (It should be in the bottom left corner of your keyboard, it will probably have the letters "ctrl" on it) and while you are holding that down press the key listed after the `^` symbol as well.

In this case you would hold down control and press the 'd' key as well, then release both of them. You'll get used to doing this for several things. It should be noted (especially in light of the next segment) that while you will often see things writte `^D` the program will in 99% of cases accept a lowercase or uppercase `d`, just as long as you're holding down control.

Things that need to be typed exactly, and often the name of programs to distinguish them from the text around them will also be shown in the `teletype` font, these are things you should type exactly as you see them written, and this font is used for clarity. These things may well also be enclosed in " and " symbols.

After an example is given you will often see things like:
where "username" is the username you are interested in.

Here is is assumed that you will substitute the word username for someones actual username, or some other piece of relevant information, such words should *not* be in `teletype` font, to indicate you have to change them to something else.

Also you may see sections of the text discussing "arguments", "flags" or "options" to commands. An argument is a word that follows a command. For example:

```
% vim foofile
```

This runs the program vim with the argument “foofile”, generally this is used to give a list of files for a program to affect. In the example above it will open the file called “foofile” in the program vim and allow you to edit it.

A flag or option is a method of changing how a program runs, for example:

```
% ps
```

This will run the command ps in its normal mode of behaviour, however if you wanted to see more information you would use the following command:

```
% ps -f
```

Here -f is the “full information” flag, most of the time flags are shown by having a `-` (dash) symbol in front of them, and no space before the letter.

Quite often you’ll notice there are two sets of flags for a program, there are the short options with a single dash followed by a single letter (or perhaps two letters) and the long options, which are shown by two dashes followed without a space by a whole word. The short options are there to save typing, the long ones for clarity.

Information on how to get the lists of flags for a program is in section 3.5.

1.5 About the author

The author of the document has been a student at Lancaster University all through his undergraduate years as a Computer Scientist, and is a keen tinkerer with Unix, as well as an active member of LuBBs (Lancaster University Bulletin Board System) and so has quite a bit of exposure to the Unix systems here.

1.6 Acknowledgements

This document couldn’t have been completed without encouragement and editorial suggestions from the LuBBs community (see 13.11). Also in particular the following (in alphabetical order):
:DAN)
AZIRAPHALE
CARROT
HEDGEHOG
HOBNOB
STEREO
TERMINAL ADDICT

And surely others I’ve forgotten.

2 User Accounts

2.1 What? Why? What can it do?

Simply put a "user account" is a username that a computer knows a user by. All the programs you run on the Unix systems will run "as your user". Your user account is linked to your identity and actions taken by it are presumed to be directed by you. Logging onto the Unix systems and thus using your user account is covered in another document:
http://www.lancs.ac.uk/iss/docs/documents/unixlogon.pdf

The short version is you'll need a copy of PuTTY which should be found in the Start menu of your lab machine, or a machine with a native SSH (Secure SHell) client, such as another Unix machine. Simply tell your SSH client to connect to "`unix.lancs.ac.uk`"

If you are connecting from a Windows machine off-campus then you can get a copy of PuTTY from:
http://www.chiark.greenend.org.uk/ sgtatham/putty/download.html
This will allow you to securely connect to unix.lancs.ac.uk from anywhere in the world pretty much.

If you are sat at a Unix machine either on or off campus then you should be able to connect by typing:

```
ssh username@unix.lancs.ac.uk
```

Where "username" is your username. If your machine doesn't have ssh then contact your Sysadmin, Vendor or visit:
http://www.openssh.org

Anyway to get back to the discussion about user accounts, what this means is that for the purposes of this documentation you are your account. If directed to do something then you will type the commands, and they will be executed as your username, you will be held responsible for actions taken by your account.

Now that's out of the way we move onto the point about why the system has many different usernames. A Unix system is not designed to be like a Windows system, they are designed from the ground up to deal with having many different users, all running programs at once, thousands of people could be logged into the central Unix server at Lancaster at any one time. As such each user must have a unique name, otherwise sharing out the resources and securing the system becomes a much harder challenge. With your username you have a certain amount of quota in terms of CPU time and disk space on various machines around campus. The programs you run will run as you, and your files will be owned by your username.

If you want to know which username you are logged in as to a Unix machine the following command typed at the prompt should tell you:

```
% who am i
```

This command will print out your username, followed by the following information (separated by spaces) the "terminal" you are connected to (see section 6), the date and time you logged on, and the domain name of the machine you logged in from.

If you want to find out more about the technical information behind your user account the following command will give you your username's UID (User Identification Number) and your GID (Group Identification Number).

```
% id
```

It will print your UID, followed by your username (in brackets), then your GID and your group (again in brackets). All you really need to know about these numbers is that they're the way the system actually tracks you, and it simply maps the UID to your username.

2.2 Home directories (∼)

Each user has what's called a "home directory", this is the directory that you can store all your files in, the directory is owned by you and in it you can do whatever you want.

The home directory is also known by the character ~. So if you see a path (think location) of a file listed as "~/documents/foo" you can tell it's in your home directory, in a subdirectory called "documents" and the file is called "foo".

Similarly other user's home directories can always be found by using "~username" as part of the path to something. So "~frank/documents/a-file" will access the file called "a-file" in the subdirectory "documents" of the home directory of the user called "frank" (assuming you have permission to). This makes life easier as you don't need to remember where another users home directory is, just their username.

2.3 Details

Your username is more than just a short name, your user account contains a few details which are publicly viewable by any other user on the system. To view the details of a username you need to use the "finger" program. So to view the details of your own account you type:

```
% finger -l username
```

Where "username" is your username. finger gets you the information, and using the "`-l`" option (long output) before the username you want to know about will get you more information. This should get you output on the username, the real name of that user, their home directory, when they last logged onto cent1 and the machine they last logged in from, if they have mail in their inbox and finally the users "Plan". This is some text that is read from the file called "~/.plan" meaning the file called ".plan" in that users home directory. This is designed so the user can add information about what they're currently working on and how to contact them and make it easily available to the other users of the system.

2.4 groups

Of course in any organisation which would be using many usernames listing all the users who can access any one file could get very silly, imagine listing all the usernames of everyone on your course who could get to a certain file? Such a thing is wasteful as massive lists would need to be maintained for every file on the system.

To get around this problem each username is also associated with a group. Groups of users can be given permission to do something on the system, thus meaning all the users in that group can do it automatically too. Usernames may have more than one group, but always have a "primary" group that is considered their main group.

To tell what groups a username is linked to use:

```
% groups username
```

Where username is the username of the user you're interested in. This will give you a series of short code names for each group that user is in. The first one on the Lancaster systems will usually be either ug (undergrad), pg (postgrad) or st (staff) denoting the type of user account. Then a group for their department e.g. "cs" for the computer science group.

2.5 Passwords and why they matter

The central Unix server at Lancaster is accessible over the Internet from anywhere for remote logins. What this means is that anyone, anywhere in the world can log in *as you*, read your email and files, store illegal files in your accounts home directory, run your programs, send vast amounts of spam or even attack other machines in a way that will make it look like it was you. The only thing that really stands between your account and them is your password, and for this reason its vital that you pick a good strong password that's hard to guess and change it regularly in case it has been discovered.

Remember, even if you don't have anything in your account you think is worth someone breaking into it to look at (e.g. your files or email) think of the other people out there who do, and remember that hijacked accounts are one of the safest ways for crackers to launch attacks at places that really matter, or to break into more accounts, and eventually they'll find something worth stealing.

It's also worth always logging into the central Unix servers using a program like "PuTTY" as this uses a secure protocol. If you use something like "telnet" then other machines on the network can sniff your password as it travels past them, and hijack your account, so remember, always if possible use a program like "PuTTY".

Now that I've hopefully told you the absolute worst that can happen to your account you'll probably want to know about how to change your password and how to pick a good one. There are plenty of good guides for picking strong passwords out there, but ISS offer:
http://www.lancs.ac.uk/iss/security/passwords/
as a solution. Every time you load the page you will be offered a selection of fairly good passwords, simply pick one you think you'll remember and use it.

To change the password of your Unix account you need to use the passwd command (note the missing o and r):

```
% passwd
```

This will prompt you for your current password (to stop people from just changing your password if you leave your desk unattended) then will ask for the new password twice to make sure it's correct. You can do this any time you like, and generally it's advised you change your password at least once every few months.

For more information on local issues for password changing see section 13.3.

2.6 Finding out other peoples usernames

On the Lancaster central Unix servers there is a program for finding out others usernames based on their surnames. To do this you use:

```
% whois surname
```

Where surname is the surname of the person you want to find. The program will then print the full names of everyone, followed by their username. For common surnames this can be a long list, but can prove a helpful program.

2.7 Seeing who else is logged on and what they're doing

There are two commands that can accomplish this, if you want to see all the other users that are currently connected to cent1 simply use the following command:

```
% who
```

This will simply print out a list of all the users connected (just their usernames however, not their real names), the terminal they're connected to, when they connected and where they connected from, much like the "who am i" example (see section 2.1).

However to quickly see roughly what the other users of the system are up to use the following command:

```
% w
```

This will show a top line which contains the date, how long the systems been running, how many users are on and whats called the “load average” which is how many processes are contending for the CPU. This is shown as three numbers: the first is for 1 minute ago, the second 5 minutes ago, the third 15. All you really need to know is that small numbers are good here.

After this w will show a header that says what the columns are for, username, tty, when that user logged in, how long they’ve been idle, the JCPU and the PCPU and finally what the process is. JCPU is the total CPU time used by all the processes attached to that terminal, the PCPU time is just for the current process running in the foreground. Quite often these two numbers will be the same. Knowing exactly what these numbers mean or how to use them isn’t really essential knowledge, but can prove useful or interesting.

Section 7 and specifically section 7.4 will cover other ways of finding out information about other users on the system, with more flexibility.

2.8 Finding out who’s been logging in when

The last command that I’ll be covering in this brief section on users and accounts is the command used to print out a list of the logins and logouts that are stored in the system. So to find this type the following:

```
% last
```

That prints the following columns: login name, terminal, where they logged in from and how long they were logged on for. However this list can be exceptionally long so its often best to use the -n option for it, which limits how many it prints out, for example:

```
% last -n 20
```

Will print out as above, but only the last 20 logins.

3 Getting Help

One of the more important things this document can teach you is how to find out more information, and more importantly useful information, for yourself. That way when presented with a problem you will hopefully be able to find what you need to solve it.

3.1 man

The primary help system on Unix systems is the so called "man pages". Short for "manual pages" these documents literally form the entire manual for the system, and traditionally were written by the programmer who wrote the associated program (although professional technical writers are often employed by companies these days). They have a reputation for being useful to getting the problem solved, if, and only if, you can get through them. They have been called terse, juvenile, unhelpful and a host of other insults by users down the years but are still one of the most valuable repositories of Unix knowledge.

Really they aren't that bad these days, but they can be quite dry reading. To access the manual for any program on your system just type:

```
% man program
```

Where program is the name of the program, e.g. "pwd". The manual for that program will then be opened, usually you'll can press the space bar for the next page, and "`b`" to go back a page. When you want to stop reading simply press "`q`". Good man pages to read include: "intro", "man", "less" and "ls", as well as any command used in an example above that you want to know more about.

3.2 info

Sometimes the man pages for commands will say something to the effect of "This is just a basic primer, the full documentation is part of the info-file system". Essentially info is a program very similar to man, as far as the user is concerned. In situations like this you should be able to press "`q`" to drop out of man and then just type:

```
% info program
```

Where upon you should have the documentation for the program, again the space bar will move forwards a page, the "`b`" key should move backwards and "`q`" should quit. For more information see "`info info`".

3.3 apropos

The man and info systems are all very well when you know what the programs you need to use are, but finding those programs if you don't know them means a little more digging about. "`man intro`" is a good place to start for that as it lists most of the commands, but searching through it for what you want can be tiresome. And for that purpose another command was invented:

```
% apropos keyword
```

apropos will search the system for documentation that relates to the "keyword" given to it as an argument. So for example "`apropos network`" will list most of the programs that deal with networking on the system, "`apropos files`" lists most of the file management or manipulation programs. This makes apropos an invaluable tool in finding the programs you need to use to get the job done.

3.4 whatis

whatis is essentially the opposite of apropos, assuming you have a program name then whatis will give you a brief summery of what that program does. Its usage is fairly simple, e.g.

```
% whatis ls
```

Will tell you that "ls" is used to "list the contents of a directory".

3.5 -help (dash help) and –help (dash dash help)

One of the final methods of getting help about a program is to use the program's inbuilt help system. This is also a good way of finding out all the short and long flags which change how the program runs. Be warned, a lot of programs will give you the same message for both of these options, and you will have to hunt amid the short and long arguments for what you want.

Simply using the `-help` (dash help) option to get a listing of all the short flags for that program. Some programs prefer `-h` to list their short options, these programs will generally complain about 'e', 'l' and 'p' being bad options, these messages can generally be ignored.

Use `--help` (dash dash help) to get a list of all the long options for that program.

These also tend to provide a short message describing the program, nothing as long as the man pages, but a brief summary. Its worth noting that some programs don't accept one of these two arguments and will instead run with their default options. This can be dangerous so its usually best to read the man pages first.

An example of this would be the "tar" program:

```
% tar -help
```

This command (which is "tar, space, dash help") outputs an error message and says "Try 'tar –help' for more information." so as you can see although it doesn't support the option no harm is done by using `-help`, so to get the full set of flags use:

```
% tar --help
```

This will actually output a full list of flags, both short options (like `-R`) and long options (like `--no-recursion`).

3.6 The Library

C floor of the Library contains the computing section. In there you will find a number of introductory and in-depth references for Unix operating systems. Borrowing one of these, or even getting it out for a quick skim could help you immensely.

3.7 Google / Linux

Google, or more specifically any search engine on the Internet, is a good place to dig up a lot of information of varying levels of quality. With the recent rise of Linux and the free *nix's Google has become a rather good problem solving tool for working with Unix systems.

Often the best way is just to offer the main http://www.google.com site a few keywords that seem related, such as the names of programs you're using and what you're trying to do. If this doesn't get you any good results then searching http://groups.google.com (which is a comprehensive Usenet archive) might well get you the answer.

Beware advice gained in this way, the populace of the Internet isn't composed of 100% Unix gurus and mistakes may be involved in the advice, always try and understand exactly what you're doing before you do it.

3.8 LuBBs (http://www.lancs.ac.uk/socs/lubbs)

The Lancaster University Bulletin Board System is covered elsewhere in this document (see section 13.11) and it contains a fairly computer-literate slice of the population. If you can't find the information you need within the above steps then trying LuBBs can be a good next step.

4 File Management

One of the more important topics for the use of your Unix account is file management, the understanding of how to manipulate the various files you have in your account. Since you will largely be connecting to cent1 via a text only console you will need to understand how all the typed commands available affect your files.

First off its worth noting a couple of points about files under Unix. Most importantly they are all stored in a case sensitive way. This means the file "foo" is not the same as "FOO" or the file "fOo". For this reason clear naming is important. The second is that file extensions do NOT have to relate to what the file is for. Whereas on a windows box you will find for example MS word documents ending in .doc, MS Excel spreadsheets ending in .xls and executable programs ending in .exe on a Unix system this is not the case. Any file may be called any thing, and any program may read any file happily. While most programs will use a file extension they are just part of the file name and the Operating System doesn't treat them as instructions on how to treat that file (this is a good thing for security).

Files are generally assumed to contain one of two things, either binary data (e.g. a program designed to be executed) or textual data (often formatted in some specific way). If you're not sure what a file contains then its best to use a very simple command that can tell you:

```
% file filename
```

The "`file`" program will tell you what type of file the file called "filename" is. It does this by examining the start of it, checking for a file extension (they're optional, but often used) and a few other tricks. `file` will tell you if something is a file or directory, and if its a file will probably tell you what type, and thus what program, should be used to open it.

4.1 Configuration files

Another convention which it helps to know about for files is the dot-convention. Any file or directory on a Unix system that starts with a dot/period/fullstop e.g. ".vimrc" is not shown during normal browsing. These files are generally assumed to be configuration files for your programs, and so not something you want to see cluttering up your view all the time. However these files can be accessed just like normal, they aren't considered "hidden" for security reasons, just out of your view for convenience.

4.2 Basic Commands

There are a handful of basic commands that can be used to manipulate your files on the system, and I'll now go through them, with examples.

4.2.1 Your current directory

The first thing to know how to do is to see where you "are" in the directory structure. Your shell will always be essentially "looking" at one directory in the structure at a time, this is known as the "current directory". So the first thing to do is type:

```
% pwd
```

"pwd" (Print Working Directory) will simply print out the current directory you are in, usually if you've just logged onto cent1 this will be your home directory.

4.2.2 Finding out what files there are

Now that you know where you are you probably want to know what files are there. This is fairly easy to achieve:

```
% ls
```

ls (LiSt files) is used to get a listing of all the files that are currently in the directory you're in. Well almost all the files, all the dot-files (see above) will not be shown by a regular ls. To see all the files, including these dot-files you need the following command:

```
% ls -a
```

Other useful features of ls include the -F flag. This is used to help differentiate many common types of file. So:

```
% ls -F
```

Shows all files a normal ls would. Except directories have "/" appended to the end, executable files have a "*" at the end and symbolic links (see section 12.3) have a "@" at the end.

These two flags can be combined together to show this behaviour for all the dot-files, simply by saying either "`ls -a -F`" or "`ls -aF`", both are valid.

4.2.3 Viewing a file

Now that you can see what files are in your current directory you might well what to know what they contain. All files can be viewed by any program, but probably the easiest and safest program for viewing them is "less". Less is a type of program called a "pager" and is designed to view files which have more lines than your terminal is tall. Its use is fairly simple:

```
% less filename
```

Here less will open the file called "filename" for viewing. less is probably what was used when you were viewing man pages (see section 3.1) and so its use should be the same. Simply press the space bar for the next page, and "`b`" to go back up a page, once you're done pressing "`q`" will quit. less can do a lot more than many people think, including searching for text, which can be achieved by pressing "`/`" and typing the word you want to search for, then pressing return. To see a full description of how to use less see "`man less`" or "`less --help`" (dash dash help).

4.2.4 Copying a file

You can make a copy of any file you have permission to read (see section 4.8). This simply creates a whole separate filename with exactly the same contents as the previous one and is often used to make backups of files. For this you will use the cp command:

```
% cp original target
```

cp (CoPy) will simply copy "original" to "target", if there is already another file called "target" it will overwrite it, so be careful. If you want to try and prevent that from happening use:

```
% cp -i original target
```

With the addition of the `-i` flag cp will prompt you with a yes/no prompt if "target" already exists, so you can be sure you're not overwriting a file you might want.

More details with how this relates to directories is shown in section 4.3.4. Both the "original" and "target" names conform to all the rules laid out in section 4.5, so you can read that if you want more information on clever things that can be done with them beyond basic copying.

4.2.5 Moving or Renaming a file

Where copying is to make a duplicate of a file moving is much simpler. The move file command simply moves the file from one location and name to another. This means that the move command can also be used to rename a file, as you simply move it from one name to another. It can be used as follows:

```
% mv original target
```

mv (MoVe) will move the file called "original" to the file called "target", simple as that, all its permissions will be the same, and even the date it was created will be the same, only the name (and possibly location) will change. mv also supports the `-i` flag just like cp does, which again helps if you want to make sure you won't be overwriting a file that already exists. So for example you could type:

```
% mv foo bar
```

This would move the file "foo" to become a file called "bar". More on using move in relation to directories is listed in section 4.3.4, and more details about how paths work are in section 4.5.

4.2.6 Deleting a file

Sooner or later you'll find a file that you don't need anymore, and in the interests of saving disk space you'll want to delete it. The following command does just that:

```
% rm filename
```

`rm` (ReMove) will simply delete the file (in the example called "filename") from the system. After this command has been run you will not be able to get the file back, there is no "recycle bin" for your shell on cent1, so always be sure before you use rm.

`rm` also accepts a `-i` flag, but it uses it slightly differently. If you type:

```
% rm -i filename
```

You will be asked if you are sure you want to delete the file "filename" from the system, this is often a good idea to ensure you haven't typed the wrong filename, but if you're deleting a lot of files can get annoying.

4.2.7 Finding out how big a file is

The files in your directories will grow and shrink depending on how much information you fill them with, however the easiest way to find out how large a file is is to use the du command:

```
% du -k filename
```

du (Disk Usage) will display how much space the file "filename" consumes on the disk, normally it uses a unit of 512 bytes, but since nobody else does then the use of the -k flag will print out the result in terms of KiloBytes (Kb) which is a fairly normal amount used elsewhere.

If you want to know how large a directory is including all the files inside it it's best to use the following flags:

```
% du -ks directoryname
```

This will display the size of the directory "directoryname", and all the files in it, and in any subdirectory of it to you in Kb, this is quite helpful for determining how much space you're really using.

4.2.8 Concatenating files

Sometimes you will have two or three files full of text that are all related, and perhaps would be better all in one file. Instead of a lot of tedious editing (see section 5) you can simply use the concatenate command. In its most basic form you simply type:

```
% cat foo
```

This will just output the file "foo" to your terminal in a rush, which isn't very helpful at the minute, however if you type:

```
% cat foo bar
```

Then it will output the contents of file "foo" followed by the contents of file "bar". After this you just need a way of doing something with the output.

```
% cat foo bar > baz
```

Now cat will output all of "foo" then all of "bar", but instead of sending the contents of both files to your terminal it will send them to the file "baz" overwriting it with the contents of foo then bar. Although this example only shows two files you can add as many as you want to the cat command, as long as you end it with "> target". For example "`cat foo bar baz gzork > quux`" will output all of the files which are its first four arguments (foo, bar, baz and gzork) to the file quux, one after another. It is also worth noting that the original files are left intact.

4.3 Directories

Up to this point a lot of discussion has taken place which has mentioned directories, however no indication of how to use them has been given. There are really only a couple of things you need to know about directories to start using them.

4.3.1 Making new directories

To make a new directory inside the current one, you can simply use the following command:

```
% mkdir foo
```

This will make a new directory called "foo" in the current directory. Your directories can be called anything you want, and common directories to find in home directories include "bin" (for personal programs), "work" (for documents you've written), "src" (for source code you've written), "Mail" (for all your stored email). As stated above you don't have to obey a structure in your home directory, and can do what you want.

4.3.2 Removing Directories

If you want to remove a directory you can simply use the "rm" command from earlier. However it will need an argument in order to work:

```
% rm -r dn
```

This will use the rm command recursively, that is it will enter the directory (called "dn" in the example, but whatever your directory is called should go here) and delete all the files in it, then move to every subdirectory and do the same before finally removing the directories and last removing the directory in question. Use of "`rm -r`" will remove the entire directory and all its contents, so be careful with it. As before you can do "`rm -ri`" to be asked about each file in turn before it's deleted from the system.

4.3.3 Changing directory

Changing directory is done by use of the “cd” command (Change Directory). If you give this a target directory you will go to that directory. For example, if you are in your home directory, and it contains a directory called “new” then:

```
% cd new
```

Will move you to the new directory (and `pwd` will prove that point). Moving back up the directory structure uses a neat feature of the directory structure.

If you move to a directory, even a totally freshly created one, and type `ls` you will see that it contains two files ‘`..`’ and ‘`.`’ - these two files have interesting properties.

The ‘`.`’ file is the same as the current directory, it literally points to it, so “./filename” points to the file called “filename” in the current directory (but then again, so does just “filename”). This is useful for some things to do with your path, which will be shown later (see section 4.5).

However the ‘`..`’ file is a pointer to the previous directory, so if you change to the directory ‘`..`’ then you move back up the directory structure by one directory, for example:

```
% cd ..
```

This will take you back to the previous directory, however the previous directory also contains a ‘`..`’ entry, so you can do:

```
% cd ../../
```

You will jump backwards two directories.

One final and important note about the use of cd is that if you just type `cd` on its own with no arguments at all then you will be returned to your home directory. This means if you are deep in a directory tree you can just type `cd`, jump back home and move off somewhere else.

4.3.4 Copying and moving with relation to directories

One of the most used copying commands is to copy a file from where it is currently into a subdirectory. This may be to make a backup of it, or to organise your files better, or even to copy it to another users space so they can edit that file. A simple example of this would be:

```
% cp output backup/
```

This example presumes you have a directory called “backup” and a file called “output”. it will copy the file output into the directory backup, but keep the name the same. If you wanted to copy it into the subdirectory and change the name you could use:

```
% cp output backup/old-output
```

This would copy the file “output” into the directory “backup” and rename it, so a copy of the file would exist called “old-output” in the subdirectory “backup”.

If you attempt to copy a file into a directory that has a subdirectory of that name then `cp` will print the error “cp: cannot overwrite directory ‘name’ with non-directory” (where “name” is the name of the file). However if you copy a file into a directory with the same name it will overwrite it.

Another likely task is to copy a file from a subdirectory into the current directory, often this is done so you can process the file without risking changing the original. An example of this would be:

```
% cp data/test1 ./
```

This will use the cp command to copy the file "test1" from the subdirectory called "data" into the current directory (here called "./" for reasons that will become clear in section 4.5). Assuming the directory and file existed after doing this a copy of the file "test1" would be in your current directory, and changing it would not affect the original.

Copying a file from the parent directory into the current directory is another thing which at first might look like it has a confusing syntax but actually makes sense. An example would be:

```
% cp ../new-foo ./foo
```

This would copy the file "new-foo" from the parent of your current directory to the current directory you are in (which would be a subdirectory of the parent) and rename it "foo". The reasons for this syntax are explained in sections 4.3.3 and 4.5. But to reiterate, each directory has a file called "`..`" (dot dot) which is the parent directory, and a file called "`.`" (dot) which is the current directory, so in essence the cp program just follows the two paths logically, starting by hopping up to the previous directory "`../`" and getting the file "new-foo", then going back to the current directory "`./`" and copying that file to the new name, "foo".

You could just write that command like:

```
% cp ../new-foo foo
```

But as you can see its clearer to type "`./foo`" instead of "`foo`" which tends to almost run into the origin path visually and thus makes spotting typos harder. Largely this is a style issue, you may prefer not typing "`./`" or you may find it adds clarity, its up to you.

Lastly you'll sometimes want to copy from one subdirectory to another, without actually going into one of those directories. This could be achieved with a command like:

```
% cp data/test1 backup/
```

This copies the file "test1" from the subdirectory "data" into the subdirectory "backup", where since a name is not specified it simply calls it the same thing. So this will enter the data subdirectory, copy the file "test1", then enter the subdirectory "backup" and write out a copy also called "test1". Of course this relies on there being those two directories and that file in existence, if this isn't the situation then cp will spit out an error.

Sometimes you will want to make a copy of a whole directory, for backup purposes or to later change a few things about that directory. If you want to copy an entire directory then you need to know about the '`-r`' recursive flag for cp, as shown in this example:

```
% cp -r Mail Mail-backup
```

What this does is to recursively copy all the files and subdirectories of the "Mail" directory (one of the more common places to store old and important email) to a directory called "Mail-backup". The new Mail-backup directory will be an exact duplicate of the Mail directory, at the point that the command was run, all subdirectories and all files within will be copied.

Similarly to the above examples some short examples for moving files in relation to directories will now be given, you will see that mv and cp work with their origin and target arguments in a very similar way.

Similarly one of the most common things you'll want to do is move a file into a subdirectory. This is commonly used as a trick for organising your files. An example of this is:

```
% mv rant writing/
```

This will move the file "rant" into the subdirectory "writing", thus filing it away neatly. After running this command the file will be gone from your current directory and will instead exist as a file of the same name in the subdirectory "writing".

Another useful task is to move a file from a subdirectory into the current directory, this can be done with a command like the following example:

```
% mv data/report ./
```

This will move the file “report” from the directory “data” and place it into the current directory (as shown by the “./” notation, again please see 4.5 for an explanation of how this works).

For similar reasons as to why you’d want to copy a file from one subdirectory to another you’ll sometimes want to move a file from one subdirectory to another. This can be done with similar syntax:

```
% mv data/test1 backup/
```

This will move the file “test1” from the subdirectory “data” to the subdirectory “backup”, after this command is run (assuming all the files and directories exist) you’ll find the file gone from the “data” directory and that it now appears in the “backup” directory.

You might want to move an entire directory, either to make space in the file-system for something else, or to rename it. This can be done by the following command:

```
% mv data old-data
```

This will move the directory called “data” to a directory called “old-data” essentially renaming it, all its contents will still be intact, including any files or subdirectories that it contained. Of course this command would also work on files called “data” and “old-data”, if you used a command like this it wouldn’t however:

```
% mv data/ old-data/
```

Although it would still work for directories, again it is a personal style issue.

Lastly you might want to move a directory into another directory, often for organising your home directory. This can be achieved by a command like:

```
% mv data backup/
```

The difference between this command and the one above is that the backup directory exists, and so instead of printing an error message it places the “data” directory *i*nside the “backup” directory. So now “data” is gone from the current directory but is a subdirectory of “backup”.

4.4 Finding files

Before too long your home directory will end up full of several directories, all with several subdirectories, and all packed with files. At this point remember exactly where you put each file gets to be a bit of a problem. This is why the find command was created, it’s more mundane uses aren’t so complex, so for example:

```
% find . -name 'foo'
```

Will start at the current directory (‘.’ remember), and search for all files in that directory, or a subdirectory called “foo”, and will print out the path to each of them.

However this isn’t very useful if you can’t remember the exact filename, so to do this you can use various characters to affect the search. For example:

```
% find . -name 'foo*'
```

This will find all files called “foo”, or that start with “foo”, the * matches any number of any characters, so this would find files called “foo”, “foobar”, “foo-the-file-you-want”, “fooooooOOO”, etc. Similarly if you use:

```
% find . -name '*foo'
```

It will find all the files that end in “foo”. This is especially useful with commands such as `find . -name '*.doc'` which will find all files that end in .doc, i.e. all the word documents in your directory.

The full documentation for find is stored in `man find` and is worth reading carefully. Find can be used to find files in a number of interesting ways, and if you want you can run a command against all the files you find, but that kind of thing is beyond this document.

4.5 Paths, relative and absolute

Most the above examples have dealt only with files in the current working directory. However files in other directories can be accessed and affected if you know their locations. There are two ways of doing this, absolute paths, and relative paths, and either one can be used to point *a*ny command at the files you want it to affect, this includes `ls`, `cp`, `mv`, `rm`, `grep`, whatever you have your $EDITOR set too or indeed anything else you can think of or write.

4.5.1 Absolute paths

Absolute paths are the easiest to describe, you simply start from the root of the file system (called ‘/’ which is always the left most symbol of any full path) and give all the directories between that and where the file is stored. An example of this would include where the ls program is stored, the full path to this is: “/usr/bin/ls” i.e. the directory called “usr” in the root directory, the subdirectory “bin” and finally the file “ls”.

If you wanted to use an absolute path you could for example copy a file from one user’s home-directory to yours (assuming you had permission). This example presumes the other user’s username was “foo” and their home directory was in the “/home/cent1/20/” directory.

However to use full paths you also need to know the full path to your own home directory, assuming your username was “bar” and your home directory was in the “/home/cent1/15” directory then the command would be as follows:

```
% cp /home/cent1/20/foo/the-file /home/cent1/15/bar/
```

Which goes right to the root directory / and goes forwards until it hits the file you want “the-file”, then for the target directory goes back to the root and forwards into your home-directory where it places a copy of “the-file”.

Whenever you’re trying to interpret full paths simply start at the left-hand side and read across to the right, that will tell you every directory and subdirectory, divided by the ‘/’ (forward slash) symbol. The last element will either be a file or a directory, but regardless of what it is it will be affected by the command.

4.5.2 Relative paths

Remembering and typing all those long paths tends however to lead to mistakes and cursing, which is why relative paths are much easier to use for most things. These can be thought of as simply all the steps needed to get to the directory you want, from the current one. A classic example is to copy something from a subdirectory to the current one:

```
% cp sub/stuff ./
```

This will copy the file “stuff” from the directory called “sub” into the current directory, which can be shown by ./ (see section 4.3.3 for reasons).

Another common task is to copy something from the directory above the current one in the directory tree into the current one, to do this you would use something like:

```
% cp ../stuff ./
```

Which tracks back to the previous directory (also known as '`..`') and reads the file called "stuff", depositing a copy into the current directory (again shown by '`./`').

You can also nest these lookups, so that the command:

```
% cp ../../../../stuff ./
```

Will interpret the path "`../../../../stuff`" from left to right, starting off by going up a directory ("`../`") then going to the "`../`" entry of that directory, which is of course the parent. Overall this command tracks up four directory levels, then copies the file "stuff" to the current directory ("`./`").

So the basic principles of relative paths is that you have to know your current working directory then start reading them from the left to right. Remember that each directory includes two entries automatically, "`../`" is a link to the parent and "`./`" is a link to that directory itself. This means for each "`../`" in the directory you see mentally think of that as going up one directory in the tree, also remember you can go up several directories before starting to go into subdirectories.

For example if you had one directory whose full path was:
/home/cent1/20/frank/stuff/data/results/set1/

And another directory who's full path was:
/home/cent1/20/frank/stuff/backup/experiments/october/

If your shell had the second directory as its current working directory, and you wanted to copy a file from the first directory you could use a command like:

```
% cp ../../../data/results/set1/* ./
```

This command looks less frightening once you start to break it up carefully. Remembering that the current directory is:
/home/cent1/20/frank/stuff/backup/experiments/october/

Then just read it from left to right, it starts with "`../`" so up a directory, that puts us in:
/home/cent1/20/frank/stuff/backup/experiments/

Then another "`../`", which would put our current directory to:
/home/cent1/20/frank/stuff/backup/

And a third "`../`" so now our directory is:
/home/cent1/20/frank/stuff/

At this point we see the command has the directory "data", so we enter that directory, `cp` is now looking at:
/home/cent1/20/frank/stuff/data/

Then the "results" directory is listed, so `cp` moves itself to:
/home/cent1/20/frank/stuff/data/results/

Finally we see the "set1" directory, so `cp` focuses on:
/home/cent1/20/frank/stuff/data/results/set1/

The path then includes the '`*`' (star) character. This character is a "wildcard" in that it matches anything. So the `cp` command copies all the files in this directory, and that's the origin finished.

`cp` then moves onto interpreting the destination for this copy, but first it resets where its focusing on to the current working directory:
/home/cent1/20/frank/stuff/backup/experiments/october/

The only symbol in the destination is the "`./`". This means that `cp` knows the destination is the current directory, and so copies all the files identified by the origin (above) to the current working directory.

4.5.3 What can be done with $\sim$

In connection with the use of relative paths there is the `~` (tilde) operator. This can be used as a substitute for the home directory in a number of ways. For example:

```
% less ~/foobar
```

Will use less to view the file called "foobar" in your home directory, whatever your current directory is. Similarly useful is:

```
% less ~bob/interesting/stuff
```

Will use less to view the file called "stuff" in the subdirectory "interesting" of the home directory of user "bob". This use of ~ makes viewing, copying and generally working with files in your and other users home directories much easier.

4.6 tar and tarfiles

There is a tool for Unix systems (and Windows systems too, but its less common to find it installed there) for dealing with bundling up a large number of files together for storage purposes. It's called "`tar`" (originally it stood for "Tape ARchive" but you don't have to be backing up onto tape to make use of tar). The archives it produces are known as "tarfiles" (and sometimes colloquially known as "tarballs") and are a single file containing many files (a bit like a large bucket with a number of items in). Below is a quick guide to using the most common features of tar. For more complete documentation, as ever, see "`man tar`".

The use of tarfiles can make transferring a large number of files, or backing up a large number of files much easier, you can for example create snap-shots of your home directory and then copy them elsewhere for personal backup, or just bundle up a large number of files before sending them via email. In its simplest form you can use:

```
% tar -cf newarchive.tar foo bar baz
```

This will use the tar command to create (the '`c`' option) a new archive, the '`f`' option means that the word after it will be the name of the new archive, in this case "newarchive.tar" In classic Unix tradition this doesn't have to end in .tar (you can call the file anything you want) but by convention it does. Finally it will add the files "foo", "bar" and "baz" to the archive.

If you want to see the contents of a tar file then the following command will do just that:

```
% tar -tf newarchive.tar
```

This will simply list all the files (the '`t`' option) in the selected file (the '`f`' option again), in this case "newarchive.tar".

The final thing you're likely to want to do routinely to tarfiles is extract the files from one so you can use them. The easiest way to do this is the following command:

```
% tar -xf archive.tar
```

This will extract (the '`x`' option) all the files from the tarfile called "archive.tar" to the current directory, subdirectories and all.

A very useful variant of this is the following:

```
% tar -xpf archive.tar
```

This will extract (the '`x`' flag) the files ('`f`' flag) from the tarfile called "archive.tar". However it also has the '`p`' flag. This causes the permissions of all the files in archive.tar to be preserved during output, meaning whatever permissions were set on those files when they were put into the tarfile they will have when extracted. See section 4.8 for more on permissions.

The `p` flag can be used in most other tar operations, so if you want to ensure the permissions are preserved when creating an archive you can use:

```
% tar -cpf archive.tar foo bar baz
```

Which will add "foo", "bar" and "baz" to a tarfile called "archive.tar" and preserve their permissions.

4.7 gzip/gunzip

If you have exceptionally large files that you don't need to access that often you can compress them using the "gzip" program. This reduces their size on disk but means they need to be "gunzipped" before they can be used again. To compress a file you simply use:

```
% gzip filename
```

Where "filename" is the name of the file you want to compress. After the program has run successfully it will rename the file to "filename.gz", and examining that file with "file" will show its "gzip compressed data".

Before you can access the data in a gzip compressed file you need to unzip it, this can be achieved in much the same way as it was compressed, with the command:

```
% gunzip filename.gz
```

After that command has run you should find that the file called "filename.gz" is now just called "filename" and it will be uncompressed and fully usable by any program.

Because gzip and gunzip are so commonly used on tar archives there is a flag that can be used with tar in order to make it automatically compress tarballs. If you use:

```
% tar -czf archive.tar.gz foo bar baz
```

It will create a tarfile called "archive.tar.gz" and put foo, bar and baz in it, then compress the archive with gzip.

To extract a .tar.gz archive you need to use the z flag again, for example:

```
% gunzip -xzf archive.tar.gz
```

That will act like a normal "tar -xzf" but will also gunzip the archive first.

On a related note you may find sometimes that .tar.gz files can have the alternative extension (such as .tgz). This indicates that they may be either packages for the Slackware Linux distribution, or that the person who made them available simply named them that as they liked to use 3 letter file extensions.

The best way to work out if something is a tarfile is to use the `file` command on it. See section 4.

4.8 Permissions

A lot of this document has so far mentioned permissions in one way or another, but nothing has been discussed in earnest how these work. The Unix permissions model is actually quite simple and is based on a few concepts. Primarily that all objects on the System (both files and directories) are owned by some user and some group.

Secondly that all these objects have three sets of "permission bits" one set for the owner, one set for the group and one set for every other user on the system. Each of these permission bits has a "read bit" shown by 'r', a "write bit" shown by 'w' and an "execute bit" shown by 'x'. These permission bits are always listed rwx and are repeated three times, the first for the owner of the file, the second for the group that the file belongs too and thirdly for everyone else.

For files the meaning of these three bits are fairly obvious, a read-bit means those users can read (and therefore make copies) of that file. The write-bit means that those users can write data to the file, changing its contents, even to the point of emptying it, and the execute bit means that the user can run the file as a program (of course if it's not really a program then this doesn't do them much good).

For directories however the meanings change a little, the read-bit on a directory means that that group of people can "read" the directory i.e. `ls` it and see what files and directories it contains. The write-bit on a directory means that they can write changes to the directory itself, meaning they can create or delete files inside that directory, even if they don't have permissions to do so to that file itself. It's worth being very careful with assigning write permissions to directories. Finally the execute-bit governs if that group of users can change to that directory as their current directory.

To find out what the permissions are on something it's best to use the ls command again, but this time with the `-l` (long output) flag:

```
% ls -l
```

This will give you a list of all the files and directories in your current directory. You should see that files have a single dash '-' followed by 9 permission bits, 3 for user, 3 for group and 3 for other, followed by other information (and a plus symbol ('+') at the end to signal the end of the permissions. Directories should have a 'd' at the start to indicate they're directories, this will be followed by their 9 permission bits.

You may notice that some entries in the directory have 'l' or indeed other characters for their first bit. 'l' indicates a symbolic link, for other symbols consult `man ls`.

To change the permissions on a file or directory you need to use the chmod command (CHange the permissions MODe of a file). It's use isn't that complex and centres around the use of three flags, 'u' for the user, 'g' for the group, 'o' for other. It also includes an 'a' flag that affects all three groups.

These group flags are then followed by a '+' or '-' symbol, depending on if you are giving them that permission or taking it away (obviously if that mode is already set it does nothing).

These symbols are finally followed by some (or all) the letters 'r', 'w' and 'x' which obviously represent the read, write and execute bits. This isn't actually as complex as it sounds, as I'll illustrate with a few examples:

```
% chmod o-rwx foo
```

This command removes the read, write and execute privileges for the "other" group from the file or directory (here called "foo").

```
% chmod u+x foo
```

This gives the user execute permissions on the file, meaning if it's a program it can then be executed by typing "./foo", since its in the current directory.

One of the more useful commands is to remove all of the permissions from the "groups" and "other" category. To do this the following command works:

```
% chmod go-rwx foo
```

Which should ensure that no one but you can access the file, or directory, called "foo" in any way. It's best to check with "`ls -l`" that this worked.

chmod can also use the '=' (equals) symbol to set permissions exactly, as opposed to adding or removing them, so something like:

```
% chmod u=rwx foo
```

Would ensure that your user had read, write and execute permissions for the file called "foo".

It should be noted you can add multiple strings to chmod, separating them with a comma, this means instead of doing multiple commands a single line like this one:

```
% chmod u=rw,go-rwx foo
```

Would ensure that only you could read the file "foo" and that no user from your group, or the other category could.

4.9 Quota

The amount of space you have available for use on the Unix systems is restricted by a quota, well in fact 3 quotas. One for your home directory, one for your email inbox and one for your space on the University webserver. These can be checked with:

```
% quota -v
```

This command will list each of the file-systems you have a quota on that are available at that point. It will list the “usage” of that file system, its “quota”, which is what you’re supposed to stay below and finally a “limit” after which point the System will start to refuse your ability to add more data to that file system (e.g. by creating new files). It will also tell you a few more interesting things.

Generally for the Lancaster Unix systems the three file systems you will see are “/export/home” which is your home directory, “/home/mail” your inbox and “/home/www” which is your web-space.

4.10 H: Drive

While at the University you have access to your “H: Drive” (Home Drive) system. This is a large block of disk space available to you from any windows machine you are logged into. However this space is also accessible via the Unix server with the use of the “lan” command (Which is part of homegrown, see section 13.1). Its use is fairly simple:

```
% lan
```

It will then prompt you for your password, and then connect to the system that hosts H:. Once there you can use `get filename` to get a filename from Central Files to your Unix current directory and `put filename` to send a file from Unix to your Central Files space.

Full documentation for “`lan`” is available at:
http://hedgehog.lancs.ac.uk:8080/mail/lan.html from any machine on campus. You can also type “`help`” when running `lan` to get a print out of all the commands it accepts. An example or two follows for clarity.

`lan` is most important for getting files from the Unix server to central files, or from central files to the Unix server. Since all the lab machines on campus have the so called “H: drive” bound to your space on central files this means the following is an example of how to move a file from your desktop lab machine to the Unix systems.

1. First copy or move the file (called “filename” for this example) onto your H:\ on the Windows Lab machine.
2. Then switch to your PuTTY login to cent1.
3. type: `lan` at your prompt, it will ask you for your password.
4. type: `get filename` to retrieve the file called “filename” from central files, it will be copied to your current working directory.
5. type: `quit` to exit `lan`. You will be dumped back into your current directory, where you will see the file.

However the other main use of `lan` is to copy files from cent1 to your lab machine, this is the best way to view .pdf files or images like .jpg’s as the shell accounts on cent1 obviously have no graphical capabilities (although if you’re part of computing you can view them in A1a on the Unix terminals). This can be done by the following:

1. Make sure your current working directory is the one containing the file you want to move to central files (which again will be called “filename” for this example).

2. type: `lan` and give it your password.
3. type: `put filename` this will copy the file to central files.
4. type: `quit` to exit `lan`.
5. Use explorer to look at your H:\ drive, you should fine there is now a file called "filename" in the base of your H: drive and you can now use Windows programs on it.

4.11 Moving files to and from the Unix systems from off campus

Quite often you will want to move files to and from the Unix machines but will not be sat at a lab machine, the following will work from either an off campus windows machine, or with some adjustment a Unix machine located anywhere.

There are two ways of doing this, either by using the "scp" program, which uses the Windows command line, or by using the GUI program called "winscp".

The easiest way if you're used to FTP style programs is to obtain a copy of winscp. This can be downloaded free of charge from:
http://winscp.sourceforge.net/eng/

Its fairly simple to use and with a minimal amount of testing you should get it working.

The alternative is to use a program called "scp" (SSH Secure Copy, see section 2.1) this can be obtained from:
http://www.chiark.greenend.org.uk/ sgtatham/putty/download.html
And runs using the Windows command line.

This program will then be used to either move files from your off-site machine, or drag files down from cent1 to your off-site machine. Unless you are running an SSH server you will never push files from cent1 to the machine.

So to copy a file from cent1 back to your off-site machine you need to roughly follow the following steps:

1. Open a command prompt on your Windows machine, generally by hitting the Start button and selecting "run" then typing `cmd` and hitting return (although for those of you with Windows 95/98/ME machines you need to type `command`).
2. Use the DOS `cd` command to move to the directory that scp.exe has been downloaded to. If scp.exe was copied to somewhere in the path (such as the C:\Windows\ directory) then you can skip this step.
3. type the following:
   ```
   scp "C:\My Documents\filename" username@unix.lancs.ac.uk:~/
   ```
   This will copy the file "filename" from the directory "C:\My Documents\" to your home directory on the University Unix systems.
4. The command will prompt you for your password, type it when prompted.
5. When its finished transferring you can close the command prompt.

To copy a file from the Unix systems to your off-site machine use the following example as a base:

1. Open a command prompt on your Windows machine, generally by hitting the Start button and selecting "run" then typing `cmd` and hitting return (although for those of you with Windows 95/98/ME machines you need to type `command`).
2. Use the DOS `cd` command to move to the directory that scp.exe has been downloaded to. If scp.exe was copied to somewhere in the path such as "Windows" directory in C:\ then you can skip this step.

3. type the following:
   `scp username@unix.lancs.ac.uk:~/filename "C:\My Documents\"`
   This will copy the file "filename" from your home directory on cent1 to the directory "C:\My Documents\" on the machine you're sat in front of.

4. The command will prompt you for your password, type it when prompted.

You should note that the syntax for scp is virtually identical to that of the `cp` command (see section 4.2.4) except that before giving the path of the file you have to say "username@machinename:" , "username" being your remote username, "machinename" being the machine you want to copy the file off, and both of those followed by a colon.

4.12 scratch ($scratch)

On the central Unix systems sometimes you will need more space than is ever going to be given to you on your quota, for this reason the scratch file space exists. The location of it is stored in an environment variable (see section 6.3) called $scratch. What this means is that if you type:

```
% cd $scratch
```

You should find yourself in a directory somewhere under /scratch/ in the hierarchy, this is your "scratch space". You can put files here of fairly big sizes (multiple hundred megs) and do things like program compiles too big for your normal home directory. However files in $scratch will be deleted after a couple of days, or if the space is needed by newer files, so be careful what you store in there. As long as you don't put anything permanent in it you'll generally be ok.

4.13 webspace ($WWWHOME)

Assuming that you've applied for webspace (if not see http://www.lancs.ac.uk/iss/registration/getapage.htm) then you can access the directory that holds your HTML files from the Unix system. The location for your directory should be stored in the variable $WWWHOME, however this will just be your home directory on the webserver, the directory that web-content is served out of is called "public_html" and is stored in that directory. So to get to the directory that should hold all your files simply type:

```
% cd $WWWHOME/public_html/
```

You can also use this method to copy files, and the following command will copy a file called "index.html" from your current directory to your webspace.

```
% cp index.html $WWWHOME/public_html/
```

4.14 mailspace ($MAILHOME)

The final place in the file system stored as an environment variable is $MAILHOME, which allows you to go to the quotad space the mail server can see, this contains your INBOX and sometimes a mail-box containing sent-mail as well. Really there is very little point going to this space (it is useful space however, as it stores all your incoming mail), but it can be helpful to know its there, for example to backup your inbox.

4.15 Dealing with awkward files

There are a number of files which are awkward to deal with via your shell account. These include graphics files, PDF files, MS office documents, Windows executables... that kind of thing.

Some of these can be dealt with on the Unix machine, this includes PDF files and word documents, although to use more graphical tools on these follow the instructions in section 4.10 to move them to another machine.

To view a PDF file you'll need to extract the text from it to read (you will sadly loose all the graphics from it) to do this you can use a command like the following:

```
% pdftotext foo.pdf
```

This program requires using the homegrown software (see section 13.1). What it does it it extracts all the text from a pdf and writes it to a file called "filename.txt" so in this case it would extract the text from "foo.pdf" to a file called "foo.txt". If you want to change the name of the file to extract too you have to provide another argument, such as:

```
% pdftotext foo.pdf text-of-foo
```

This will extract all the text from "foo.pdf" and will write it all to the file "text-of-foo".

Word documents are a slightly different story, they require a program called "antiword" to read. This is stored in the ∼clinch link-farm (for more information see section 13.8). An example would be:

```
% ~clinch/bin/antiword report.doc | less
```

This will extract all the text from the file "report.doc" (assuming its really a word document, try using `file` on it), and then present the text for viewing in the pager `less`, so space bar for down a page, '`b`' for up a page, '`q`' to quit.

For other files (such as graphics) its often best just to use `lan` or `scp` (see sections 4.10 and 4.11 for more information) and view them using your lab machine.

5 Editing Text

5.1 why is this important?

Many people when reading this for the first time, and inexperienced with Unix systems may question why such emphasis is often placed by Unix-types on the editing of just plain text. There are a number of historical reasons behind this, but essentially this boils down to a number of factors.

Primarily it is because most things you do with a computer really boil down to entering words that will later be read by someone else, email, Usenet, writing web-pages and reports... all things which largely consist of chunks of text for the human eye.

Secondly this is because Unix systems are designed with a fair degree of sensible engineering, thus everything that controls them, all the configuration files from the core of servers to how your individual programs work, is controlled by a verity of small simple text files, thus to control the text controls the System.

Thirdly, and with reference to the first point, Unix is a modular system, designed to link components together (see section 10.1 for examples of this) and so it seems silly to reinvent a simplistic text editor for say an email client, when the email client can simply invoke the user's favourite text-editor and once that user is done collect the text from it.

These factors tend to make a lot of Unix users fairly careful in their choice of text-editor, selecting one from many to make their personal favourite.

Luckily for you the casual user really doesn't need to care that much about which editor they end up in, and following this is a brief guide to the most commonly used ones, which are available on the Unix server at Lancaster, as a brief foreword it's worth noting that "`vim`" and "`emacs`" are both widely used and thus many people will be able to help you with them (and Google has many good guides to using them), using a less well known editor risks making it harder to find information about it.

Its also worth noting that the precursor to `vim` (the venerable `vi`) was used for years by Secretaries at AT&T to write patent applications, and I've also heard tales that staff at MIT who weren't programmers used to write LISP code to extend the copies of `emacs` they were using (they were told it wasn't programming, just using the editor, they *k*new they couldn't program, but were fine with "just using the editor" when in reality LISP is one of the oldest languages known to Computing), so these editors are *n*ot too hard for non-technical users to use.

Further documentation is available online at:
http://hedgehog:8080/help/editors.html

5.2 vi and vim http://hedgehog:8080/help/vi.html

One of the true old classic editors of the Unix world is the editor "vi", which has been recently superseded by the more up-to-date "vim" editor (Vi IMproved), because of this the use of vi should probably be discouraged and new users should learn vim to start with. To start editing a file with vim simply type:

```
% vim filename
```

It should start up and display almost a screenful of lines starting with the tilde character ('~'), this signals that those lines are empty. The last line should read " "filename" [New File]".

Typing anything at this point will probably get you confused, so don't, you see vi (and hence vim) are moded editors. This means you are either entering commands (e.g. save this file, copy that text) or inputting new text (like typing the word "dog"). This, as expected, tends to confuse new users quite a bit.

When using vim you should remember that pressing escape a few times guarantees that you will be in command-mode, and from there you can go back into insert-mode if you wish.

To go from command-mode to insert mode simply press '`i`' after this you should find that any character you type simply appears in the file. Once you're done writing new things press `escape` and you will go back to command mode.

You can only move the cursor about in command mode, your cursor keys may work, depending on your terminal, but if they don't you can use '`h`' to move left, '`l`' to move right, '`j`' to move down a line and '`k`' to move up a line.

Other useful commands include '`a`' which will put you into insert-mode at the end of the current line, and '`o`' which puts you in insert mode on a new line below the current one. If you want to save the file type '`:w`' while in command mode, to quit type '`:q`'. If you want to save and quit at the same type type '`:x`'. Deleting characters is done in command-mode and the '`x`' key (assuming your backspace key doesn't just work in insert mode, it might or might not, again depending on your terminal) will delete one character, typing '`dd`' will delete the current line.

This brief introduction doesn't really do justice to what vi or vim can do, once it's loaded you can type '`:help`' in command mode for its help system, or '`:help tutor`' to go through a tutorial and learn how to use it properly.

For more on vim simply google for words like "vim" or see the following:
http://www.vim.org/
http://vimdoc.sourceforge.net/
Where the first is the homepage of the Vim editor, and the second is a large repository of documentation for the editor.

5.3 emacs http://hedgehog:8080/help/emacs.html

Emacs is the other true classic editor for Unix. Where vi is small and minimal emacs is large and comprehensive, infinitely reconfigurable and considered by many to be more user friendly. To start it simply type:

```
% emacs filename
```

Once it's started you will see a top bar listing various familiar looking options "File", "Edit" etc. followed by a blank screen waiting for input (assuming that "filename" is a new file). It's worth noting that if you start Emacs without giving it a file to edit (just type "`emacs`" at your prompt) you will get a nice introductory text, and some instructions on various commands written as things like "`C-x u`", what this means is hold down control (the 'C' part) and press '`x`', then release both control and '`x`' and press '`u`'.

Most importantly you can press "`C-h t`" at any point. This will start the emacs tutorial (Hold control and press 'h', then release both keys and press 't'). The emacs tutorial will cover far more than this document and give you a good grounding in using emacs for everyday editing.

Since you're using a text-only console then the menu's at the top of the screen cannot be clicked on with the mouse. Instead they are accessed with the `F10` key. However they don't produce a pretty "folding down" animation. Instead Emacs opens a buffer at the bottom of the screen listing the options available, each with a key associated with it. This makes the menus fast to navigate once you start to learn a few series of keystrokes to get through them, and it also makes exploring the menus fairly easy once you get the hang of it.

Its worth noting that at any point you can press `C-ggg` to cancel whatever Emacs is doing, be that typing a command, viewing the menus or whatever. Simply hold down control and press the '`g`' key three times and Emacs should cancel the action.

Otherwise the operation of Emacs is fairly simple, any text you type should just appear in front of you in the file in question, your cursor keys may, or may not work. If they don't you can use '`C-f`' to go forwards a character, '`C-b`' to go backwards a character, '`C-n`' for the next line and '`C-p`' for the previous line. '`C-d`' will delete a character and '`C-k`' will delete the current line. Pressing the F10 key on your keyboard should allow you to move through the menus by pressing the first letter of an option.

Finally to save your file simply press "`C-x C-s`" (Control and x followed by Control and s), and to quit type "`C-x C-c`".

More information about Emacs is available by Googling for key words such as "emacs". Good examples of places to look include:
http://www.gnu.org/software/emacs/emacs.html
http://www.emacswiki.org/
http://www.lib.uchicago.edu/keith/tcl-course/emacs-tutorial.html
http://www.xemacs.org/
The first is the homepage for the Emacs editor, the second is a wiki (a collaboratively built site) about Emacs and the third is another introduction (again, please see "`C-h t`" for the inbuilt tutorial). The forth is a link to the Xemacs homepage, which is a version of Emacs that "forked" from the project several years ago. See that page for more information.

5.4 pico http://hedgehog:8080/help/pico.html

One of most friendly editors, which is also one of the least powerful is pico. It is available via the homegrown section of cent1 (see section 13.1 and is fairly simple in operation:

```
% pico filename
```

This will open the file "filename" using pico. If this is a new file you will see a blank screen with "UW PICO(tm) 3.7" in the top left of the screen. At the bottom should be two lines reminding you of the most common commands, all of which are written as `^x` where `^` means "hold down control" and x means "press this key". So to "write out" (what pico calls saving your file) you need to hold control and press '`o`' (then either select an option, or press enter). Since the commands are listed at the bottom I see little point repeating them here.

Apart from these commands pico's use is fairly simple, just typing characters will cause them to appear, if your terminal can manage it (see section 6.4) your cursor keys will work to move about the text, if they don't then `^g` lists some keys that will work for movement.

5.5 ue (Lancaster Version) http://hedgehog:8080/help/ue-master.ps

`ue` (also known as microemacs) is always available on cent1, and its operation is quite simple:

```
% ue filename
```

Will start it editing the file "filename". Once its loaded you should just be able to type and characters will appear, and help is shown in a window at the top of the screen (see section 5.4 for the notation used). Your cursor keys will probably not work with ue, and so you will have to use the ones offered in the help at the top of the screen, `^n` for down a line, `^p` for up a line, `^o` to go forwards a character, `^t` to go backwards a character. To save a file press `^f` then press the 'f' key, it should say "[Wrote N lines]" in the status bar at the bottom.

Full help for `ue` is available by pressing escape then '`h`'.

5.6 Line Counting (wc)

Following in the steps of the modular editing design discussed above you'll find that most editors are lacking in several features you would expect from a Windows editor, this includes doing things like counting the number of words in a file. To do this there is the Unix command wc, whose use is fairly simple:

```
% wc file
```

Will print out three numbers describing the file shown as "file", the first is the number of lines the file has, the second the number of words and the third the number of characters (well actually number of bytes, but each character is a byte). If you only want to see the number of lines then use:

```
% wc -l file
```

To only print out the number of words in a file use:

```
% wc -w file
```

5.7 spell checking (ispell)

Also none of the Unix editors includes a spell-checker, this is because the System itself has a spell checking program in the form of “ispell” (originally “spell” was used, but ispell is far nicer to use). This is invoked by typing:

```
% ispell filename
```

This will go through the file spotting words it thinks are spelled incorrectly and asking you what to do about them, offering choices numbered from 0-N and some other options at the bottom of the screen. In its default mode it will also create a backup file called “filename.bak” which has the original text in, in case of mistakes. ispell is fairly powerful and if you want to know more you can use `man ispell`.

6 Prompts and Shells

6.1 What is a shell?

When you log into a Unix system you will see a bunch of text, followed by the hopefully now familiar prompt (which may be a `$`, or a `%`, or may be more complex). What you are seeing at this point, and what you will be interacting with to start all your programs is a program known as a "shell". A shell is just a program like any other, its not really that special. In it's simplest form all it does is display a prompt and wait for a line of input. Once you type in a line and press return it works out what program you want to run, starts that program executing, and when the program finishes and the shell has control of the terminal again it prints out its prompt, waiting for another command. The default shell used at Lancaster University is "tcsh".

More information on tcsh can be found by googing for terms like "tcsh". There is a wiki about the shell at:
http://www.tcsh.org/

But its advised that you at least skim this section before reading that.

Modern shells offer far more power to their user, customisable prompts, management of environment variables, running more than one program at once (see section 7.2 for more), completion of filenames for the user and flow control to help with writing shell scripts, but these topics aren't essential for the user to understand at first, just helpful.

6.2 Name completion

One of the more useful features of modern shells is "name completion". This is where for example you type half of a filename, or a program name in your path then press a key and it will either complete it if there is only one option or offer you the list of possible completions.

For tcsh the key that does completion is '`^d`' (hold control, press 'd'), you can test this by typing virtually any single character and pressing this key combination. Its worth being a little careful with this however as pressing '`^d`' when you have just a prompt and haven't typed anything yet will logout of the system.

6.3 Environment variables

Environment variables are simply that, they are variables (which in its simplest form means they are names associated with values), and they contain information about the environment your programs are running in.

In practise most of them can be left alone to their default states, but its worth knowing about how they work in case you want to change the way in which your sessions work. The first thing to know is how to see what's in an environment variable.

```
% echo $USER
```

The above line runs the program `echo` which simply echoes its arguments back to the user's terminal, since its argument is the environment variable $USER then it will print the contents of it, which should be your username.

The conventions with environment variables are that they are usually written in ALL_CAPITAL_LETTERS and that they have no spaces, any spaces are normally filled with underscores ('_'). Also while being set or stored they are just written normally whenever you want to access one it must be prefixed with the `$` character.

The ability to print out single environment variables isn't very helpful without knowing what variables there are in use. Too see them all simply use this command:

```
% printenv
```

This will simply print all the current environment variables that your shell knows about, there should be about 30 or 40 of them, and most can be safely ignored, some however are useful to know about:

$USER as mentioned before contains your username.

$HOME contains the path to your home directory.

$PATH contains the path of directories to search for programs, this is covered in section 4.5.

$MAIL will point to your inbox.

$SHELL will give the path to your shell, this will probably be "tcsh".

$TERM contains your terminal, This will probably be "xterm".

$PAGER contains the path to the program that deals with lots of text, this will probably be "less".

There are also a number of useful environment variables which may not be set by default, these are designed to make a user's life easier as programs will consult these before running that type of program. These include:

$VISUAL which contains your visual (full screen) editor (see section 5) and programs that want you to edit a block of text will often simply look at the $VISUAL variable and run whatever program that contains.

$EDITOR is another variable that again should contain your editor of choice, can often be ignored as most things will only consult $EDITOR if $VISUAL isn't set, however some things check this.

$MAILER often contains your Mail User Agent (see section 8) of choice, definitely optional.

$UAEDITOR is a very Lancaster specific option, the default editor for messages and pages in the LuBBs system (see section 13.11). It defaults to running ue and so you may wish to change it if you prefer another editor. It is worth noting that LuBBs also supports the $UAVISUAL environment variable, although it treats them both the same.

6.3.1 Changing environment variables

Environment variables are controlled by the "`setenv`" and "`unsetenv`" programs, which can add or remove environment variables (setenv can also change existing ones). Their use is fairly simple, to add a variable simply use:

```
% setenv VISUAL "vim"
```

This will set the environment variable called $VISUAL (note the lack of `$` (dollar) sign when setting it) to contain the string "vim". This means that whenever a program wants to get some text editing done for you it will spawn a copy of vim to handle it. If you prefer for example the Emacs editor you would type:

```
% setenv VISUAL "emacs"
```

Even if the $VISUAL variable is already set, this command will change it and overwrite its contents with the new information.

6.3.2 Using environment variables inside setenv

Its worth noting that when using setenv under tcsh that you have to be careful when including another variable in your current one, otherwise they don't expand to their values properly. The classic example is changing your path to include a personal bin file. This can be done with the following line:

```
% setenv PATH "${HOME}/bin:${PATH}"
```

You'll note that when using a variable inside setenv you need to write `${NAME_OF_VARIABLE}`, essentially put it inside braces, but the `$` (dollar) symbol outside.

6.4 What is a terminal?

The word “terminal” has been thrown about quite a bit as well, and at this point it probably needs definition. In the old days Unix computers (and in fact computers in general) were big giant things that you had one of, living at the centre of your network, nobody had Personal Computers on their desktops, instead they had terminals. These terminals got better and better over the years and so gained the ability to do much more, and so when you connect to an old Unix machine you need to tell it what kind of “terminal” (or these days emulated terminal) you are.

First things first is to find out what kind of terminal it thinks you are running, the following command will print out the terminal type:

```
% echo $TERM
```

Probably the answer you will get back will be the line “xterm”. An xterm is a reasonably advanced type of terminal, and it tends to mean that your cursor keys will work, however programs you run won’t be able to use colours to make things clearer (well the cent1 version of xterm can’t do colours, other Unix machines version of xterm can).

Other terminal types you’re likely to encounter include “vt100” and “xtermc”. vt100 is the classic fallback, a really basic terminal that won’t support cursor keys and may, or may not, support the F-keys. It will however support all the standard A-Z keys. xtermc on the other hand supports all the advantages of xterm (cursor keys, F-keys) adding colour support.

6.4.1 Changing your terminal

In order to change your terminal settings you need to change the environment variable TERM, this will be fully explained in sections 6.3 and 6.6.

The other thing worth understanding is the meaning of TTY, you’ll see it listed in several things (“`who`” for example lists the TTY the user is connected to). Basically it originally mean the port the users terminal was connected to (TTY1, TTY2, TTY3 etc...) but these days with so many terminals connecting over the network you’ll often find that TTY1-TTY7 are available only at the machine itself, if you connect over the network you a get assigned a ptty (Pseudo TTY). The TTY of ptty you get assigned isn’t really that important, its more just another way of dividing up resources among those who want them for the system.

6.5 Prompts

By now you should be quite familiar with with the default prompt on the Lancaster Unix server, which is just a single `%` (percent) symbol. However this prompt can be changed to contain quite a bit more useful information if you want to. In many shells it is controlled by an environment variable (see section 6.3) called $PS1 and changing this will cause the prompt to change to its contents. This applies to “sh” and “bash”, which are commonly used on Unix systems (see “`man bash`” for details).

The University systems however use the tcsh shell by default (see “man tcsh” for all the things this shell can do), this uses “shell variables”, similar to environment variables but only accessible by the shell, to control itself. The one that controls the main prompt is simply called “prompt” and changing the variable will change how your prompt looks. Including certain characters in the “prompt” variable gives interesting results as you can put more information in your prompt, for full details see “`man tcsh`”. Examples of these include:

`%/` gives the current working directory.
`%~` gives the current directory, but with the path to your home shown only by ‘~’.
`%c`N the last N characters of the path.
`%m` the hostname of the machine you’re logged into.
`%t` the time in 12 hour AM/PM format.
`%T` the time in 24 hour format.
`%%` an actual `%` sign.

`%n` your username.

`%l` the tty (terminal) your shell is running under.

`$FOO` prints the conents of shell variable $FOO.

`%?` prints out the return code of the last program to run, generally 0 means it ran ok, 1 means it failed somehow.

These can be combined to make your prompt more useful and informative, for example I normally have my prompt set with the following command:

```
% set prompt="%n@%m:%~ %% "
```

This displays my username, then an '`@`' (at) symbol, followed by the hostname of the machine I'm logged into (in this case "cent1"), then a single ':' (colon) before printing the directory I'm currently in, it finishes with a space followed by the a `%` symbol and another space. After this is where I'll type my commands.

The reason I do this is because I've got several accounts on multiple machines, so it helps to know what machine I'm logged into (the hostname) and what username I'm running as (my username). If the only Unix account you use is the central Unix servers its reasonable to use something like:

```
% set prompt="%~ %% "
```

Which will give you a prompt of your current directory followed by a `%` symbol to let you easily see where the prompt ends and your typing starts.

If you want to be more adventurous in your prompt usage you could include something like:

```
% set prompt="[%? %t]%~ %% "
```

Which would give you the return code, followed by the current time both enclosed inside square brackets, then the current directory (and again the % symbol to show end of the prompt). This gives you more information but a rather cluttered first line.

To unclutter it you could try something like this:

```
% set prompt="<%n@%m:%l %T> %~ \n%? %% "
```

That lot is fairly simple if taken from left to right, first a `<` symbol just because, then the username, an '@' (at), the hostname, a ':' (colon) then the TTY your terminal is connected to, followed by the time in 24 hour format and another `>` symbol to show the end of that information (the two '<' and '>' symbols are purely cosmetic and could be anything you want, or even left out). This is then followed by your path. After this you'll note the '\n' sequence, this gives you a new-line, so now your prompt is split over two lines, the new line contains the return code of the previous command followed by a single `%` (percent) symbol for the end of the prompt.

While prompts themselves aren't necessary for you to use the Unix systems they can give you a lot of information that saves time in the long run, as you don't have to type "`pwd`" a lot to remember where you are, the best way to find a prompt you like is just to spend a few minutes playing about with the examples here and the information in "`man tcsh`".

6.5.1 Using the output of other commands in your prompt

Sometimes you'll want different information than is offered by the escapes above in your prompt. For situations like this you can actually embed the contents of another command in your prompt. This uses the system of encasing commands inside "backticks" (see section 10.3). A simple example would be a modified version of one of the prompts already mentioned. The following prompt will display user@machine:n:directory. Where n is the number of files in the current directory, which is listed as directory. To do this just type:

```
% set prompt="%n@%h:`ls -l | grep -c '^-'`:%~ % "
```

While this may look fairly intimidating its actually not that hard to decode. It includes the %n, %h and %~ escape sequences which you should already be familiar with. Between the %h and %~ escapes you should notice a series of commands enclosed between two backtick characters (which should be available by pressing the key immediately above your tab key, and to the left of the 1 key). What this means is that before printing out the shell tcsh will run these commands.

The commands that are listed are: `ls -l | grep -c '^-'`, sections 4.2.2, 10.1 and 11.1.3 will help you to understand how this command works, in short it runs `ls` with the `-l` flag. This lists the long details of all the files in the directory, which of course is one file per line. `grep` then searches for the pattern `'^-'` which means look at the start of the line (the ^ character) for a dash. `grep` also has the `-c` flag which means "output a count of matching lines, not the lines themselves".

So an overview of this command is that it prints out all the details, then searches for each line representing a file, and prints out the count of those lines. In other words it counts the files in the current directory (again see the referenced sections above for more information).

6.6 What happens when you log in

You may have noticed by now that a lot of the customisations given (environment variables, prompts etc) don't stick between sessions. In order to make these changes permanent you'll need to understand a little about what happens when you log into a Unix system.

Logging in using the C-shell (which "tcsh" is a version of) mainly centres around two configuration files (see section 4.1). These are called "~/.login" and "~/.cshrc" and each contains slightly different things.

The .cshrc file is run first, it should contain all your specific commands that affect the shell itself. Your prompt should be set in here, as well any any other options you want to set (see "`man tcsh`"). Lastly this file should contain any aliases (see section 6.9.1) you want to set.

.login should contain all the setting of your environment variables, so all the lines running setenv on your $TERM, $VISUAL and $UAEDITOR variables for example. It can also contain any commands you want run whenever you login at the end, for example lines to do with home grown access (see section 13.1) or commands like running the "fortune" program (see section 13.8).

There is no real format for both of those files, they should simply contain the commands you want to run, each on a separate line. If any line has a '`#`' (hash) symbol in it, everything after the hash will be ignored, and these lines are used for comments so you can remember exactly what and why each real command is in there.

6.7 What happens when you log out

When you type "exit" at your prompt, or if using tcsh simply hit ^D then you will be "logged out" from your shell, disconnecting you from the Unix systems.

Before this happens then the file called "~/.logout" is checked, and any commands listed in here are executed. This means that you could run a program to backup a couple of files when you logged out, or print status information about your session.

An example of this is the following one-liner, which if added to your ~/.logout file will print a nice message before you disconnect:

```
echo "Goodbye" `grep $USER /etc/passwd | -F: '{print $5}'` "you were logged in from" `who am i |
```

Here echo will run, echoing all its arguments, some of which are strings (such as "Goodbye") and the sections in ` (backticks) will execute commands. See sections 10, 10.3 and 11 for more information on how this works.

6.8 running programs

So now you understand about shells, terminals and prompts you might be wondering quite what happens when you type a command and press return.

Assuming that you don't give an absolute path to a program your shell has only the relative path to work off, and for this it uses the $PATH environment variable.

The default $PATH for shells on cent1 looks like:
"/usr/local/bin:/usr/bin:.:/usr/ccs/bin:/usr/openwin/bin:/usr/dt/bin"

So given the command "`ls`", for example, your shell will look in /usr/local/bin, then look in /usr/bin, at which point it will find it. If it hadn't it will progress to look in '.' (the current directory) and so on.

This means controlling your path variable is a good idea, a good way of setting up the path to include "~/bin" (for any scripts of personal programs you have installed) is covered in section 6.3.2.

6.8.1 Working out where programs are

Since the path has now been documented you might want to see where these programs are that you've been running, or even just quickly check that a program is available without using "find" to trawl through the whole hard disk. To do this there is a function built into your shell called "which", which is used to find a program. For example:

```
% which ls
```

Will simply return the path to the "ls" program (usually "/usr/bin/ls"). This can be done for any program, and is quick to run, making it a fast way to find out if a favourite program is available, and also to see if that program is part of homegrown or not (see section 13.1).

6.9 useful things your shell can do:

6.9.1 alias

Now that you've got a shell set up with nice variables and prompts to your tastes, and you're used to regular tasks like editing text, you may find yourself becoming more familiar with commands. Sooner or later you will find a set of options you tend to type each and every time you run a command ("ls -F" for example). To make this easier you can create aliases in your shell, these allow you to run a long and complex command with a few key strokes. For example:

```
% alias ls "ls -F"
```

The above command will set up an alias for the string "`ls`", and every time that's typed the shell will execute "`ls -F`" instead of /bin/ls. Really aliases aren't much more complex than that ever, other common ones I use are:

```
% alias ll "ls -lF"
```

Which means I can see the "long ls" and thus all the permissions of files with "`ll`" or "`ll filename`".

6.9.2 Running multiple programs one after the other

Sometimes you'll want to run a series of programs one after the other, there are several ways to do this. The easiest is to use a semi-colon.

```
% first ; second
```

This will run the program "`first`" followed by the program "`second`". This is mainly used to run a series of programs in sequence. For example:

```
% reset ; clear ; source ~/.cshrc ; source ~/.login
```

This will take your shell and terminal back to the state it was in when you first logged in. First it will reset the terminal, then clear the output of that away, then run .cshrc and .login, which will reset all your environment variables and other associated settings.

A better way of doing this is the use of the `&&` (ampersand, ampersand) construct. This is used to conditionally link commands together. To return to our example:

```
% reset && clear && source ~/.cshrc && source ~/.login
```

In this case it will do exactly the same thing, reset the terminal, clear it and source the two configuration files. However because these are linked with `&&` instead of ';' symbols the flow there is an important change. If any of the commands fails (for example reset for some reason fails to work) then the shell will stop executing commands right there. This is good as it means later commands which rely on earlier ones won't fail, and also it allows you to easily review the error messages from the failed command without having to go digging through the output on your terminal.

6.9.3 clear and redrawing

There is a program (or sometimes its written into the shell, but anyway) called "clear" that does exactly that. If you've got a terminal full of the output of other commands and its cluttering your view simply use:

```
% clear
```

And your terminal will sudden be returned to just a prompt in the first line.

Similar to this is the `^L` key combination. This causes your terminal to redraw itself. If there aren't any programs running it will function exactly like clear, however most programs (mutt, vim etc) will also acknowledge `^L` and will redraw themselves on your screen. So if you are running a full-screen program and it's display becomes corrupted just press `^L` (control and 'l').

6.9.4 reset

There are special non-printable characters (i.e. ones you can't type) that can be sent to a terminal to make it behave differently, display colours, move the cursor backwards and so forth and various programs (editors, mailers, anything that takes over the whole terminal basically) use these to do various effects.

Sometimes if one of these programs fails and quits suddenly, or doesn't reset the terminal properly it will act funny after its run. Perhaps the cursor keys will stop working, perhaps all the text will have a strange background colour, that kind of thing. To deal with this there is a command that puts your terminal back to its default state:

```
% reset
```

It will cause your terminal to flicker a little then output a line telling you "Erase is delete." then a line about "Kill" and a last one about "Interrupt". Basically it'll fix about 99% of programs with terminals going wrong.

6.9.5 mesg

There is a command called "mesg" which relates to how your shell interacts with others attempting to contact you. If you use the command:

```
% mesg -n
```

All requests for talk or writes to your terminal (see section 12.5) will be automatically denied, making sure you can work in peace and quiet. If you want to switch them back on simply use:

```
% mesg -y
```

6.9.6 MOTD

When ever you log into the Unix systems you should find that your terminal contains a lot of text, this is the Message Of The Day (MOTD) which contains important information about problems with the system and reminders of maintenance and anything else the Admins think you should know.

However, if you don't want to see this information then you can simply execute the following command:

```
% touch ~/.hushlogin
```

This will create an empty file called ".hushlogin" in your home directory, if this file is there then your shell will not print the MOTD when you log in. However be aware that this means you might miss out on important information. To read the MOTD at other times simply use the command:

```
% message
```

Which will let you see the message of the day with less. If you want to start seeing the MOTD when you log in again use the command:

```
% rm ~/.hushlogin
```

6.10 Gotchas

There are a number of interesting features of Unix terminals that are worth knowing about in case you ever see them. Most of them exist for historical reasons that never went away, and sometimes they can be useful, sometimes annoying.

6.10.1 Backspaces

Sometimes you'll find either due to weirdness from the OS manufacturer, or a process feeding your terminal strange control codes your backspace key will stop working correctly. While use of "reset" will normally fix this problem (and its not that common, but it does happen sometimes) if it happens while you are in the middle of something it can be annoying. In those cases its worth knowing about "^H".

Really old terminals (I think vt100s, I wasn't using them then so I wouldn't know for sure, see section 6.4) didn't include backspace keys. In order to delete a character a combination of keys had to be included and Control and 'h' was chosen. So in situations where your backspace key fails you should be able to hold control and press 'h' to delete characters until you have the chance to type "reset".

6.10.2 Changing the size of your terminal

Normally if you just drag the corner of a PuTTY window, or whatever you're using to log into cent1 to make the window larger your terminal will automatically pick it up and start displaying output at the new size.

Sometimes however this won't happen, the terminal won't get the correct resize control codes, or perhaps the program you're using to log in won't send them. Anyway, to solve this problem simply type:

```
% resize
```

This will force the terminal to re-detect its size. When it runs successfully it will print a couple of lines about "noglob" and in-between those will print out the values of "COLUMNS" and "LINES", which should be the number of characters wide the terminal is now, and the number of lines of characters high.

6.10.3 Suspend and Resume

Before pagers were common place when people were dealing with large amounts of text being sent to their screen they would use a key combination that suspended the output, the terminal would then sit, storing new output to be displayed. When they had read that screenful they would resume console output, pausing it again if needed.

This functionality has been left in most terminals and should be available to your Unix shell account. Normally you should find your suspend key is set to "`^s`" and your resume key to "`^q`" using the notation above.

So if all output stops appearing on your terminal you might have press suspend, try pressing resume and see if that fixes it.

If you want to stop being able to suspend and resume your terminal (for example if you accidentally hit `^s` a lot) then you can use the following command to switch off this feature of your terminal:

```
% stty -ixon
```

Adding this to your .login will cause it to be run each time you login, thus meaning you will not be able to use the suspend/resume feature.

If you have used stty to switch this off and want to be switch it back on simply type:

```
% stty ixon
```

6.10.4 "There are suspended jobs."

Sometimes when trying to log out you will be greeted by the message "There are suspended jobs.". The reasons for this are covered in section 7.2 and you should see that section for how to deal with it.

7 Processes

7.1 What is a process

A lot of talk in this document has gone on about "processes" and "executing programs" and so forth, so this section will give a broad and generic overview of what a "process" is and what you need to know about them.

A program is a set of instructions that lives on disk, any file that contains instructions as to how it should run is a program. A process is an image of that code in memory that is actually running at that time. So any time you run the "ls" program a process is spawned, executes the code and returns its results.

So first some information about processes, the most important things are that they have a name that is basically the command line used to run them, that includes their name and all their arguments. Secondly they have a unique PID (Process IDentification number), this number is used by all the programs that affect processes, because while there might be 10 processes called "vim" only one of them has the PID "20963"

Processes also have whats called a PPID (Parent Process IDentification number), this is the PID of the processes that spawned them, so all processes which are started from your shell will have its PID as their PPID.

Lastly they also have a UID, which was responsible for starting them, or in certain situations has accepted the running on them in the future. Usually the UID of your processes will be the same as your user name.

7.2 Running multiple jobs

While it's technically a function of the shell this section will deal with management of multiple jobs by the tcsh shell.

If you want to see a list of the currently running jobs then it's fairly simple, just typing:

```
% jobs
```

Will list all the currently running jobs, if it returns nothing at all then that indicates you aren't running any other processes in the background.

Making your shell run multiple jobs at once is fairly easy. If you're running another program, say a mailer or an editor, and you want to quickly drop to your shell to run another program (e.g. an ls, to check the name of a file) you can press "`^z`" (control and 'z') to suspended the current process. While suspended the process will stop totally if its something that is still doing some work (like a long running process, for example a number crunching program, or a long find) it won't carry on running.

After pressing `^z` you should get your prompt back, and there should be a line of text above it saying "Suspended". At this point typing "`jobs`" will give you a list of the running jobs, each will have a number in square brackets at the front, and will say its status, "Suspended" has already been discussed but you might find some processes which say "Running", this means they are running quietly in in the background.

Processes which take a long time to run are best run in the background, to do this simply type an '`&`' (ampersand) at the end of the command line. This effect can also be achieved by typing "`bg %N`" where N is the number in square brackets in the listing from "`jobs`".

If you can see a process on the jobs list that you want to stop running you can kill it with the "kill" command (see section 7.3) however for ease of use you don't need to know its PID, if you look at the jobs list and find the number in the square brackets then just type:

```
% kill %N
```

Where N is the job number, the `%` symbol before it means it will be looked up on the jobs list, and not the process list.

Finally if you want to switch back to running a process fully in the foreground you will need the "`fg`" command. It accepts one argument of the job number you want in the foreground, so typing:

```
% fg %2
```

Will put the second job on the list back into the foreground.

7.3 Kill

The "kill" command is actually rather misleadingly named. It really sends a "signal" from a selection to a process. These signals can include pause, resume, redo from start and that kind of thing. However its default signal is the TERM signal, which tells a process to halt.

Use of kill to stop processes is fairly simple, first look up the PID (see section 7.4) of the process you want to kill (or its job number, see 7.2), then just use the following command:

```
% kill PID
```

Where "PID" is the PID of the process you want to kill (it will always be a number) or is `%` followed by the job number. Kill will normally not return anything if it runs successfully but the process that's being asked to die will sometimes say something like "Terminated", or "Exiting on signal 15.", or something else appropriate.

Sometimes you will find a process you want to kill has wedged in some way that means it's not responding to a TERM signal (which can be thought of as a polite way to ask the program to exit). In those situations you should use the following command:

```
% kill -9 PID
```

Again replacing "PID" with the actual PID of the process in question. Using the -9 option sends the signal called SIGKILL, which makes the operating system simply slay the process in question. This command is to be used only on wedged processes as it will cause them to die right away, and they may leave temporary files laying about the file system and that kind of thing.

For details of the other things that can be done with the kill command consult "`man kill`".

7.4 ps

There are several commands which can be used to find out information about the processes running on the system, the most common of which is "`ps`".

In its simplest form it will print out all the PIDs and names of all the processes running on that TTY, to do this just type:

```
% ps
```

And you will see a list that probably just includes your shell (tcsh). While this is fairly useful in and of itself you'll often find that you're logged in to two or three TTYs at once doing various things, and so you'll need to find out more than just the processes running on that login.

To see a list of all your running processes use:

```
% ps -U $USER
```

Where $USER can be either the environment variable "$USER" (as that contains your username), your actual username, or if you want to see the processes of another user their username. Since ps defaults to only showing your processes using -U with $USER is in most situations fairly pointless.

However that list can be quite long, so its worth knowing that you can do the following command to search all of your processes for ones called a certain thing:

```
% ps -U $USER | grep ' command$'
```

This uses a pipe (see section 10.1) and the grep command to find just the processes you're interested in, simply replace "command" with the name of any process e.g. "tcsh" to find all your shells, "emacs" to find emacs sessions, "less" for all your pagers etc.

Another option for ps that is quite useful is the "full information" flag, this can be used as shown below:

```
% ps -f
```

This will show you information about your processes with much more information including the UID, PID, PPID, TTY and the command line and all its arguments used to start the command.

If you wanted to know this about another user you could simply use:

```
% ps -fu username
```

Where "username" is the name of a user whose processes you want to see.

Lastly if you want ps to show you information about all the processes running on the system then you need to -e flag:

```
% ps -e
```

Most of time this isn't really needed, but sometimes you will want to be able to find out information like this. It should also be noted that the -e flag can be combined with the `-f` flag, to show all the processes on the machine and full information about them.

7.5 top

If you want a general overview of the processes that are running on the system at any one time then the "top" command is the one to use. Its use is simple, just type:

```
% top
```

You will then see a fairly regularly updating list of the top 10 or so programs, ranked by basically how much system resources they are consuming. You will also get information about how much CPU and memory are being used. Once you're finished looking then just press "`q`" to quit.

Of course the interesting thing is that top itself if running on a fairly loaded system can be a fairly large processes in itself. Its constant monitoring can be fairly consumptive of system resources. To get around this problem simply use the following line to print out the top 10 programs then exit.

```
% top -n 10
```

This will also give you all the information about memory and CPU, and is just generally a friendlier thing to use on a potentially full system.

8 Email and Unix

8.1 A general overview of email under Unix

Email under Unix is not just a large black box whereby mail flows magically into a mailbox, but its a definite part of the system and is broken down into a number of sections. A rough diagram of how mail gets into your mailbox is as follows:

Network --> MTA --> MDA --> Your inbox

Where "Network" represents any form of outside machine that could be sending you mail. MTA stands for "Mail Transport Agent", normally an SMTP server like sendmail or exim running on the Unix machine. The MTA is responsible for transporting mail between machines. MDA stands for "Mail Delivery Agent" this is a program responsible for delivering mail which is addressed to you from the machines MTA to your Inbox.

Sending mail works similarly, and a rough map is below:

MUA --> MTA --> Network

Here MUA is your "Mail User Agent", basically a mail client, that hands off to the MTA (see above) which if the mail is destined for a remote host sends it off over the network. If however the mail is for another user on the local machine the MTA hands it to the MDA and that delivers it to the user's mailbox.

Now before I get into discussing which parts of these systems you need to know about there will be a short description of what an email address really is. Consider the example below (Nb: there isn't a Unix machine at melody.lancs.ac.uk, this is used purely for example):

frank@melody.lancs.ac.uk

Now essentially what this means in terms of Unix systems is that there is an account called "frank" on the machine at "melody.lancs.ac.uk", and when mail arrives at melody.lancs.ac.uk addressed to frank the MTA hands it to the MDA and it's put in his inbox, simple as that.

8.2 Forwarding Email

If you want to forward the email from your Lancaster Account (please don't, forwards do occasionally break through no fault of the University and wind up bouncing) then its fairly simple to do.

You can either use the "Management" option for your account after logging into the Wing system (http://wing.lancs.ac.uk) or you can use a .forward file.

To change your forward at the Unix prompt simply use the following command:

```
% echo "forward@domain.tld" > $MAILHOME/.forward
```

And replace "forward@domain.tld" with the address you wish mail to be fowarded to.

8.3 Mail User Agent

The choice that will most affect your experience of Unix MUA ("Mail User Agent"), like the Editors section (see section 5) there are several popular MUAs that will be discussed in this section, and all of them are simple textual applications. While there are many graphical mail clients for Unix this document is an introduction to shell accounts, so I won't be dealing with any of them.

Historically the standard mail client for reading mail with on Unix systems was "mail". However this is not a very powerful or user friendly MUA and so its use is discouraged, however it is available if you wish to look at it. Documentation for mail (and the newer version "mailx" is available from "`man mail`" as ever.

The supported MUA for the University Unix systems is a program called "elm". It is more advanced than mail and offers far more options as well as user friendliness to its users, this document includes a short section on elm.

Elm is documented at: http://hedgehog.lancs.ac.uk:8080/mail/ and also in "`man elm`".

There are also two more MUA's installed on cent1, these are mutt and pine. It's worth noting that these MUA's are part of the homegrown package (see section 13.1) and thus there is no official support for them, so if you misconfigure them and manage to lose mail then you probably won't get any help. However they offer a much better user experience for reading and writing email.

8.3.1 mutt

mutt is a fairly simple to use MUA, said to combine the best features of elm and pine. This will only be a brief introduction but full information is available via "`man mutt`" and http://www.mutt.org/

In its default configuration it will display a line at the top of the screen listing various options (including '`q`' for quit and '`?`' for help). Pressing any of those will perform that action. The status bar at the bottom of the screen displays information about your mailbox (messages count, new message count, size of the mailbox etc). The cursor keys should move you up and down the list of messages in your inbox, failing that '`j`' will move down and '`k`' will move up. If you press the space bar on a message you can read it, it'll open in a pager much like "less", and the top bar will display the keys that affect it ('`i`' to go back to the message view, '`-`' (dash) to go back a page, space bar for next page etc.).

Mutt is well liked by many people, it can be made quite simply to use colours to highlight certain messages (like new ones), display a threaded view of your messages in part of the screen and the rest of the message in another, and by default it uses your $VISUAL variable for the message editor. It also has many more features, www.mutt.org contains a list.

Mutt is started simply with the command:

```
% mutt
```

8.3.2 pine

The other main console mail tool is pine (a "Program for Internet News & Email"), it too is fairly simple to use, and is fully documented at:
http://www.washington.edu/pine/

When you first start pine you will get a brief introductory message telling you about it, the bottom of the screen will include the various options you can do with key-strokes, the letter in question in bold. So press '`e`' to leave the greeting.

After that you will be at Pine's main menu, this is what you will see every other time you start it, and will allow you to view the list of messages or a current message, or write a new one. There is also a "setup" menu here, available from the letter '`s`', this will take you through a series of menus to define how you want Pine to be configured without having to touch its configuration file by hand.

Pine is started with the very simple command:

```
% pine
```

Its worth noting that in its default state Pine uses pico (see 5.4) for its default editor, and even if your have $VISUAL set will ignore it. To make it use your choice of editor you need to press '`s`' from its main-menu for the setup, then look for the alternative-editor options.

Alternatively edit the ~/.pinerc file and add (or change) the line that reads "editor=" putting the name of your editor after the '=' (equals) symbol. Then when you've typed in the To: and Subject: fields of your message in pine press `^_` (control and '`_`' (underscore)) to start the editor.

8.4 fetchmail

fetchmail is a program that has an apparently simple task, to fetch your email from a remote mail server and deliver it to the local one so that it can be delivered to your inbox (see section 8) as normal. Before writing a description of fetchmail it is worth noting that it is installed as part of the "homegrown" section of cent1 (see 13.1) and subject to all the normal restrictions about support i.e. there isn't any officially and if you misconfigure it and it goes wrong you might get in trouble. It's worth repeating this especially for fetchmail as misconfiguring it can lead to very noisy mail bounces.

Anyway, fetchmail is fully documented at:
http://catb.org/~esr/fetchmail/

Fetchmail is controlled by a "~/.fetchmailrc" file, for a start this must have the permissions set so that only your user can read it (see section 4.8). A template configuration file which has been commented is available from section 14.

8.5 procmail, your MDA

The MDA on the system is a program called procmail. Procmail is a general purpose program that can filter mail based on an amazing number of criteria, sorting it into various mailboxes, and do virtually anything, even trigger other programs, on incoming email. However searching for procmail on cent1 won't actually find it, as all the real email processing is handled by several other machines. However, procmail is installed on them.

Since procmail isn't even installed on the machine, much less homegrown, no support is offered for it, and since mistakes in its configuration can make it silently destroy all incoming email (in many amusing ways) or forward it to a scammer in outer Mongolia it's worth being very careful about configuring it.

This document will not teach you how to use procmail, as the subject is just too complex, this is just a note to let adventurous users know it's installed. Anyone wanting to know more is encouraged to read http://www.procmail.org/ and send test mails to be sure of your config.

As a last point if you create a ".procmailrc" file in your home directory then fairly soon (about every hour) a cronjob (see section 12.7.1) will run, moving it over to the correct place on the Mail servers for it to take effect, if this is too slow for you then use of the $MAILHOME variable should prove enlightening (again, be very careful with procmail, it is not supported and is complex).

9 The Net and Unix

Unix is classically associated with networks, and has flourished as a networked operating system for decades (yes decades, stick that in your Microsoft pipe(tm) and smoke it). This document will teach you a few useful things to know about networked Unix systems, and a few useful programs for your use.

Another more in-depth introduction is available at:
http://en.tldp.org/HOWTO/Unix-and-Internet-Fundamentals-HOWTO/

9.1 Browsing the web

It's possible to browse the web from a text only terminal. The reasons for this are many, but essentially this is helpful if there is a technical reason why the web-browser on the lab machine you're logged into won't work, and also if you just want to browse quickly, as text-only browsers don't have to download pictures or flash-animations or anything like that. To use the web browser installed simply use the command:

```
% lynx www.google.com
```

Where "www.google.com" can be any URL you want, lynx will then load, and render the page. The options at the bottom of the screen will as usual tell you what keys do what, '`h`' will give you more help, '`q`' will quit, '`g`' will allow you to enter a new URL, '`G`' will allow you to enter a new URL but will leave the current one in the buffer (useful if you just want to put another page name on the end of the current domain for example).

Movement is by cursor keys if they're working, the space bar will scroll down, '`b`' will go back up, use the enter key to follow a link.

9.2 Downloading files

While lynx will allow you to download a file sometimes you already know the location of one, and perhaps you want to download one in the background while keeping lynx in the foreground. Or maybe you want to download a whole website, following links and downloading from them too. For all these purposes there is wget.

In the examples below http://www.somedomain.net/location-of-file is used, but wget supports http:// links and ftp:// links, as well as several other types. Also the examples below deal with one option flag at a time, these can be combined.

Full documentation on wget is available at:
http://www.gnu.org/software/wget/wget.html

```
% wget http://www.somedomain.net/location-of-file
```

This sort of command (replacing the "http://..." line with the location of something you want to download) will cause wget to download the file, however there is much more wget can do.

```
% wget -c http://www.somedomain.net/location-of-file
```

The '`-c`' flag is the "continue" flag. If there is already a file called "location-of-file" in the current directory wget will assume you're trying to continue its download and start the download from the network having jumped forwards the size of the file on disk (and will then append what it downloads to the end of the file). This is helpful if a download stops halfway through.

```
% wget --limit-rate=20k http://www.somedomain.net/location-of-file
```

"`--limit-rate`" does exactly what it says, it limits the rate at which wget will attempt to download a file to the amount listed after it. This is normally listed in 'k' (for kilobytes per second) or 'm' (for megabytes per second) and is designed so that if you are on a large connection (like the University's) you don't thrash smaller sites, or if you are on a smaller connection you don't hog all the bandwidth.

```
% wget -b http://www.somedomain.net/location-of-file
```

The '-b' option will cause wget to automatically drop into the background, and save the file into its current directory, which can be handy for downloading multiple files.

```
% wget -i urlfile
```

The '-i' or "input file" option, will cause wget to read the file called "urlfile" (which should just be a series of URLs, one on each line) and download each file listed.

```
% wget -O filename http://www.somedomain.net/location-of-file
```

This will cause the file to be saved as the filename given just after the '-O' option (the Output filename). If this name is '-' (dash) then the file will be piped to standard output (see section 10.1).

9.3 Querying DNS (Domain Name System)

DNS deals with mapping domain names (for example "www.lancs.ac.uk") to IP addresses (such as 148.88.8.1) so that the packets sent to either one get correctly routed to the right machine. The easiest tool available to users is "nslookup". This is however in "/usr/sbin" on the central Unix server and so you will either need to add /usr/sbin to your path or use the command:

```
% /usr/bin/nslookup www.lancs.ac.uk
```

It's use can be fairly simple, and giving "nslookup" an IP address will cause it to try and look up the domain associated, giving it a domain will cause it to return the IP address(es) associated with that domain.

Full documentation is, as ever, in "`man nslookup`".

9.4 Seeing network activity on the host

The following section is useful for anyone who wants to know details of current network connections to and from the machine they are logged into, however this is *not* required knowledge, and not all of the concepts will be explained, its just being documented so interested parties know the functionality is there.

At any time you can see what connections are being made to and from the Unix machine you are logged into by use of the following command:

```
% netstat -f inet
```

netstat on its own will simply list all connections on all networks, this will list a lot of traffic on Unix sockets and loopback that aren't really that interesting to anyone not maintaining the machine. The '-f' option restricts netstat to looking at a family of traffic, and the "inet" family is all "Internet" traffic i.e. TCP and UDP traffic. However this will probably produce a prodigious amount of output and so the following is recommended to view it in less (as normal see section 10.1 for the reasons why):

```
% netstat -f inet | less
```

And as usually scroll down with the space bar, up with 'b' and quit with 'q'.

9.5 Seeing if another machine is running

The canonical way of seeing if another machine is running is to "`ping`" it. The ping command sends a few packets across the network to another machine, which is deemed "alive" if it responds. An example of using ping is seen below:

```
% /usr/sbin/ping www.google.com
```

If the machine you give it the address of (in this example "www.google.com" but any machine can be used, even "unix.lancs.ac.uk" which you are logged into) is working and responding to pings (they can be switched off) the command will tell you it "is alive". If it cannot reach the machine (usually because the Internet is broken between unix.lancs and the host, or the host is down) it will print "no answer from www.google.com" (or whatever machine you pinged).

It is worth noting that some systems are configured by their administrators to not respond to ping requests these days. This will either be because the administrators are trying to "stealth" the machine and hide it from viruses and port-scanners (Nb: This is not a totally effective method) or because they think it will increase the security of their machines. Either way it breaks RFC 1122 (http://www.faqs.org/rfcs/rfc1122.html) and should not be done.

Often a better method of doing this is the `traceroute` command. This will try to get to a host and print out a list of every network device it passes through along the way (so you can trace the route your packets will take). Its use is as follows:

```
% /usr/sbin/traceroute www.google.com
```

9.6 Logging into remote machines

Assuming that you have accounts on other Unix machines you'll probably want to know the best way to log into those machines from cent1. This is best accomplished with the "`ssh`" command (Secure SHell), the example below illustrates this:

```
% ssh frank@melody.lancs.ac.uk
```

This returns to the earlier example (see section 8) of the Unix machine melody.lancs.ac.uk and its user account "frank". The use of this command will log you into the machine just as if you were sat in front of it, and anything you could do there you can do via this remote login.

While the telnet command may also be used to log into machines running a telnet server its use is discouraged as it doesn't encrypt your traffic (like ssh does) and thus your password may be sniffed off the network.

9.7 Copying files from remote machines

The best way these days to copy files between Unix machines is to use ssh, or at least a program that runs over it, called "scp" (literally SSH CoPy). This works just like the normal copy program but requires a slightly different syntax (see section 4.2.4), simply use:

```
% scp frank@melody.lancs.ac.uk:~/foo ./
```

This will copy the file called "foo" from the home directory of the account "frank" on the machine "melody.lancs.ac.uk" to the current directory (known as '`./`'). Of course to copy a file back the other way you'd use:

```
% scp bar frank@melody.lancs.ac.uk:~/
```

This will copy the file called "bar" from your current directory to frank's home directory on melody.lancs.ac.uk.

scp can also be used to copy whole directories, which is fairly simple, for example:

```
% scp -r stuff frank@melody.lancs.ac.uk:~/
```

Use of the '-r' (recursive) flag will copy the whole directory called "stuff" including all its contents to the home directory of frank on melody.

scp can do far more than this, and even do automated copies without passwords if you have the correct keys installed (see "man scp" as ever).

10 Plumbing

As mentioned in passing many times above (see section 5) Unix is designed as a modular system, with many small programs all doing one job well. This in itself doesn't make these small programs very useful until you start encountering the ways that Unix allows you to glue them together.

10.1 pipes

The pipe character '`|`' (a single vertical bar, normally found by pressing the backslash key with shift held down) is the single most useful feature of Unix systems. It has already been encountered in passing, (see section 9.4 and 7.4) but its use has never been fully explained.

Simply put the pipe sends the output of one program into the input of another program. Formally it links the standard out (stdout) to the standard in (stdin) of two processes and transmits a "stream of bytes" between them. This idea is best shown through example:

```
% ls | grep 'a'
```

This runs the "`ls`" command on the current directory, but instead of sending its output to your terminal it instead sends it through the pipe and into the input of "`grep`". Now grep is used to search streams of text for strings, and here it's searching for the string 'a'. So what this does is give you all the output of "`ls`" that involves the letter 'a' in any way.

While this doesn't seem very useful now remember that you can also do things like this:

```
% who | less
```

Which sends the output of the who list to your pager (less) enabling you to view its output more easily.

Virtually any number of pipes can be put between any number of commands, forming rather complex filters, a good example of this would be if you wanted to see a list of a certain user's logins you could use a line like this:

```
% last | grep 'frank' | less
```

Which would print the last list (see section 2.8), send that through grep to search for the string "frank" and finally pipe anything it did find into less, so it could be viewed with a pager.

Of course if you have $PAGER set to something else, or are using many options with it you could use a line like this:

```
% last | grep 'frank' | $PAGER
```

Which would do exactly the same thing, but would run the contents of $PAGER as a program and send the results of the grep into that.

While piping into grep or into less are the most common tasks you are likely to perform you can do rather more complex things. For example if you have a rather large directory containing amongst other things a series of HTML files, each one named for its date, and you wanted to find the newest file you could use:

```
% ls *.html | sort | tail -n 1
```

This runs ls, and makes it look only for files with the pattern "*.html" (i.e. those files that have anything ending in .html). The output of this command, which should be all .html files in the directory is then sent via a pipe into the `sort` command, which sorts its input in descending order. The sorted output is then sent into the `tail` command, which has the options "`-n 1`", this makes tail only output the last line (number of lines outputted is set to 1), and so this outputs the newest file.

The same functionality can be done with the line:

```
% ls *.html | sort -r | head -n 1
```

Which sorts in reverse order (because of the '`-r`' flag), and then uses "head" to get the first line (which because of reversal will be the newest).

The same functionality can also be done with the line:

```
% find . -name '*.html' | sed -e 's/\.\///' | sort | tail -n 1
```

Which uses find to find the files, then sed (see section 11.1.7) to remove the string './' from the start of the line, then sorts it and uses tail on it.

As you can see when you start combining commands with pipes an amazing range of ways of doing things becomes available, and will grow as you become more familiar with the various commands.

10.2 redirects

Another feature that can be quite handy to know about is that of being able to redirect the output of a program. With pipes we're already seen how to redirect it from one program to another, but its often handy to capture it into a file for later use. And for this you need redirects.

This section will cover the basics of using redirects. And really they're fairly simple. For example if you wanted a list of all the files in your home directory you could do this:

```
% find . -name '*'
```

Now doing this is all very well, but assume that you wanted to store a copy in a file itself (say called file-list). Now you could send it via a pipe into your editor (if your editor supported this, vim can do it by giving - as a filename). This would leave you with a command like this:

```
% find . -name '*' | vim -
```

However then you'd have to wait for the system to start up your editor, and need to save the contents to the file "file-list", there is a much easier way, as your shell can write the output anywhere you want it.

```
% find . -name '*' > file-list
```

The angle bracket '`>`' is used almost like an arrow, it points the output from the command into "file-list", which if it already exists it overwrites.

10.2.1 Appending with redirects

If you want to append the output of one command to an earlier one then you'd have to use `>` to output it to another file, then join them together with the cat command, this is obviously a pain as you have to type two commands instead of just one. However this can be gotten around with the append redirection operator.

```
% ls >> foo
```

Now this will run ls as normal, then append the results of this to the file "foo", or course if the file doesn't exist it will create it, but it's often safer to use `>>` rather than a single `>` just in case you name a file that already exists, then it won't destroy its contents, just append some output to the end.

10.2.2 Redirecting input

There are certain situations in which you want to send the contents of a file into the input of a program. Assuming that you had a large file containing unsorted words that needed to be sorted into the right order you could use the following command:

```
% cat big-file | sort
```

This will echo the contents of “big-file” into the input of the sort command. However that requires invoking a whole separate program (`cat` in this case), a cleaner way of doing this is:

```
% sort < big-file
```

This will do exactly the same thing as the command above, with one less pipe and thus process. Also there are some situations where a program wants a list of commands sent into its input, in which case this syntax is usually better.

However when doing this you can also redirect the output of the sort (or whatever command you’re running) to another file as usual, this is done in exactly the way as is shown above but its worth providing an example for reference as it can look daunting at first glance:

```
% sort < big-file > sorted-big-file
```

This will run sort, inserting the contents of “big-file” into its standard input. The output of sort (which will be the sorted input) will be written into the file “sorted-big-file”.

10.3 Embedding commands in other commands

Now that you understand pipes and redirects you can do a fair deal more with your shell, combining commands using pipes as filters for output should improve your efficiency a fair degree. However there is one more operator that you can use which can prove fairly useful in writing Unix command lines, and this is the back-tick operators, which allow you to embed the results of another command inside a current one. It sounds fairly confusing but this example should clear things up.

In the description below you will encounter a character called a backtick. This character looks like this ‘ or in the font used for examples this `` ` ``. It is entered by pressing the key which is above tab and to the left of the ‘1’ key. You don’t need to press shift or control, just the key.

Now let’s assume that you wanted to see the long ls (“ls -l”) information about the command ftp. Normally you’d need to run “which ftp” and then, remembering the output of that command, run “ls -l” followed by the location. Using back-ticks you can make this easier:

```
% ls -l `which ftp`
```

This starts by running the command in the back-ticks (“`which ftp`”). This gives the result “/usr/bin/ftp”, and the shell then runs “`ls -l /usr/bin/ftp`”, then prints the result of that to your console.

Back-ticks are a handy tool in your toolbox for writing Unix commands and can be used inside any command, including aliases. An example showing the many ways to do things would be a single command line to finger the last user who logged onto the system. Now there are two ways of doing this:

```
% last | head -n 1 | awk '{print $1}' | xargs finger
```

Which gets the first line from the “last” command and uses awk to print out the first column from it (which is the username, see section 11.1.1 for more information) then passes this to the xargs command (see 11.1.14), which runs finger with that username as its argument.

With the use of backticks you can avoid such logically messy methods and make your command-lines easier to get a mental image of, which makes them easier to write and use:

```
% finger `last | head -n 1 | awk '{print $1}'`
```

What this does is run the whole section inside the back-ticks first (which runs last, gets the first line, then gets the first column of the first line with awk) and then the shell runs “finger” followed by that output as its argument, meaning it does the same thing but is more obvious as “finger” is at the start of the line, not the end.

11 Shell scripting and more useful commands

By now you should be at least familiar if not quite friendly with the command line, able to pipe commands, redirect output and so forth. And even with alias you might have noticed that typing in a lot of the same commands over and over again is getting pretty old. For this reason you need shell scripts.

A shell script is a simple method of programming that enables anyone to use the power of the Unix command line to their advantage, while they tend to be a little idiosyncratic in their behaviour they are useful.

All you need for a shell script is the following template:

```
#!/bin/sh
# Your commands go under this line
```

The most important point is the first line must contain "`#!`" followed by the program you want to interpret the commands. After that you can just list a series of commands, just like you'd type in, each on its own line.

Common things to act as interpreters include "/bin/sh" (as it's fairly small and thus isn't a load on the system), "/bin/bash" (as many people like the way it interprets commands) and the shell which you default to "/usr/bin/tcsh" which at least will ensure that you can always test the commands properly.

Each shell has its own abilities to structure these files, including flow control like if statements, for the differences see the man pages for each. But most people should be fine using "/bin/sh" for simple scripts.

Anyway, after writing the script file simply chmod it (see section 4.8) so that you have execute permissions on it then it should run just like any other command on the system, this can be done with a command like:

```
% chmod 755 scriptname
```

This uses the octal numeric modes (as described in `man chmod`) to change the permissions for the file "scriptname" and give the user rwx permissions and all other users `r-x` permissions (they need Read permissions for their interpreters to interpret the script and execute permissions to run it as a program). This could have been done with symbolic symbols like this:

```
% chmod u=rwx,go=rx scriptname
```

11.1 More useful commands

While this section won't offer many details on each of the programs listed it's more useful as a quick reference section, to find out the names of some of the more common programs and what they do, as if you need a filter to act in a certain way you can often convince a common program to do it for you instead of writing your own.

As ever see the man pages for details.

11.1.1 awk

awk is a "general scanning and processing language" to quote its man page. The most common use of awk is to divide the input into columns and only print certain columns, or to print them in a different order than they first appeared in (although it can do far more than this). An example to print the first and third columns of some input, with a tab between them would be:

```
% who | awk '{print $1 "\t" $3}'
```

11.1.2 tac

You've already been introduced to the "cat" command (see section 4.2.8), tac does the same thing as cat (in that it concatenates files) but it does so in the reverse order that the files were given to it. So if you use the command:

```
% tac afile bfile cfile > output
```

It will put all of "cfile", then "bfile" and finally "afile" into the file called "output". This is just handy for when you need to do a reverse concatenation, and don't want to write complex code to reverse the arguments.

11.1.3 grep

grep has been discussed on numerous occasions, but never really documented. Essentially it is a program that is designed to accept input of plain text and filter it, only outputting the lines that match a pattern. This makes it useful to search the results of other programs for just the sections you want.

There is also an enhanced version called "egrep" which accepts a wider range of patterns, and some important flags.

The absolute basics of pattern matching are that you can put '.*' (dot star) wildcard where you want to match anything, so:

```
% who | grep '.*foo.*'
```

Will grep for any string with "foo" in it anywhere, using ".*foo" searches for lines that end in "foo", whilst "foo.*" for ones that start in foo. This is because '.' means "any character" and '*' means "any number, even 0". For more details see the man pages for grep.

The example above was written in that style to make changing it for your own uses easier. However if you want to search for the pattern ".*foo.*" (i.e. the word foo, at any place in any line) you can just use:

```
% who | grep foo
```

And grep will automatically search for any occurrences of "foo" anywhere in the input. Indeed even `who | grep 'foo'` would produce the same results.

The advanced version of grep, egrep, can do some powerful pattern matching. However one of its most useful features is the '`-i`' flag. This makes its pattern matching case insensitive, so:

```
% egrep -i bar filename | less
```

Will run egrep in case insensitive mode ('`-i`'), searching the file called "filename" for any line with "bar" in it anywhere. It will then pipe the results out to "less" for viewing.

Because `-i` was used this command will also find "BAR", "Bar", "bAr" and any other combination of case.

grep and egrep can be useful tools for more than just searching for text and looking at whats found. They can be used as the basis for useful one line script utilities. For example if you had a directory called ~/Mail/ in your home directory that contained many files with email stored in, and you wanted to know how many messages you had overall you could do the following:

```
% egrep -r "^From .*" ~/Mail/* | wc -l
```

This relies on the knowledge that all mail messages in mbox formatted mail files (which is the most common format for email programs) start with a line that matches the patter "^From .*" (i.e. the word "from" at the start of a line then a space, followed by any number of any characters). This is generally followed by an account name, and a timestamp and is whats known as the envelope header (see "`man mbox`" for more). What this does is search recursively for all lines that fit that pattern ('^' means "start of the line") in all files (the '*' means all files inside the ~/Mail/ directory) then count the number of lines of output that egrep produces using wc. Since it'll output one line per email this will give you a total number of emails in total.

egrep has a `-c` (dash c) flag that means it will only output the total number of matches, and not the matches themselves. However since the example above was counting the output from multiple mailboxes (every file in ~/Mail) then this wouldn't have worked. However to count the number of mails in a single mailbox you should be able to use:

```
% egrep -c '^From .*' $MAIL
```

Which will match the same pattern as the example above, but will only do it to the mailbox contained in the variable $MAIL (your system inbox) and will print out the number of matches, instead of matched lines (the `-c` option).

11.1.4 head/tail

head and tail have already been encountered in passing (see section 10.1 for examples with them in). Essentially tail can get you the last n lines (defaulting to about 20) of a file or a programs output, head does the opposite, getting you the first N lines. These programs are handy to filter output, but tail also has another good feature:

```
% tail -f filename
```

The '`-f`' option means that tail will output the last few lines of the file "filename" then will stay running and any new input that goes into it will be outputted to the terminal. This makes it handy to watch log files generated by a programming running in another terminal as they are created.

11.1.5 wc

Word Count was detailed in section 5 but has another often overlooked behaviour. Since it can count lines of a file it can be attached to the end of a pipe-line of other commands to count how much output there is. This will allow you to count the number of occurrences of things, as illustrated by the following example:

```
% grep foo filename | wc -l
```

This will grep the file called "filename" for lines containing "foo" and instead of outputting them will simply count the number and print that. This is mostly handy for auditing log files.

11.1.6 bc

bc is an "arbitrary precision arithmetic language", basically it is a calculator for the command line, and can be used in scripts. While its operation is actually rather complex (see "`man bc`" and "`man dc`" for a related program) it can be used quite simply as the following example illustrates:

```
% echo "3 - 2" | bc
```

Echoing simple strings into it that contain normal arithmetic symbols (see "`man bc`") results in it returning the results, here it would print "1" to the command line.

If you are using the bash shell (see section 13.12) you could alternatively do:

```
% echo $(( 3 - 2))
```

To use its inbuilt maths functions. This is generally fractionally quicker from within bash as it doesn't need to invoke a seperate `bc` process.

11.1.7 sed

sed is a "Stream EDitor" it is essentially an editor that is designed to change parts of "streams of bytes" (see section 10.1) as it passes. Essentially you can write regular expressions (see 11.1.3) to spot patterns then change them to something else.

A simple example of this would if you wanted to see a list of all the hosts that are being used to remotely log into a machine:

```
% who | grep '(' | sed -e 's/.*(//' -e 's/).*$//' | sort | uniq | less
```

So what this does is runs who (to see who's logged in), and if you look at the output of who you'll notice that the last column contains the hostname of where that user is logged on from, which is surrounded by two brackets in the format "(hostname)".

So the first thing to do is pass it to grep, and grep for '`(`' which is every line with the an open bracket character '`(`' in. After this point only the lines which deal with remote users will be shown.

Then the results of that grep are passed to sed, and sed executes two commands to replace text, each is proceeded by a -e argument, and the whole replacement is shown in single quotes `' '`.

The first one reads "`s/.*(//`" which means a substitution (the 's') of the first part (contained inside forward slashes) for the second part. The first part is ".*`(`" which means anything followed by a `(` symbol. The second part is just //, since there is nothing between the two forward slash then ".*`(`" is replaced by nothing (i.e. it is deleted).

The second replacement reads "`s/).*//`' which looks for a `)` (closing bracket) followed by anything '`.*`' and replaces it again with the nothingness between the two forward slashes.

Now that we've just got a large list of hostnames these are passed to the "uniq" program (see section 11.1.17, note they also are passed through "sort" (see section 11.1.9), as uniq requires sorted input) which ensures there are no duplicates in the list, and passes its results to "less" for viewing.

11.1.8 tr

tr is used to "translate characters", that is to do read a stream of input and swap one character for another (or sequences for other sequences). Anything that tr can do can also be done by sed (see section 11.1.7) but tr can often do it more easily. For an example there is often a command called "users" that prints out a simple list of all the users logged into a machine, separated by spaces. However cent1 doesn't have a users command, so we have to make our own:

```
% who | awk '{print $1}' | sort | tr '\n' ' '
```

What this does is get the who list, pass it through awk to get the first column only (the username) and pass that through sort to alphabetise it. All that should be fairly familiar from earlier examples. However it then passes it through tr.

What tr does is exchange the characters inside the first quotes for those in the second. In this example there is a "`\n`" inside the first quotes, which generally means "a new line", the second quotes simply contain " " (a space). So tr simply gets all the input from sort and swaps all line endings for spaces, thus giving us our space separated list.

tr's simple syntax allows for quick editing jobs on the command-line where only very simple things need to be changed. For example if you had a file full of data, all of which was separated by tab characters, and you needed to separate them with comma's (tab and comma separated files are actually fairly common) you could use:

```
% tr '\t' ',' < tabbed-file > comma-file
```

This will input the file "tabbed-file" into the standard input of tr (see sections 10.2 if this syntax is confusing). tr will then change every tab character (shown by the standard escape '
t') into a comma and then send the output to comma-file.

Another useful feature of tr is the '`-d`' flag. This is used to simply remove characters instead of exchanging them for another character, as shown in this example:

```
% ls -l | grep "^-" | wc -l | tr -d " "
```

This prints out the number of files in the current directory. It does this by doing an '`ls -l`' to get the long list of files in the current directory, grepping that for a '`-`' (dash) character as files start with that (directories start with 'd'). It then word counts the number of lines (thus the number of files) and finally uses tr to simply remove all the space characters, thus making the number of files not have the space prefix that wc adds.

As a final example of using `tr` lets return to the example of `sed` above. The same thing could be done using `tr` as follows:

```
% who | grep '(' | awk '{print $6}' | tr -d '()' | sort | uniq | less
```

Here you can see it greps the output of `who`, gets the last column with `awk`, removes the brackets with `tr`'s `-d` flag, sorts it, removes duplicates then makes it viewable in `less`. Learning when its best to use `tr` and when to use `sed` is generally a matter of experience, but for simple things like this `tr` is often better, sed however is far more powerful and better suited to anything even a little more complex.

11.1.9 sort

`sort` has already been encountered in numerous examples, but to reiterate it simply takes input in, sorts it either alphabetically or numerically and then outputs it. Its handy for formatting output before showing it to the user, or for sorting lists that then get head or tailed.

sort also supports a '`-r`' option to reverse its behaviour, and its man pages contain details of how to create more complex sorting rules.

`sort` supports a rather useful feature for frequent scripters. With the use of the '`-u`' flag (unique) then it will function the same as using "`sort | uniq`" and print only the first occurrence of clumps of similar results. Using "`sort -u`" is the more efficient method as it saves you having to run the separate uniq process.

11.1.10 date

`date` does exactly that, it outputs the current date in a standard manner (for example, the current date is: "Fri Aug 6 16:46:01 BST 2004"). However date can be used to get other formatted time, for example:

```
% date +%H:%M
```

Will print out the hour in 24 hour format (that's the %H) then a : then the minutes (that's the %M). The rule of thumb is that you need a + before the string describing the output, and any symbols inside it preceeded by % (percent) symbols will be replaced by what they mean e.g. %H for hour, %M for minute, %B is the full month name.

"`man strftime`" contains the full list of % codes for date (usually "`man date`" would, but with Solaris this is not the case).

11.1.11 diff

`diff` is used to find differences between two byte streams (either two files, or a file and the standard input to diff). The idea is that it if you have two very similar files (say you copied a file, then made changes to the copy) you could use diff to spot the differences without needing to do it by hand, or write a horrible script to compare them line by line. diff's use is fairly simple, just use:

```
% diff foo bar
```

And `diff` will print out the differences between the file "foo" and the file "bar". Stuff from the first file will be prefixed by '`<`', and stuff from the second file prefixed by '`>`'.

See "`man diff`" for full details.

11.1.12 tee

`tee` is a useful command to know about in scripting, as it can be used to dump the results half way through a long command line (amongst other things). Primarily it can be used to write the input it gets to both a file and its output. For example:

```
% who | tee who-list | grep $USER
```

This runs who as usual, and hands the output of who over to tee. tee writes all of its input to the file "who-list" and then outputs all of it to grep, grep then greps for your username. So the user will see only their own user-details outputted to their terminal but the full who list stored in a file in the current directory.

Again, see the man pages before using this program.

11.1.13 md5sum

Assuming you want to be sure that one file is exactly the same as the other you could use diff and see if it returns anything, or you could use a program called md5sum. This will generate checksums of the file using the md5 algorithm. This guarantees (for all practical purposes) that if the two checksums are the same then the two files will be.

Its use is fairly simple, for example to generate the checksum for a file called "filename" type:

```
% md5sum filename
```

You will then see a very long string of numbers and characters followed by some space, then the filename. This is the checksum.

11.1.14 xargs

`xargs` is a program you've already seen the use of (see section 10.3). Essentially what it does is take a program as an argument, then make its standard input the arguments for that program. While this may sound a little confusing consider:

```
% ps -U $USER | grep vim | awk '{print $1}' | xargs kill
```

What this does is search the processes list for all your processes (see section 7.4), grep that for a specific program (in this case vim), and pass that list to awk to print out the first column, which will be the PID (see section 7). So at this point xargs comes into play.

`xargs` has a single number for its input, which is the PID of the process you want to kill (this example has a flaw in that if there is more than one processes running by that name it'll probably not work) and its one argument is the string "kill".

This means that it will run the command "kill" and give the kill command its input (the PID) so kill will then issue a TERM signal to that PID, and kill the process.

As ever see the man pages, xargs can be helpful in getting around problems of flow in scripting.

11.1.15 split

split is a program used to literally split a file, or its input, into sections. It outputs each section to a file called the same as the originally file followed by "aa" and so on. It can divide files by bytes, kilobytes, megabytes or line count (for textual files). See "`man split`" for details.

`cat` (see section 4.2.8) is probably the easiest way to reassemble them afterwards.

11.1.16 time

If you're ever interested in how long a command takes to run you can simply use the "time" command, as in this example:

```
% time who
```

As told in its man pages time will output the "real" time the command took to run, then two times relating to the CPU time used, see the man pages for details.

11.1.17 uniq

`uniq` is a very simple utility, shown already in multiple examples. It simply takes its input (which must be sorted) and removes any duplicate lines it encounters. This is handy for situations where you only want the output to contain one instance of that thing (e.g. a username list).

The man pages contain its full instructions, it can optionally display a count of how many times each line was in the input, or only output non-repeated lines.

As shown in section 11.1.9 the program sort can perform the job of a "`sort | uniq`" pipeline by the use of "`sort -u`", and for this reason you will more often use `sort -u` than uniq. However if you have a program that generates already sorted output then uniq can be helpful.

11.1.18 join

The join command is similar to the relational database operator (for those of you familiar with that), essentially it can take either two files, or one file and its input and run a "join" against them, only outputting lines that are in both. For this reason it's important that both files (and any input involved) has been passed through "sort" (see section 11.1.9) previously, as otherwise ordering will cause this command to be useless.

See "`man join`" for more information.

12 Other useful commands and information

12.1 Changing user accounts

Although you will only have one Unix account from the University sometimes it might be necessary, or convenient, to change which user account you are logged into in the middle of a session (If for example someone is logged in and you need to log in to quickly show them a file or something).

While it is always preferable to just open another login to the Unix server sometimes you will want to just change to another user account. The command which does this is called "su". And its usage is fairly simple:

```
% su - newusername
```

Where newusername is the username you'd like to log in as. Be warned that not writing a username here will prompt you for the root (administrative user) account's password. And all attempts to log in as root on unix.lancs.ac.uk are logged (and Admins will want to know what you were doing trying to gain control of their machine). You have been warned.

The `-` (dash) flag is generally used when using su as otherwise while you will change over to the new account your environment will be preserved, will generally causes issues. Use of the `-` flag means you will have the same environment as if you logged in as the other user instead of using su.

12.2 Finding out about your local Unix system

There are lots of commands that we've seen output information about the currently running system, ps and who etc. But these are more concerned with the processes and users, who could be logged into any type of Unix. There are two more commands that are helpful for finding out about the machine you're logged into.

`uname` prints out the name of the operating system running on the current machine. This can be handy, but often isn't very helpful as you will know which machine you are logged into (it can be handy for putting in scripts that will run on multiple machines however). However uname can give you a lot more information about the machine, for example:

```
% uname -a
```

While most of this is useless to the average user it can prove quite interesting if you're interested in the system you use.

Another interesting command to know is one that will tell you how long the system has been running, thus if you were wondering why a long process you were running died (see section 13.6 for a more likely explanation), or a number of other things, simply type:

```
% uptime
```

12.3 Links, symbolic and hard

Several times throughout the file management section the concept of links was mentioned (see section 4). They aren't essential to the use of Unix but will certainly enhance your ability to use the system, as they're quite handy, there are two types, both created by the `ln` program.

Both can be seen by using "`ls -l`", symbolic links will contain the filename followed by an arrow symbol "`->`" then the path that the link goes to.

Hard links (since they can only be files) can be seen by value in the second column of "`ls -l`" which is the reference count, this is equal to the number of hard links pointing to that inode.

12.3.1 Symbolic Links

The first are symbolic links, these can be thought of as pointers in the file system. A simple example might help to illustrate:

```
% ln -s ~/stuff/foo bar
```

This will create a symbolic link that goes between the file "~/stuff/foo" to a file called "bar" in your current directory. Now whenever you do anything to bar (such as view or edit it) it will affect "~/stuff/foo". However it is a separate file in its own right, so deleting bar doesn't affect foo (however if you delete foo then bar will become a "broken link" pointing at nothing).

You can form symbolic links from anything to anywhere, you can even link other directories to somewhere else, so a common trick is to place a link from your $scratch directory and $WWWHOME directories into your home directory for ease of use.

12.3.2 Hard Links

Hard links are a little more complex, and it's worth reading "`man ln`" for the full description of them. Essentially a hard link is a separate directory entry that points to the same inode (an inode is essentially the bit of the disk that contains the file's data). At this point there are simply two file names that point to the same data, and thus they are literally indistinguishable.

It's worth noting with hard links that the data pointed to isn't deleted until the last hard link is gone. So be very careful with them.

12.4 screen

The screen program is a "terminal multiplexer". What this means is that using screen you can log into a machine once and then run multiple shells on it at the same time (Which gives you one for mail, one for news, one for a remote login and one for random commands for example). This is handy as it reduces the number of logins you need to a single machine (thus reducing load on that machine) and also it means if your connection dies you can simply log back in and reconnect to your screen, and all your programs will be running the same as before.

The full documentation for screen is available via "`man screen`" and
http://www.gnu.org/software/screen/

As a quick introduction screen can be started simply by typing:

```
% screen
```

You will then see a quick message before you get your shell back. Except thats not the shell you had before, it's your first shell inside screen. In its default state screen commands are prefixed with `^a` (Hold down control and press '`a`'). The most common ones you'll want are:

`^a c` (in other words, hold control and press 'a', then release those two keys and press the 'c' key). This will create a new shell inside screen.

`^a w` This will list all the shells you have open in screen. The one you are currently in will have a * before its name, ones with new output you haven't seen will have a @ symbol in.

`^a n` Go to the next shell.

`^a p` Go to the previous shell.

`^a a` Will actually send your shell a control+a combination.

`^a ?` Will print out all the keyboard commands that affect screen.

To quit screen simply close all the shells in the normal way (with the "`logout`" command or "`^d`").

Screen is controlled by a file called "~/.screenrc". The two most important entries are how to change the key that controls it to something else (its normal position `^a` gets in the way of some software, such as emacs). And giving it a permanent status line.

To change the key that controls it you simply need to put a line like the following one in your .screenrc

```
escape ^nn
```

Where 'n' is the character you want, popular choices include 'o' and 'r' (although ^r also clashes with some things).

To make screen display a status line, listing all the currently running screens, you will need to add the following line in your .screenrc

```
hardstatus alwayslastline "\%Lw"
```

To change screen back to its default behaviour simply delete these lines from your .screenrc

12.5 talk and write

There are several ways to contact another user logged into the same machine, firstly if they're in the same lab as you, (check who, and also look around) you can simply go and speak to them. Secondly if you don't need them straight away you can email.

However if you want their direct attention you can use "`write`" or "`talk`". write is used to send a message directly to the terminal (see section 6.4) someone is logged in at, not their username or their shell. For this reason it's advisable to use who and make sure you've got the right terminal.

If you want a more one-to-one real time discussion you can use `talk`, (actually its best to use `ytalk`). This will enable you to chat with another user. Be aware however that things typed into talk are not buffered until you press enter, the letters appear as you type.

Users may however elect to disable the ability for others to contact them like this, by the use of the mesg command (see section 6.9.5). Other methods of generally getting in contact with users on the same machine as you include LuBBs (see 13.11).

12.6 Looking at binary files

Sometimes you will have a binary file (i.e. one not composed of plain text) that you want to see the contents of, often to try and work out what it does. The best program for this is `strings`. This simply goes through a file (any file) and extracts all the parts that would display as ASCII characters in your terminal. Using it is very simple:

```
% strings filename | less
```

This will use strings on the file called "filename" and pass the output to less for viewing. This can be used on any file, although the output is often not very interesting it can be handy for analysing binary files.

Another program that is quite useful for working with binary files is `od` which stands for "octal dump". This program can be used to print out the contents of a file in octal codes (or hexadecimal, or as integers). For example:

```
% od filename | less
```

Will print the contents of filename where each "word" (16 bits) is translated into an octal number. It will also print the offset from the beginning of the file for each line of these. See "`man od`" for more information.

12.7 Scheduling commands to run later

Sometimes you may want to run a command regularly, such as running fetchmail to get your mail every half hour or so (Remembering that fetchmail is part of homegrown and thus subject to a number of restrictions, see section 13.1) or you want to schedule a single command to run at a later time. For these two situations there are two commands. The first is cron, the second is at.

12.7.1 cron

cron is the name of a scheduling daemon that checks its crontab every minute and starts any jobs that need running. It is designed to be used for jobs that are scheduled every X interval (such as every 10 minutes, once a day, every Monday, once every 6 months), and this service is made available to regular users of the system.

```
% crontab -e
```

The above command will open a file in your $VISUAL, this is your crontab and has to be formatted according to the rules laid out in "man cron". Essentially it is five fields, minute, hour, day, month and day of week. Any of these fields may continue the values that should trigger it or a single '*' as a standard wild-card. This is followed by the command you want to run. An example will clear this up:

```
0,15,30,45 * * * * date >> ~/date-stamp
```

This will execute the command "`date >> ~/date-stamp`", echoing the current system date into a file in your home directory, this is fairly useless but serves as an example only.

If you look at the format of the columns before it you'll see the minutes column has the entry "0,15,30,45" and every other column has a wild-card. This means the command will run at those times past the hour on every hour, of every day, of every month.

If you wanted to run a backup script to scp a file of data from your home directory to another machine, and run it once a week at 9:00am on a Monday morning the command would look like this:

```
0 9 * * 1 scp ~/vital-data frank@melody.lancs.ac.uk:~/
```

This would run at the 0 minute, of the 9th hour on any day of the month, and any month of the year that was also a Monday, so on otherwords every monday at 9am. Again would make use of the fictional account "frank@melody.lancs.ac.uk".

12.7.2 at

at is used to schedule more one off jobs, and its operation is a little different from cron. The full details are contained in "man at" but a short example follows:

First you need to invoke at and tell it the time the program should run at, so the first line you should do will be:

```
% at 18:58 August 6th
```

This will schedule the job to run at that time and that date (you can also use "now" to mean right now, but the applications beyond testing of that are limited). at will then present you with a prompt that looks like this:

```
at>
```

The idea is that you then type in the commands you want executed, one at a time, pressing enter after each of them. Once you have entered the last command press `^d` (control and 'd'), and you should get your prompt back.

At will at this point run the command at the specified time, and mail the results back to your user account, however at has a few other functions that come in handy:

```
% at -l
```

The '`-l`' (dash ell) flag will list all the jobs you currently have scheduled with the "`at`" command.

```
% at -r 976385710.a
```

The '`-r`' flag will remove a scheduled job from the list, "976385710.a" is an example of the kind of job ID that "`at -l`" will give you.

13 Localised Advice

The following sections detail, or bring attention to, the localised setup of unix.lancs.ac.uk and how it differs from other Unix systems out there, it also contains quick details of useful local programs.

13.1 homegrown

As mentioned in many other places in this document there is a selection of programs installed in what is called the "homegrown" section of the Unix server.

Programs that are in "homegrown" are not officially supported by the Unix admins, instead they are supported by the person who installed them, to quote from the documentation:

> Be aware that if you install software in this area then you are expected to be able to provide support to any users who have problems with the software which you install

Also use of these programs that annoys the admins in any way may get your account suspended (such as creating a lot of email bounces with fetchmail). Full information is available with the command:

```
% less /software/homegrown/doc/about_homegrown.txt
```

In order to use the homegrown software (which you will probably want to as it contains an awful lot of really useful programs) you need to run a command to adjust your shell environment variables. This command is:

```
% /usr/local/homegrown/bin/setup_homegrown.csh
```

Typing this every time you login can become tedious, and so to get around this problem simply add the following line to your .login:

```
source /usr/local/homegrown/bin/setup_homegrown.csh
```

Be aware that the concept of a "homegrown" area is not common to most Unix systems.

13.2 Changing your account details and forgetting your password

If at any point you need to have something changed about your account, the name being wrong, your groups (see section 2.4) being wrong or even that you have finished your Undergraduate years but will be staying on to do postgraduate studies there is one port of call for getting all this sorted.

ISS Reception (which is opposite the Librarians, ground floor of the library) can deal with all these matters.

Also if you forget the password for your account, which does sometimes happen, then ISS reception can change it to a new one (at which point you should then change your password to something else, see section 2.5). They cannot find your old password however, as for security it is held in an encrypted form.

13.3 Password Synchronisation

When your account at Lancaster is created the passwords between your Unix account and Windows account are synchronised, so if the Windows password is changed the Unix password will also change.

However some people would rather have different passwords for Unix (and also Wing) and Windows. This can be done with the use of the `sync_passwd` script.

To determine if your passwords are synchronised or not simply use:

```
% sync_passwd show
```

Then this can be toggled by typing:

```
% sync_passwd
```

13.4 procmail and the config moving monster

See section 8.5 for information about procmail, and how the local system is setup to deal with .procmailrc files.

13.5 scratch

See section 4.12 for information about the "scratch space" available to you. Please note that again this is an additional feature of unix.lancs.ac.uk and is not standard to other Unix systems.

13.6 Process slaying

If you attempt to run very long jobs, or leave screen sessions running unattached for extremely long times, you may notice that your processes get killed and you receive a mail from root. This is the so called "process killing monster" that runs on Cent1 and keeps processes from running for too long and getting in the way of other people working.

Note that processes will be killed after about 24 hours, if you need to schedule a job that's going to run for longer than that you should go and speak to the Unix admins (see section 13.13).

13.7 Getting to central files

The "lan" command that enables you to get to your space on Central Files (see section 4.10) is a localised command and thus information about it won't be available via Google.

13.8 ∼clinch

There are a number of programs installed in various people's home directories. One user has taken to collecting symbolic links to all of these programs in their own home directory, to make it easier for you to find them all. Consult
http://www.lancs.ac.uk/ clinch/almost-homegrown/ for details

13.9 Printing from Unix

The command that prints files from Unix systems is `lp`. This is documented in `man lp` and http://hedgehog:8080/help/printing.html

13.10 Printer Budget

To check your print budget on the Unix cluster simply type:

```
% spoobudg
```

That will print out a message informing you of how much (in pounds and pence) your username has. For additional printer budget visit ISS Reception on the ground floor of the library.

13.11 LuBBs (http://www.lancs.ac.uk/socs/lubbs/)

LuBBs is the Lancaster University Bulletin Board System, essentially it is an old-style BBS that runs entirely on the University Unix server. It features conferences which you can post to and read messages on diverse topics such as books, arts-crafts, computers, politics and current-affairs. LuBBs also has a system of real-time chat to discuss things with other members. Full details about its use and history are available from
http://www.lancs.ac.uk/socs/lubbs/

The short answer is that once you've setup your account for homegrown access (see section 13.1) you can just type "lubbs", it will recognise you are a new user and help you setup an account (be warned that your account will be limited at first until the SysOp has varied it, this will commonly take between 10 minutes and a few hours).

13.12 Changing your shell

As documented in section 6 the default shell for the Lancaster Unix systems is tcsh. While this is a perfectly serviceable shell for day to day use, and even scripting, there are other shells available that you may wish to use.

In the event of wanting to change shell, most people will want to change from tcsh to the "Bourne Again Shell" (bash). This is actually a fairly simple thing to do, at the bottom of your ~/.login file (after all the setenv lines for your environment variables, and probably sourcing the homegrown script (see section 13.1) simply add:

```
exec bash --login
```

And thats "dash dash login". Once the shell thats interpreting .login reaches this line it will stop and execute the bash shell (which will of course be in its path). The bash shell will start up and will read its own config file at this point (which is `~/.bashrc`).

Bash has several useful features, these being much nicer scripting control and also several useful features (such as `^r` to search backwards through the history for a previously typed command).

For more documentation on bash see:
http://www.tldp.org/LDP/Bash-Beginners-Guide/html/index.html
http://www.gnu.org/software/bash/manual/bash.html
http://www.tldp.org/HOWTO/Bash-Prompt-HOWTO/
http://www.tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html
http://www.tldp.org/LDP/abs/html/
`man bash`

The most noticeable differences will be that setting the prompt is slightly different, as you change an environment variable called $PS1 and the key to offer name-completions (see section 6.2) is the tab key by default.

13.13 Contacting the Unix Sysadmins

The Unix system administrators are available via email for any questions or requests you might have about the systems. If you send mail to `systems` from cent1, or to `systems` at lancaster.ac.uk then it should reach them.

Be aware that this is not a method of getting support for the Unix systems. Support can be gained from the ISS helpdesks (in the Library). The sysadmins do not like being bothered needlessly and might well not appreciate frivolous emails.

However if you have serious comment then thats the best method.

14 Sample configuration files

Sample configurations for some of the programs mentioned here are available from:
http://www.lancs.ac.uk/ tipper/luui/configs/

Please note that these are *SAMPLE* configuration files, they may do things you don't expect or make your software behave strangely, please try to understand them before use. Again no responsibility will be taken for mistakes made using these config files.

15 Quick Reference

File Management

Name	Purpose	Example	Explanation
ls	Lists files in the current directory	`ls`	
pwd	Prints the current working directory	`pwd`	
cd	Changes the current directory	`cd foo/`	changes to the directory “foo”
file	Describes the contents of a file	`file foo`	prints a message about what type of file “foo” is
cp	Copies files	`cp a b`	creates a copy of file ‘a’ called ‘b’
cp	Copies directories	`cp -r foo bar`	Copies the directory foo to the directory bar, copies all files and directories inside it too.
mv	Moves files	`mv a ./foo/`	moves file ‘a’ into the directory “foo”
mv	Renames files	`mv a b`	moves the file ‘a’ to be called ‘b’
rm	Removes (deletes) files	`rm foo`	deletes the file called “foo”
rm	Removes (deletes) directories	`rm -r foo/`	deletes the directory called “foo”
less	Views the contents of files	`less foo`	opens the file “foo” in less. Use space for next page, `b` for back a page, `q` to quit.
mkdir	Makes new directories	`mkdir foo`	creates a new directory inside the current one called “foo”
quota	Tells you how much disk quota you are using and how much you have left	`quota -v`	

Tar Archives

Name	Purpose	Example	Explanation
tar	Bundles many files together into one file called a "tarfile"	`tar -cf files.tar foo bar baz`	Creates a tarfile called "files.tar" that contains the files "foo", "bar" and "baz".
tar	Extracts files from a tarfile	`tar -xf files.tar`	Extracts all the files from "files.tar" into the current directory.
gzip	Compresses files to save space	`gzip foo`	Zip's the contents of the file "foo", removing the original file and replacing it with a file called "foo.gz".
gunzip	Uncompresses files	`gzunzip foo.gz`	Unzips the file "foo.gz", renaming the uncompressed version "foo" and removing the original zipped file.
tar	Stores many files into a tarfile and compresses them	`tar -czf files.tar.gz foo bar baz`	Creates a tarfile called "files.tar.gz" and stores the files "foo", "bar" and "baz" in it. It also passes the tarfile through gzip automatically.
tar	Extracts compressed tarfiles	`tar -xzf files.tar.gz`	Uncompresses "files.tar.gz" to "files.tar" then extracts all the files from it into the current directory.

Getting Help

Name	Purpose	Example	Explanation
man	Displays manuals for programs	`man ls`	shows the manual for the "ls" command
info	Displays info-pages for programs, an alternative manual system	`info ls`	shows the info page
apropos	Searches for a command by keyword	`apropos keyword`	Displays a list of documentation relating to "keyword" for example "networking" or "files".
whatis	Gives a summary of a program	`whatis ls`	Gives a description of what the "ls" program is for.

User Accounts

Name	Purpose	Example	Explanation
who	Tells you who's logged into the system	`who`	
who am i	Tells you about yourself	`who am i`	
finger	Prints out information about user accounts	`finger -l username`	prints out information about the user called "username".
groups	Lists what groups a user belongs to	`groups username`	Prints out the group names that the user "username" belongs to.
whois	Searches for Users by surname	`whois smith`	prints out the usernames and real-names of all users who's surname is "smith"
last	Shows the last N logins	`last`	

Documents

Name	Purpose	Example	Explanation
wc	Counts words in a document	`wc foo`	outputs three numbers relating to file "foo", number of lines, number of words and number of characters.
ispell	Spell checks a document	`ispell foo`	spell checks the document called "foo", creates a backup file called "foo.bak" first.
head	Displays the first few lines of a file	`head foo`	Prints out the first 10 lines of the file called "foo".
tail	Displays the last few lines of a file	`tail foo`	Prints out the last 10 lines of the file "foo".

Processes

Name	Purpose	Example	Explanation
ps	lists all the processes running for that login	`ps`	
ps	lists all processes running for all logins of a user	`ps -U username`	displays all the processes of user "username", this can be your own username.
kill	stops a process from running	`kill 17829`	Kills the process with the PID "17829". Process Identification Numbers are found with "ps".
top	Lists the largest processes running	`top -n 10`	Prints out the top 10 processes and a lot of information about the system.

16 GNU Free Documentation License