An Introduction to UNIX

Michael Rutter
mjr19@cam

Michaelmas 2006
(Minor updates 2009 – 2021)

Basic UNIX

One cannot teach in a few dozen pages all the UNIX you need in order to operate effectively in a UNIX-based environment. Time spent discovering more about UNIX, by:

- attending free courses (e.g. those provided by the University Information Services: `http://www.training.cam.ac.uk/ucs/`)
- reading on-line resources (including the computing section of TCM's intranet site!)
- reading books
- asking those who know more

is time well spent.

Time spent learning from those who know little is less well spent. The blind leading the blind rarely has an auspicious outcome.

The first part of this booklet, in large type, is pretty basic. The second half exists to convince you that there are (many) more useful tricks which are worth finding: it is not exhaustive!

CLI or GUI?

The terminal is a most useful and necessary application. It provides a command-line interface to the operating system. A command line interface (CLI) is generally more powerful and flexible than a graphical interface. And commands typed in a terminal can trivially be turned into a script which can then be run multiple times, or on a remote supercomputer via a queueing system, or set to run in the background whilst one logs out.

The terminal itself is a resize-able window, with a vertical scroll-bar, in which a *shell* or *command interpreter* runs. The latter obeys our commands and produces a prompt consisting of the machine name followed by the name of the current *directory*. (A directory is what the real world calls that which Windows and MacOS call a 'folder'.)

Many alternatives to the default terminal exist, all with slightly different features. All take much more memory and start-up time than the simple `xterm`, which dates from the 1980s. More modern alternatives include `gnome-terminal`, `xfce4-terminal` and `konsole`.

On many systems (including TCM!), not all installed programs have entries in the menu system, and the command line is the most obvious way to find them. The amount of typing required in a terminal is not that great once one understands the virtues of laziness (see page 5).

Simple File Operations

Files can be copied, moved (renamed) and removed (deleted) using the commands `cp`, `mv` and `rm`. Their names are listed by `ls`.
Be Careful!: deletion is irreversible!

```
pc30:~$ cp thesis.tex new_book.tex
pc30:~$ ls
mail            results.dat
new_book.tex    thesis.tex
```

It may be possible to recover files deleted from one's home directory from the backup. Please ask a CO.

Tidiness

It is best to place files in tidy groups in subdirectories, rather than having everything in one directory. The command `mkdir` creates a directory, and `rmdir` will remove one provided it is empty. The `cd` command changes the current directory.

```
pc30:~$ mkdir test
pc30:~$ ls
mail            results.dat     thesis.tex
new_book.tex    test
pc30:~$ cd test
pc30:~/test$ ls
pc30:~/test$
```

Trees

Directories form a tree: each directory has one parent directory, and may have multiple subdirectories. A filename is assumed to refer to the current directory. Other locations can be specified by forming a *path* using '/' to separate the components of the path, and '`..`' to refer to a directory's parent.

```
pc30:~$ ls -F
a/  results.dat
pc30:~$ cp results.dat a/results.dat
pc30:~$ ls -RF
.:
a/  results.dat

./a:
results.dat
pc30:~$ mkdir b
pc30:~$ cd b
pc30:~/b$ cp ../a/results.dat .
pc30:~/b$ cd ..
pc30:~$
```

`ls -F` distinguishes directories by placing a '/' after their names.
`ls -R` lists all subdirectories in a recursive fashion.
`cd` typed on its own returns one to one's home directory.

Laziness

You have probably discovered that the cursor keys allow you to edit the current command and recall previous commands in an intuitive manner.

Note also that pressing {TAB} when part-way through typing a filename will cause the rest of the filename to be filled in automatically if it is unique. This trick also works for command completion (type `his` then press {TAB}).

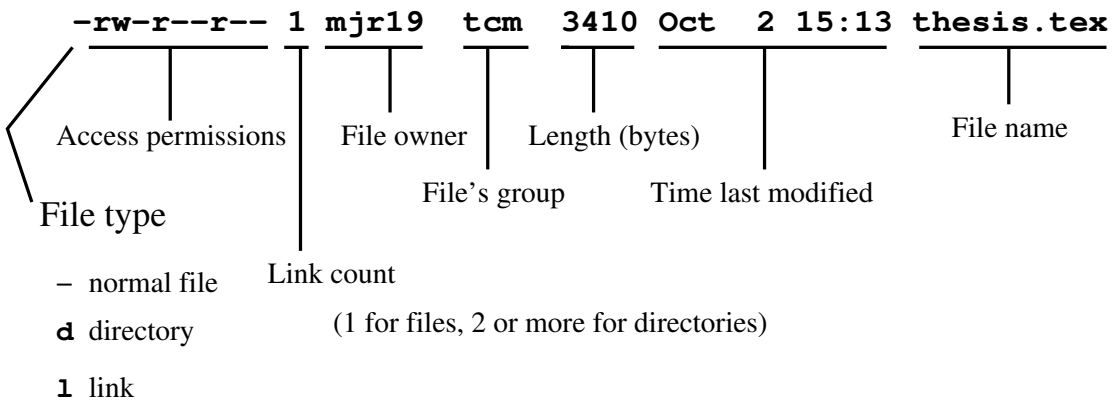
The *wild-cards* '`?`' and '`*`' can be used to stand for any one character, and any string of characters respectively:

```
pc30:~$ ls
a.dat  b.dat  results.dat  write-up.txt
pc30:~$ ls ?.dat
a.dat  b.dat
pc30:~$ ls *.dat
a.dat  b.dat  results.dat
pc30:~$ ls *
a.dat  b.dat  results.dat  write-up.txt
pc30:~$ rm *
pc30:~$ ls
pc30:~$
```

`rm *` should be used with considerable caution. `rm -r *` should be use with EVEN more caution as the -r means a recursive removal of files and directories.

More `ls`

```
pc30:~$ ls -l thesis.tex
-rw-r--r-- 1 mjr19  tcm  3410 Oct  2 15:13 thesis.tex
```



Access permissions:

	File	Directory
r	Readable	`ls` works
w	Writable	File deletion and creation permitted
x	eXecutable	Can access files and dirs within dir.

The first three characters refer to the file's owner.

The next three to people in the file's group.

The final three to everyone else.

By default, all files are readable by everyone. Other options make collaboration harder.

To prevent other people from reading a file, type:

```
pc30:~$ chmod go= new_book.dat
```

Note the space after the '=', and note the change this produces in the output of `ls -l`.

`ls -ld *` will list actual directories in long format, whereas `ls -l *` will list the contents of each directory in long format.

One Thing at a Time

The traditional UNIX philosophy is that a program should do one thing, and do it well (the Windoze philosophy often appears to be the opposite). This does often mean that one needs to invoke several commands to perform even a relatively simple thing. However, UNIX provides pipes, which enable one to feed the output of one command into the input of another simply.

There is a command, `less`, for displaying text files one screenful at a time. Hence almost no program offers this functionality: if you want it, use the program in combination with less!

```
pc30:~$ ls -l | less
```

here the output of `ls -l` is fed into the input of `less`.

The command also works to display text files:

```
pc30:~$ less thesis.tex
```

The | symbol is not the vertical line at the top left of the keyboard, but rather the (sometimes broken) vertical line to the left of 'z' on a UK keyboard. On a US keyboard it is found above the enter key.

The `less` Command

You may wish to be familiar with the following keypresses:

{space}	next page
{enter}	next line
d	scroll about half a page
/text	search for next occurrence of `text`
?text	search for previous occurrence of `text`
n	repeat previous search
i	toggle case sensitivity of searches
v	start `vi`
q	quit
{ctrl}L	redraw screen
b	previous page
j	previous line
u	reverse scroll c. half a page
G	goto end of file
numG	goto line number `num` (1G for beginning)
{ctrl}G	display current position in file

The name 'less' is a pun on the older and simpler UNIX command 'more': less is more than more! If faced with having to use more, be aware than reverse scrolling with pipes may not be possible, and commands such as 'G' might not exist.

Searches actually use *regular expressions* (see later). You may need to place a \ before some characters, especially '.','*','[' and ']'. If really stuck, replace an awkward character with a '.', for a full-stop will match any character.

To exit `vi`, after accidentally starting it, type '`:q!`{enter}'.

Links

One useful feature of the UNIX filesystem is its support for symbolic links. These are similar to Windows shortcuts, and MacOS's aliases. Unlike WIndows shortcuts, they are interpreted by the kernel, so behave consistently with all applications.

A soft link (also called symbolic link) is simply a file which names another file (or directory) to use whenever it is referenced. It is created with the `ln -s` command, which takes two arguments, source and target.

If in TCM one wishes to be able to move to one's `rscratch` directory as though it were within one's home directory, then

```
pc1:~$ ln -s /rscratch/spqr rscratch
pc1:~$ ls -l rscratch
lrwxrwxrwx 1 spqr tcm 14 Sep 21 18:03 rscratch -> /rscratch/spqr
```

is the answer. Similarly, if one wishes a 90MB+ Castep executable to appear in one's home directory in `~/bin` so that one can simply type `castep` to run it, then

```
pc1:~$ ln -s /rscratch/CASTEP/bin/8.0/castep-8.0_ifort14_mklfft \
          ~/bin/castep
pc1:~$ ls -l bin/castep
lrwxrwxrwx 1 spqr tcm 50 Sep 21 18:13 bin/castep ->
          /rscratch/CASTEP/bin/8.0/castep-8.0_ifort14_mklfft
```

But beware of surprises should the target be modified or removed.

Using `ln` without the `-s` will produce a 'hard' link. These are generally less useful and more confusing.

Using `rm` on a link deletes the link, not the target; `cp` copies the target, not the link.

The target does not have to be an absolute path. It can be relative, and it can contain '..'.

Note that the size reported for the link by `ls` is simply the number of characters in the target's name, which accurately suggests how a link is stored.

Text Editors

A text editor is not a word processor, and vice versa. Word processors break up lines spontaneously and concern themselves with the minutiæ of typography. This is not what a programmer wants.

The ubiquitous UNIX text editor is `vi`. It is based on a line-mode editor, and has an interesting user interface as a result. The most important thing to know about it is how to get out of it, and the answer is to press escape, followed by ':q!' and enter.

Although `vi` is fast, powerful, and possibly worth learning, most people prefer to start with emacs. Emacs is much, much more complicated (and slower) than `vi`, but is friendlier in letting one type and use the cursor keys in a natural fashion.

As for quitting emacs, {ctrl}X{ctrl}C is the answer.

Many alternatives to emacs exist: gedit and kate being two alternatives. They are less powerful than emacs, but look more like editors on MacOS/Windows.

For working with LaTeX documents, a dedicated LaTeXenvironment, such as texstudio, may be preferred.

Emacs

Emacs brings up its own windows if it can, and is best started with a filename specified on the command line.

```
pc30:~$ emacs my_prog.f90 &
pc30:~$
```

The final `&` ensures that one immediately gets a prompt back in the xterm. Otherwise, the prompt will not reappear until one exits the editor.

Emacs has a nasty habit of changing it behaviour subtly based on the sort of file it thinks you are editting, with different 'modes' for text, C, LaTeX, etc. These are meant to be helpful changes.

A process started with `&` is often referred to as a *background* process, as distinct from a *foreground* process which keeps full control of the terminal until it exits.

Emacs will use a text mode interface if it cannot produce its own window. If you get the errors such as '`Suspended (tty output) emacs`' or '`Stopped emacs`' type '`fg`' to make emacs a foreground process again, then use {ctrl}X, {ctrl}C to exit.

All sensible text editors can search (forwards and backwards), search-and-replace, goto specified line number, say which line you are on, discard changes made since last save, save file under new name, cut and paste sections, and much else besides. It is worth becoming familiar with your favourite editor. Poor choices of editor include `xedit` and `pico`.

Some Emacs Commands

(Many of these commands are also available from the menus.)

{ctrl}G	Cancel current operation
{ctrl}{Home}	Move to beginning of file
{ctrl}{End}	Move to end of file
{ctrl}A	Move to beginning of line
{ctrl}E	Move to end of line
{ctrl}K	Delete to end of line (**K**ut)
{ctrl}{space}	Set mark
{ctrl}W	Delete to mark (**W**ipe)
{alt}W	Copy to mark
{ctrl}Y	Paste last thing copied/deleted (**Y**ank)
{ctrl}_	Undo (recursive) (underscore, not hyphen)
{ctrl}S	Search (exit with {ctrl}G)
{alt}%	Search and replace
{ctrl}X 5 2	Open second window
{ctrl}X {ctrl} S	Save current file
{ctrl}X {ctrl} F	Open new file
{ctrl}X {ctrl} C	Exit
{alt}X goto-line{enter} n	Goto line n
{alt}X auto-fill-mode	Toggle automatic line wrapping

Using {TAB} will automatically complete the long {alt}X commands.

Emacs refers to {alt} as {meta}, and abbreviates it to 'M'.

Having two windows displaying different, overlapping, or identical parts of the same file works perfectly, changes in one are immediately reflected in the other.

Remote Display and X11

It is possible for a graphical application that is running on one computer to display on another computer.

When you ssh from one computer to another various magic can happen that allows X traffic to pass between computers so that:

```
pc25:~$ ssh pc2
pc2:~$ emacs &
```

will display emacs on the computer you are sitting at (pc25 in this example, with emacs running on pc2).

Remote Display outside TCM

`ssh` is the answer, whether from inside to out or outside to in.

However some graphical applications produce a lot of traffic so things might slow down due to the encryption overheads.

Over a slow link (e.g. mobile broadband), use `ssh -C`, which first compresses the data before encrypting it.

Some ssh clients don't forward X connections by default. They need the options `-X` or `-Y`.

(If you are addicted to VNC, this needs to be tunnelled over ssh and to have the server started as `vncserver -localhost`. If you don't know VNC, I tend to regard it is being too complicated to be worth investigating.

```
pc0:~$ vncserver -localhost -encodings "copyrect hextile"

laptop:~$ ssh -fNL 5901:localhost:5901 spqr1@pc0.tcm.phy.cam.ac.uk
laptop:~$ vncviewer ::5901
```

And even then it stores its password unencrypted in your home directory, which is far from ideal.)

A Process

A process is a single copy of a program which is running or, in some sense, active. The shell is a process, as is the xterm, the window manger, emacs, and even ls.

A process has resources, such as memory and open files, it is given time, *scheduling slots*, executing on a CPU with a certain priority, it has resource limits (maximum amounts of memory, CPU time, etc. it can claim). Lastly, it has a parent. Each process is associated with a single user.

These resources are exclusive to each process, and no process can change another's resources. Processes are mostly independent.

Each process has a unique *PID*, its Process ID.

A UNIX process has a current working directory, and a place for the three streams defined in C: `stdin`, `stdout` and `stderr`. To Fortran programmers, these are respectively the things which respond to `read (*,*)`, `write (*,*)` and the place that the 'floating point exception' error messages get written.

It also has a collection of *environment variables*. These are simply character variables of the form
`USER=spqr1`
(see also later)

Processes

It is usual to run several processes simultaneously: an xterm, a command shell, an editor, a compiler, a program you have written, the window manager, etc.

Occasionally it is necessary to exercise some direct influence on an individual process.

The command '`ps aux`' will show all the processes running on a machine. This may be overwhelming, so
`ps aux | less` or
`ps aux | grep uid` may be more helpful. To request that a process with a PID of 1234 exits, type

```
$ kill 1234
```

If that fails, try

```
$ kill -KILL 1234
```

though this gives the application no opportunity to shut itself down neatly.

If that fails, see your system administrator.

The command '`top`' produces a continuously-updated display of *active* processes. It is not a substitute for `ps` which can list all processes. Use 'Q' to exit from `top`.

`ps aux | less` causes the two processes `ps` and `less` to be launched 'simultaneously', and output to be transferred from one to the other, possibly (probably) after some buffering.

Note that you cannot stop processes you do not own!

uid is short for UserID/UserIDentifier.

The `grep` command extracts those lines containing the given text: see also later.

Redirection

Most Unix commands can have their output redirected to files.

```
pc30:~$ ls
results.dat  rubbish.dat
pc30:~$ ls > ls.out
pc30:~$ ls
ls.out  results.dat  rubbish.dat
pc30:~$ less ls.out
ls.out  results.dat  rubbish.dat
```

Use > to send output to a file, and >> to append output to a file. To redirect error messages as well, use `>&`. Unfortunately `>>&` does not exist.

```
$ ls -ld womble > output
ls: womble: No such file or directory
$ cat output
$ ls -ld womble >& output
$ cat output
ls: womble: No such file or directory
```

Used in combination with `&`, it is possible to start a process, log out and go home, leaving the process running.

```
pc30:/scratch/spqr1$ ./cgion.x >& SiC.out &
```

Note that `ls` is unusual in that it changes its behaviour when used with > and switches to single column output. You will not be able to reproduce the above exactly!

The `cat` command display the contents of a file by copying the named file to stdout. For this functionality it is much safer to use `less`, which copes correctly with binary files.

The non-obvious solution for `>>&` is '`>>output 2>&1`'.

File Transfer

The command `scp` provides basic file copying functionality.

```
pc3:/scratch/spqr1$ scp input.dat pc9:/scratch/spqr1/
```

The syntax of `scp` is is almost identical to `cp`, except for the addition of a machine name in either the source, or the destination. Like `cp`, one can specify multiple source files if the destination is a directory:

```
pc3:/scratch/spqr1$ scp *.dat pc9:/scratch/spqr1/
```

`scp` will prompt for a password if needed.

For anonymous file downloads from remote sites, http is probably the best answer: everyone has a WWW page. Use `wget` for command-line downloads. The older answer was `ftp`, which should never be used with passwords as it offers neither encryption nor challenges. The secure replacement for `ftp` is `sftp`.

See also the section on `tar` for more ideas.

Variables

All shells support two classes of variables. The most important, *environment variables* are passed on to programs launched from the shell, and the other class, *shell variables* are not.

Shell variables are used for defining your prompt, setting (or unsetting) automatic logout or mailcheck features, and other aspects of the shell's behaviour. They can also be used as a programming convenience.

To set a shell variable:
`x=5`

To set an environment variable:
`x=5 export x`

To see the result:
`echo $x`
(`echo` simply prints (echoes) its arguments).

One shell variable sets the default prompt string. Convention says that this string ends in '%' for `csh`, '>' for `tcsh` and '$' for `sh` and `bash`, but '#' for all shells if the user is root. However, modernists seem to be breaking this convention.

Several characters are treated specially, hence the standard TCM prompt of

```
$ PS1='\h:\w\$ '
> set prompt='%m:%~%#'
```

which gives prompts such as

```
pc30:~/talks$
```

For the full list of options, see the `bash` man page as appropriate.

Finding Programs

When a command is typed, it is first checked against the (short) list of shell *built-in* commands. If not found, it is assumed to be an *external* command, and is searched for by looking in the directories specified in the environment variable called `$PATH`.

The path is an ordered colon separated list of directories to be searched:

```
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin
```

For efficiency, Bourne-like shells remember where they last found a command, and never look elsewhere for commands they have found once.

Tradition places the subdirectory `bin` of your home directory on the default path, so that you can install and use programs merely by placing them in this directory (which you may first have to create with `mkdir`).

Bourne shells can be confused by commands moving, and '`hash -r`' is the solution.

. and /

The current directory, '.', is a special case for the path. If present it will always be searched without reference to hash tables. However, it shouldn't be present.

If the command name contains a '/', the path is not used, and the precise command specified is executed.

```
$ cat test.f
      write(*,*)'Hello'
$ f90 -o test test.f
$ test
$ ./test
Hello
```

All UNIX systems have a command called `test` already as a shell built-in function, so the first form will not execute the newly-compiled program whether or not '.' is on the path.

Should '.' be on the $PATH, and, if so, where?

Some believe it should be first: if someone puts a program called 'test', or 'wish' in their current directory, they clearly want that version executed, not the standard one. This is insane, as one then cannot do anything in a directory to which others have write access, for others can booby-trap commands there, inluding `ls`. If '.' is last on the $PATH then common command misspellings (e.g. 'ks' and 'mroe') can still be used for this trick.

Command Arguments

In the wonderful world of DOS, the first 126 characters one types including the command name are simply passed to that command, unchanged. The command is responsible for all the parsing.

UNIX is very different. The command expects its arguments to be presented as a list of *words*, and wild-card expansions, variable substitutions, division into words, and similar processing, are done for it. This has one clear advantage: whereas in DOS some commands understand how to process wild-cards such as '*' and '?', and some do not, in UNIX all behave in the same manner, because the processing is always done by the shell before the command is even started.

This interfaces directly with C's idea of `argv`.

```
$ cat > args
#!/bin/sh
echo "The first  argument is: $1"
echo "The second argument is: $2"
^D
$ chmod +x args
```

'`^D`' means type {ctrl}{D} – it indicates that no more data will be forthcoming. The `cat` command with no arguments copies stdin to stdout. Here stdout is redirected to a file, and stdin is the terminal input, so it acts as an extremely dumb 'editor'.

Shell scripts refer to their arguments as `$1`, `$2`, etc.

Note the use of `chmod` to turn this text file into something that can be executed. In this case, a script, the first line must start '`#!`' and then specify the program which is to run the script, here, `/bin/sh`.

Hello world

```
$ ./args hello world
The first  argument is: hello
The second argument is: world
$ ./args "hello world"
The first  argument is: hello world
The second argument is:
$ ./args         hello      world
The first  argument is: hello
The second argument is: world
$ ./args hello\ world
The first  argument is: hello world
The second argument is:
$ x=hello ; y=world
$ ./args $x $y
The first  argument is: hello
The second argument is: world
$ ./args "$x $y"
The first  argument is: hello world
The second argument is:
$ ./args '$x $y'
The first  argument is: $x $y
The second argument is:
```

Note the silent removal of excess spaces between words.

The double quote protects spaces from being treated as argument separators, whereas single quotes prevent any expansions.

The semicolon separates UNIX commands placed on the same line, just as it would in C or perl.

Hello, again

```
$ mkdir test
$ cd test
$ touch hello
$ touch world
$ ls -l
total 0
-rw-r--r-- 1 spqr tcm 0 Dec 20 9:37 hello
-rw-r--r-- 1 spqr tcm 0 Dec 20 9:37 world
$ ../args *
The first  argument is: hello
The second argument is: world
$ ../args ~ ~mjr
The first  argument is: /domus/spqr
The second argument is: /home/mjr
$ ../args '*' '~'
The first  argument is: *
The second argument is: ~
```

The character '~' is expanded to the home directory, and a tilde followed by a userid to that user's home directory.

Yet more examples can be found in the advanced section on page 39.

The `touch` command will create a file of zero length, or, if the file already exists, alter its last-modified time to the current time.

Wild-cards

Most people are familiar with the *wild-cards* '*' (any number of any character) and '?' (any single character), and the fact that neither will match a leading '.'. These are expanded by the shell, and are not passed to the program.

One can also specify a sequence of characters using square brackets.

```
$ ls
apple  Bill  pear
$ ls  [a-z]*
apple  pear
$ ls [A-Z]*
Bill
$ ls [a-mA-M]*
apple  Bill
```

This assumes the sane collation ordering of A-Za-z. If one suffers a system which uses AaBb-Zz, then `[A-Z]*` expands to all files starting with a letter other than 'z'!

Those using `bash` can quickly see different collation orders by typing
`$ LANG=en_US ls`
and
`$ LANG=C ls`

The 'traditional' order is that produced by `LANG=C`. Other settings ensure that things such as é, è, ê and e are all considered equivalent. Many programs assume a particular setting of the various LANG variables. Few are tested with all possible settings.

(LANG variables include LANG, LC_ALL, LC_COLLATE, LC_CTYPE, LC_MESSAGES, LC_NUMERIC. (LC abbreviates locale.) Some programs will ignore all of them, some will respond to some of them...)

Scripts vs Typed Input

A shell script is run in a separate process from the invoking shell. Thus any changes it makes to its environment are lost when the script exits.

To read commands from a file into the current shell, and interpret them as though they had been typed in, `bash` users can type '`source filename`' or use the older syntax '`. filename`'.

```
pc52:~$ cat silly
#!/bin/sh
cd /
pc52:~$ ./silly
pc52:~$ . ./silly
pc52:/$
```

Note that the prompt shows when the current directory of the shell changes.

There is more about shell scripts in the 'More Advanced UNIX' section, later in this booklet.

Printing

Three commands allow interaction with the print spooler:

`lpr` sends a file to a printer.
`lpq` lists files queued for the printer.
`lprm` deletes a job from the queue.

All take an optional argument of '`-P`' to specify the print queue:

```
$ lpr -Ppsc foo.ps
```

In general files sent to printers should be PostScript or plain text: printers don't understand pdf, gif, compressed PostScript *et al.*

Pipes can be used with `lpr`:

```
$ psnup -2 foo.ps | lpr -Pps
$ gunzip -c foo.ps.gz | lpr
```

Local commands may exist for turning on duplexing, printing to OHPs, and other special requirements.

The `psnup` command rearranges a PostScript file to fit several pages onto a single page, thus saving trees. This booklet was printed with
`$ psbook foo.ps | psnup -2 | duplex -land -Pps2`
which does rather more processing. The `duplex` command is unlikely to be found outside of TCM, and `psbook` reorders the pages suitably.

The `lpr` command rarely (never?) loses files, so careful use of `lpq` and `lprm` is generally better than approach of firing off multiple jobs until one is printed on a printer that you can find.

Sending EPS figures to a printer is often pointless: EPS is a fragment of PostScript (vaguely) akin to a subroutine – it is intended to be included in other documents, after suitable scaling and translation, and does does not in general print. Some applications produce EPS which is printable PostScript, and which is (almost) conforming EPS, so does print. Some don't. Commands such as `eps2ps` may exist.

Remote Commands

Using ssh you can trivially run commands on other UNIX computers to which you have access. Both ssh and rsh copy their standard input to the remote command, standard output of the remote command to their standard output, and the standard error of the remote command to their standard error.

```
$ ssh pc2 ls /
```

To print a document stored on your local computer on a remote computer:

```
$ cat foo.ps | ssh pc52 lpr -Pps2
```

This also works the other way around:

```
$ ssh pc52 cat bar.ps | lpr -Pps2
```

Remember about quotes and escaping characters? (`:0.0` is the local DISPLAY value, `localhost:12.0` the remote.)

```
$ ssh pc52 echo $DISPLAY
:0.0
$ ssh pc52 "echo $DISPLAY"
:0.0
$ ssh pc52 'echo $DISPLAY'
localhost:12.0
$ ssh pc52 echo \$DISPLAY
localhost:12.0
```

Configuration Files

Unix programs often store their configuration files in one's home directory, with names starting with '.'.

Often known as 'dot files.'

Such files are not shown by `ls` by default, nor does '`*`' match such files.

To find them, `ls -a` will serve.

`.bash_profile`	commands executed by bash login shells
`.bash_login`	commands executed by bash login shells (if no .bash_profile file is found)
`.bashrc`	commands executed by bash interactive non-login shells
`.config`	directory tree full of configuration for XDG-compliant applications
`.fvwm2rc`	`fvwm2`'s configuration
`.mozilla`	directory tree full of Mozilla and Firefox configuration
`.pinerc`	alpine's configuration
`.profile`	commands executed by [ba]sh login shells (if neither .bash_profile nor .bash_login files are found)

Many are text files which you are free to change, and there are many not listed above.

Rampant Customisation

People creating hundreds of personalised settings could be regarded as wasting their time. It is probably a bad path to follow, and copying large numbers of settings from other people without checking them can lead to all sorts of problems. You should not let people give you 'dot files' which you don't understand.

Maths

`bc -l` starts up a simple calculator (**b**asic **c**alculator) accepting things like:

```
pc30:~$ bc -l
(4+6)/5*2
4.00000000000000000000
quit
pc30:~$
```

Note the non-algebraic, but left-to-right, precidence.

Without '`-l`' it acts as an integer calculator. The only function is `sqrt`. With it, it acts as a fixed point (20 decimal places) calculator, and defines the functions `s` (sine), `c` (cosine), `a` (arctangent), `e` (exponential), `l` (natural logarithm) and `j(n,x)` (Bessel function).

```
pc30:~$ bc -l
4*a(1)
3.14159265358979323844
scale=40
4*a(1)
3.1415926535897932384626433832795028841968
scale=0
5/4
1
obase=16
255
FF
```

It can be used with pipes too:

```
pc30:~$ echo '5*7' | bc
35
```

There is also a GUI calculator, `xcalc`, but its lack of support for cut and paste is tedious. More modern alternatives include `kcalc` and `gnome-calculator`, but these are useless in scripts.

Manual Pages

The `man` command is probably the most important UNIX command: it displays the on-line manual, which will explain all the others anyway.

An individual page covers a single command, routine, or file. Some are a few lines long, and some (such as that for `bash`) many thousand of lines.

The pages are grouped into chapters, depending on the class of the item described. Important chapters include:

1	user commands
1x	X-based user commands
2	system functions
3	C functions
3f	Fortran functions (if present)
5	configuration file formats
8	administrative commands

The Synopsis section should give a brief summary of the syntax for the command and a summary of what it does. Things inside `[ ]` are optional, and the syntax [a | b] shows that a and b are mutually exclusive options. It is possible to nest the brackets and/or symbols.

Single-character options are often grouped, so that '`[-ab]`' means any, all, or none of the options, i.e., nothing, '`-a`', '`-b`', or '`-ab`'.

Linux's man pages tend to be somewhat patchy in quality – commercial UNIXes are often better.

Some man pages (such as `exit`, `printf`, `mkdir`, `cvs`, `crypt`) appear in multiple sections. The `whatis` command will display all appropriate summaries, whereas the `man` command may display just one, using a precedence order which is non-obvious.

To specify a precise page, one must specify the section too, as in
`man 3 printf`

Reading a man page

WC(1) FSF WC(1)

NAME

wc - print the number of bytes, words, and lines in files

SYNOPSIS

wc [OPTION]... [FILE]...

DESCRIPTION

Print line, word, and byte counts for each FILE, and a
[etc.]

The header line repeats the name of the man page, 'wc', and gives the chapter number in brackets. It may also give the author in the centre (Free Software Foundation).

The next line is very important: it is a one-line summary of the page, and this is the line which is used when searching for man pages, and which is returned by the `whatis` command.

```
$ whatis wc
wc (1) - Counts the lines, words, characters, and bytes in a file
$ man -k words | grep 1
wc (1) - Counts the lines, words, characters, and bytes in a file
```

The `whatis` command prints the one-line summary of a manpage.
The `man -k` command (equivalent to the `apropos` command) searches the whatis database for the keyword given. In the example above, the output is piped through `grep` to ensure that only answers containing '1' (i.e. from chapter one) are given.

More Advanced UNIX

A small, random selection of topics in greater detail follows.

Searching for Text - `grep`

The `grep` command is very useful for searching files. It will print every line which contains a given string:

```
$ grep 'TOTAL ENERGY' output.dat
 TOTAL ENERGY IS          -745.4575585
 TOTAL ENERGY IS          -783.9824520
```

with the '`-v`' option it will print every line which does not contain a given string:

```
$ ps aux | grep -v root
```

The `tail` command can also display from a file as it grows:

```
$ cgion.x >& output.dat &
$ tail -f output.dat
```

One can even use
`$ tail -f output.dat | grep 'TOTAL ENERGY'`
To stop `tail -f`, press {ctrl}C.

Some characters need quoting from the shell, and it tends to be safest to enclose the search string with quotes. For instance, if looking for running processes,
`$ ps aux | grep R`
is mostly right, but
`$ ps aux | grep ' R '`
will avoid any process with an 'R' in its name, and just match those with an isolated R (presumably the status column).

If one merely wants to count the number of matches,
`$ ps aux | grep -v root | wc -l`
certainly does the job. However, the '`-c`' option to `grep` is somewhat quicker and simpler:
`$ ps aux | grep -cv root`

Regular Expressions

Most people are familiar with the shell wild-cards '*' and '?' used for filename 'globbing'. However, the general syntax for wild-cards for matching text, as used by `grep`, `perl`, `vi`, `emacs` and many others, known as *regular expressions*, is rather different.

The character corresponding to '?', which matches any single character, is '.'.

```
$ grep 'independ.nt' /usr/share/dict/words
independent
```

The file `/usr/dict/words` traditionally exists on UNIX systems, and contains a list of English words, one per line. More modern UNIXes prefer to call it `/usr/share/dict/words`.

```
$  wc -l /usr/share/dict/words
   304495 /usr/dict/words
```

The characters '^' and '$' match the beginning and end of lines respectively:

```
$ grep 'pret$' /usr/share/dict/words
interpret
$ grep '^pret' /usr/share/dict/words
pretend
pretense
```

etc.

If one needs to search for a real '.', '^', or '$', it must be preceeded by a \. (LaTeX also uses this convention to 'escape' most of its special characters, as does the shell.)

```
$ grep '1.2' results.dat
 1.234
152
31.27
311423
$ grep '1\.2' results.dat
 1.234
31.27
$ grep ' 1\.2' results.dat
 1.234
```

Repeats and Ranges

The character '*' means any number (including zero) of the preceding character.

```
$ grep 'a.*e.*i.*o.*u' /usr/dict/words
adventitious
facetious
sacrilegious
```

Thus '.*' is the equivalent of '*' as a shell wild-card.

Square brackets denote ranges, just as for shell wild-cards.

```
$ grep -c '^[A-Z]' /usr/dict/words
4974
$ grep '[aeiou][aeiou][aeiou][aeiou]' /usr/dict/words
aqueous
Hawaiian
obsequious
onomatopoeia
pharmacopoeia
prosopopoeia
queue
Sequoia
```

(Words starting with a capital letter, and words containing at least four vowels in a row.)

`grep` and Regular Expressions

Most `grep`s offer extended regular expressions, enabled by specifying '`-E`'. These enable one to specify repeats more explicitly:

```
$ grep -E '^o.*[aeiou]{4}' /usr/dict/words
obsequious
onomatopoeia
$ grep -E '^a.{9,}d$' /usr/dict/words
aboveground
abovementioned
absentminded
aforementioned
```

One can also specify multiple expressions to match with extended regexps:

```
$ ps aux | grep -Ev '^root|^rpc|^lp|^exim'
```

More ideas

A ^ as the first character of a range negates the range (even for non-extended regexps).

```
$ grep -Ei '^[^aeiou]{6,}$' /usr/dict/words
rhythm
syzygy
```

(The '-i' makes the search case-insensitive, thus removing UNESCO from the answer.)

Find lines containing only numbers
```
$ grep '^[0-9+.eE-]*$'
```
(note '.' stands for itself with a range, and '-' for itself if it is the first or last character.)

Find lines containing more than 72 characters
```
$ grep -E '^.{73,}$'
```
or simply
```
$ grep -E '.{73}'
```

Find lines containing two or more adjacent capitals
```
$ grep -E '[A-Z]{2,}'
```

And read the man page...

More regular expressions

The search facility of `more` and `less` (and hence of `man`), and also of `vi`, is based on regular expressions. Hence one can get funny results if searching for a special character such as '.', '[' or '*'.

This can be avoided by preceding such characters with a backslash.

```
$ grep '\.' /usr/dict/words
e.g
i.e
Ph.D
U.S
U.S.A
```

Emacs offers both a fixed string and a regexp search.

It is worth learning a little about regular expressions: they can be very useful, and very many programs can use them: `awk`, `ed`, `emacs`, `expr`, `grep`, `less`, `more`, `perl`, `python`, `sed`, `vi` to name a few.

To perform replacements more complicated than single character substitutions, one needs to use `sed`.

```
$ sed 's/colour/color/g' <english.txt >foreign.txt
```

This finds all occurances of "colour" and replaces them with "color". Note that "Colour" will not be replaced. Without the `g` flag, only the first occurance on each line will be replaced.

Finding files: `find`

The `find` command finds files based on their metadata (not their contents). It can find by name, size, modification date, type, etc., and it will descend a tree starting at a given directory. Hence

```
$ find ~ -type l -print
```

will list all symbolic links in your home directory, and

```
$ find ~ -size +4m -ls
```

all files larger than 4MB.

N.B. Some `find` commands need `+4096k`, not `+4m`.

The first argument to `find` is the directory to start searching from so, if you wish to start at the current directory, a dot must be used.

```
find . -size +4m -ls
```

And *NEVER* try something like

```
$ find / -size +4m -ls
```

because this will search through all remotely-mounted disks too, which could be a significant fraction of a TB.

Instead use

```
$ find / -xdev -size +4m -ls
```

if you really must search the root filesystem. The `-xdev` flag will prevent find from moving across mount-points.

All finds have a flag with the functionality of `-xdev`. Unfortunately, some call it `-mount`, others `-x`, ...

The find command effectively evaluates a string of conditions, stopping when the first one evaluates to false. So

```
$ find . -name '*ps' -size +1m -ls
```

will list all files whose names end in 'ps' and which are over 1MB in size. The operator '`-print`' prints the current filename, and returns true. The operator '`-ls`' prints something like the output of `ls -l` for the current filename, and returns true. Use neither, and nothing may result.

Replace '`-size +1m`' by '`-mtime -8`' for all ps files modified in the last week.

(If you suffer from a `find` which does not support `-ls`, then
`find . -name core -exec ls -l {}\;`
is the answer.)

File Archives

UNIX's standard archive program, `tar`, produces a single file containing an archive of all its input files.

```
pc30:~$ tar -cf thesis.tar thesis
```

will produce an archive called `thesis.tar` contain the directory `thesis` (and its subdirectories). Add a '`v`' to the options to see each filename as it is added to the archive.

To examine the contents of such an archive, use

```
pc3:~$ tar -tvf thesis.tar
```

and to extract (which will over-write existing files)

```
pc3:~$ tar -xvf thesis.tar
```

and to remember those odd arguments:
c – create
t – table (of contents)
x – xtract

However, `tar` is rarely used on its own. It is often used for medium or long term storage, in which case it is usually used in conjunction with one's favourite compression program:

```
pc3:~$ tar -cf - thesis | gzip > thesis.tar.gz
```

and to reverse

```
pc3:~$ gunzip -c thesis.tar.gz | tar -tvf -
```

Here naming a file '–' stands for stdin or stdout as appropriate. The default file which `tar` will write to in the absence of any '`f`' argument is the tape drive (which probably doesn't exist). Yes, `tar` abbreviates Tape ARchive.

Gnu's version of `tar` accepts the option `z` to mean compress with gzip, so that one can use:

```
pc3:~$ tar -czf thesis.tar.gz thesis
pc3:~$ tar -tvzf thesis.tar.gz
```

but do not rely on all tars accepting this. (If z is not given, recent versions of Gnu's `tar` automatically detect most forms of compressed archive, and decompress appropriately.)

Another use of `tar` is for copying whole directory trees. This is much easier with Gnu's syntax:

```
pc30:/scratch/spqr1$ tar -cf - run1 | rsh pc20 \
  tar -C /scratch/spqr1 -xf -
```

to copy the contents of the directory tree `/scratch/spqr1/run1` on pc30 to the same place on pc20, or even

```
pc30:/scratch/spqr1$ tar -cf - run1 | tar -C /usb -xf -
```

which will copy the directory tree to `/usb/run1` on the same machine.

If suffering from a 'traditional' tar, these commands become

```
pc30:/scratch/spqr1$ tar -cf - run1 | rsh pc20 \
  cd /scratch/spqr1 \; tar -xf -

pc30:/scratch/spqr1$ tar -cf - run1 | ( cd /usb ; tar -xf - )
```

Shells

Shells and Redirection

The use of '`|`', '`>`', '`>>`', '`>&`' and '`>>&`' has already been covered. Assuming one is using a Bourne shell, it is also possible to redirect stdout and stderr separately:

```
$ ls -ld .
drwxrwxrwt 6 root root 4096 Dec 18 18:41 .
$ ls -ld . > output
$ cat output
drwxrwxrwt 6 root root 4096 Dec 18 19:48 .
$ ls -ld womble > output
ls: womble: No such file or directory
$ cat output
$ ls -ld womble > output 2>errors
$ cat output
$ cat errors
ls: womble: No such file or directory
```

Indeed, in the Borne shell the > syntax is really short for `unit no>`, with a unit number of 1 (stdout) assumed if none is given. The unit number of stderr is 2. (The C unit numbers are the ones which are relevant here, as UNIX is written in C. That to Fortran stdout is usually 6 is irrelevant.)

Note too that if no output is produced, the file to which output is redirected with > will be truncated to zero length, losing its previous contents.

One can redirect stdin too:

```
$ cat input.dat
7+5
$ bc < input.dat
13
```

This is very useful for running commands non-interactively.

Finally, there is a special file called `/dev/null` which simply discards anything written to it. This can be useful for throwing data away (though such tricks should be used with caution...):

```
pc30:~$ rm /womble
rm: /womble: No such file or directory
pc30:~$ rm /womble >& /dev/null
pc30:~$
```

Shells and Command Arguments

It is the shell which is responsible for expanding command arguments and passing them to the command. Hence here a few more examples.

```
$ x="hello world"
$ ./args $x
The first  argument is: hello
The second argument is: world
$ ./args "$x"
The first  argument is: hello world
The second argument is:
$ ./args '$x'
The first  argument is: $x
The second argument is:
$ ./args \$x
The first  argument is: $x
The second argument is:
```

Note:
variables expanded first, then result split into words.
variables in double quotes are expanded.
variables in single quotes are not expanded.
\ escapes the $ character.

Backquotes

There is one other sort of quote in 7 bit ASCII, the *backquote* or *tic*, usually found at the top left of the keyboard. Anything between backquotes is executed in a sub-shell, and is substituted by anything sent to stdout.

```
$ ls
hello  world
$ ../args `ls`
The first  argument is: hello
The second argument is: world
$ echo "3+4" | bc
7
$ ../args `echo "3+4" | bc`
The first  argument is: 7
The second argument is:
```

Note that stderr is not collected by the backquotes:

```
$ ./args `ls womble`
ls: womble: No such file or directory
The first  argument is:
The second argument is:
```

Shell Startup

Whenever a new shell is started, it sources certain files, depending on both the type of shell and whether or not it is a *login shell* (not by default). For `bash` the sequence is:

`/etc/profile` (if a login shell)
`~/.bash_profile` or `~/.bash_login` or `~/.profile` (if a login shell)
`~/.bashrc` if not a login shell

(For `sh`, as `bash` but without files containing 'bash' in their name.)

Scripts read on login may produce output ('Good morning, sir, you are 4KB below your disk quota again'). Scripts read by non-login shells may not – if they do, all sorts of oddities occur, especially with `rcp`, `rsh` and similar.

Bash's startup sequence is also slightly odd. Some systems are configured so that `.bashrc` is read by login shells too, and some so that a global `/etc/bashrc` exists.

Any shell is either a login shell, or the descendant of a login shell, so anything which will be inherited need only be set once in the files read only by login shells. Unfortunately many installations of X get this wrong.

Shell startup is depressingly common: many programs do it to expand wild-cards, or because a C library function called `system()` neatly does the `fork()`, `exec("/bin/sh",...)`, `wait()` magic one needs to launch another program and wait for it to finish. Thus simple shell startup needs to be fast. For `/bin/sh` it is: no configuration files read unless it is a login shell.

Needless to say, a file whose contents are executed every time one logs in is really quite important. Mistakes here can even prevent one logging in at all. (So, if you feel the urge to change one of these files, do test it by

logging in (`ssh` or `rlogin` to `localhost`) *before* logging out and then finding that you cannot get back in.)

Do also make sure that you keep half an eye on the contents. In TCM one can delete these files and still have a workable account. For systems where this is not true, tradition places minimal working examples in `/etc/skel`, or otherwise as advertised (TCM currently uses `~/../skel`).

Scripts in which shell?

Writing scripts to do common tasks can save much time, and the shell provides a simple language well-suited for manipulating files and jobs.

The first problem is to choose one's shell. The Bourne shell is not invariably the correct answer. One should not forget alternatives such as `awk`, `perl`, `python` and even `sed`.

There is also the matter of which of the various Bourne shell derivatives to use. Here we shall look at the lowest common denominator, which is therefore likely to lead to best portability. There are some useful extensions in other shells: `bash` supports basic arithmetic operations. However, one can live without.

The shell gets used for the very basic control structure, and almost everything else is done by external programs. The result is often somewhat inefficient, so really serious scripts should be written in `perl`, `python` or C.

Special variables

We have already seen that $1 to $9 contain the first nine parameters passed to the script. $0 actually contains the script name itself, as typed, and $# the total number of arguments.

The command `shift` moves all arguments, except $0, up one, with $1 disappearing, and the tenth parameter (if any) becoming $9, and $# decremented. It is an error to `shift` when $# is zero.

Other special variables include $? which returns the exit status of the last command, and $$ which returns the script's PID.

A variable name can be enclosed in {}, and must be if followed by an alphabetic character.

```
$ x=foo
$ echo $xbar

$ echo ${x}bar
foobar
```

Return Codes

Each process should exit with a code of zero if it has been successful, and non-zero if it has failed. As there are more ways of failing than of succeeding, there are more ways of expressing failure than success. One can usually find the return code of the last process by typing `echo $?`.

```
$ ls -d /
/
$ echo $?
0
$ ls -d /womble
ls: /womble: No such file or directory
$ echo $?
1
```

Here the `ls` command, a separate process, has produced a return code of zero when it has been successful, and an error message and a non-zero return code when it failed to do what was requested of it.

It is good practice to check return codes when writing shell scripts.

Simple `if` statements

```
#!/bin/sh
if cd $1
then
  echo We can change directory to $1
  if touch womble
  then
    echo And we can create files in it
    rm womble
  fi
else
  echo We cannot cd to $1
fi
```

Note that the condition expression is simply a command. A command which executes successfully gives a return code of zero, which is considered to be true. One which fails, non-zero, and false.

Yes, this is the opposite of most programming languages. Note too the use of `fi` for 'endif'.

`test`ing times

The command `test` allows one to test most aspects of file existence, and some string operations too. Common uses are

`test -d dir`: true if dir exists
`test -f file`: true if file exists
`test str1 = str2`: true if strings are equal
`test int1 -eq int2`: true if integers are equal

The command `[` is a synonym for `test`, and if invoked as `[` the expression must be terminated by a `]` preceded by a space. So

`if test -d /temp`
can equally, and more usually, be written
`if [ -d /temp ]`

For integer comparisons, `-lt`, `-gt` and `-ne` exist, and all comparisons can be negated with a leading '!', such as `[ ! -d /temp ]`

See '`man test`' for more.

`expr`**essing Oneself**

The `expr` command performs simple arithmetic and string matching functions. It is fussy about spaces, but more universal than `bc`, and it can handle strings. See its man page for full details.

```
#!/bin/sh
x=1
while [ $x -le 12 ]
do
  echo $x `expr $x '*' $x`
  x=`expr $x + 1`
done
```

It is tempting to use the `bash` arithmetic extensions, and to write the loop body as

```
  echo $x $(($x * $x))
  x=$(($x + 1))
```

If you are tempted, be sure to change the first line to read `#!/bin/bash`, rather than fall into the Linux `sh`-is-always-`bash` trap.

For loops

```
#!/bin/sh
for f in `seq 1 9`
do
  rsh pc$f status
done
```

used to display the status of TCM's first nine PCs.

Processes: their Life and Death

Parents are important at two points in a process's life. At birth, when it receives its inheritance, and at death, when it needs to inform its parent of its demise. Orphans are not permitted: they are immediately adopted, or

re-parented by a special process called `init`, whose PID is one, and which is responsible for many housekeeping tasks.

When a process dies, its resources are immediately freed. It will get no more scheduling slots on the CPU, its memory will be reclaimed, its open files closed, etc. Its final act is to inform its parent of its death. The main reason for having children is to get a specific job done. Thus a process needs to be informed when its children die.

The parent will receive a signal when a child dies, and then it must collect the final message from the child, which will be a *return code* indicating whether the death was voluntary or compulsory, and whether the process had a successful life. Generally a record is also written to the process accounting file when a process exits.

A child which has died but which cannot successfully communicate this fact to its parent becomes a *zombie*. Hence it is important for `init` to re-parent things.

Zombies are marked by a 'Z' in the output of `ps`. Killing zombies is not very important: they do little harm. When their wayward parents die, `init` can re-parent and remove the zombies.

Note also the meanings of the memory fields in the output of `ps`:

```
~$ ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
spqr1      558 94.4 17.3 6206120 1060292 pts/0 Sl+  10:02 244:34 matlab
```

RSS: Resident Set Size, amount of physical memory being used
VSZ: Virtual SiZe, upper bound on amount of physical memory wanted
%MEM: RSS as percentage of machine's memory

X11 and GL

The standard way of displaying accelerated 3D graphics on UNIX is via a system called GL (similar to Direct X on MS Windows systems). All our PCs support GL, with varying amounts being done in software or hardware.

For almost everything done in TCM, hardware acceleration is unnecessary. Relatively few applications use it at all, and even those which do (VMD, pymol, gdis, xcrysden, Mathematica) do so much other processing that even an infinitely fast graphics card would only increase the frame-rate by a factor of two to three in most cases.

The point at which the distinction between proper hardware acceleration, and software emulation, becomes important is when the hardware acceleration is buggy. At the moment, this is the case on rather a large number of TCM's computers. Fortunately most applications either do not trigger the bugs, or do so in a harmless way.

It seems to be harder to turn off hardware acceleration with recent X servers, and certainly the mechanism for doing so keeps changing. Rather than putting instructions which are likely to date rapidly onto paper, at this point it seems best to point to
`http://www.tcm.phy.cam.ac.uk/internal/computers/GL.html`

One unfortunate aspect of X11 is unchanging. If you exceed your disk quota, and then ssh between two TCM machines, you will find yourself unable to open any more windows until you correct the quota problem *and log out*. I assume this piece of poor design is left in the `xauth` program as an instructive lesson to others.

Compiling

Though the concepts are simple, it is amazing how many misconceptions survive.

Anyone sane writes code in a standardised language (C, C++, Fortran, etc), then relies on a compiler to translate it into the language of a specific CPU. That language, machine code, is just a heap of unreadable binary. Assembler is a human-readable language with a trivial translation to a specific machine-code, and potentially containing friendly label names which can be removed by an assembler. Attempting to write assembler is not bright unless one wishes to worry about the precise instruction set of today's processors, change it when tomorrow's processors come out, and to worry about the precise function calling mechanisms of your favourite operating system(s).

A compiler translates source code (C/C++/Fortran) to an object file which contains machine code for the functions and subroutines in the source file, and also any data sections required for constants. It needs to contain the names of the functions and global variables, but no information about the number of arguments (which may be variable), the return value, or any names of local variables, line numbers in the original source, etc. Debugging options may be available to include increasing amounts of information about local variable names and original source line numbers.

A library is simply an archive file containing multiple object files in a single file.

In C/C++, a header file (or include file) contains function prototypes, information about the arguments and return codes of functions. Without this the compiler cannot check whether calls to functions are correct, and one is living very dangerously (not least because a return type of `int` for all functions will be assumed). In Fortran such information ends up in module files, which not human readable, and are compiler-generated from the corresponding source file.

The final stage of building an executable is linking. An object file contains no information about how to find those functions which it itself does not define, and almost all programs have some, such as `printf`, `read` or `write`. It also lacks the necessary startup code which the operating system requires when starting a program.

Compiling and linking often happen with a single command, such as
`gfortran test.f90`
but that does not mean there are not two distinct phases, which are readily separated.

```
gfortran -c test.f90
gfortran test.o
```

Two things to be aware of. Firstly, compilers tend to read from left to right, so commands such as

```
f90 test.f90 -O3
```

are probably mistakes – compile `test.f90`, then turn on optimisation. What is wanted is

```
f90 -O3 test.f90
```

Also rubbish is

```
f90 -lnag test.f90
```

which means use the nag library to resolve any unresolved symbols, then compile and link `test.f90`. Given that the compiler has not been asked to do anything, there will not be any unresolved symbols when it meets the library. What is wanted is

```
f90 test.f90 -lnag
```

Secondly, the library option `-lfoo` is merely a convenient shorthand for 'search for `libfoo.so`, then `libfoo.a`, in a set of standard locations.' If you have installed a library somewhere non-standard, there is nothing wrong with specifying its precise location explicitly

```
f90 test.f90 /rscratch/spqr1/lib/libfoo.a
```

has always seemed to me simpler than the alternative of

```
f90 -L/rscratch/spqr1/lib test.f90 -lfoo
```

not least because there can be no ambiguity about which `libfoo` one gets in the first version.

Some modern compilers are less fussy about the order of their arguments. This can cause surprises, and, in some cases, is caused by the front-end reordering the arguments then calling a fussy backend. This can go very badly wrong if the re-ordering is not the one you wanted!

Old-fashioned people would expect commands such as

```
cc -O3 test.c -O2 test2.c
```

to build an executable from those two source files, compiling the first at optimisation level 3, the second at optimisation level 2.

Installing Software

It is likely that at some point you will wish to use some software which is not installed in TCM, despite being distributed at no cost. If several people wish to use it, then it may be best that it is installed centrally and supported. If there is an official Ubuntu package, it may be simplest to hassle the IT support people to install it anyway.

But, for a quick experiment at a weekend, why not install it yourself? Almost no Linux software requires root privilege to install. Certainly Castep, Matlab, Mathematica, gnuplot, jmol, LibreOffice, Firefox, etc. do not. Many are distributed as binaries, and simply need to be untarred. Even if compilation is necessary, the recipe is generally simply

```
tar -xf foo.tgz
cd foo
./configure --prefix=/rscratch/spqr1/opt
make
make install
```

or, for Python modules

```
tar -xf foo.tgz
cd foo
python ./setup.py install --prefix=${HOME}/.local
```

or simply

```
pip install --user foo
```

(One should read any files entitled 'README' or 'INSTALL' after the initial untarring.)

The UIS runs a good course entitled 'Unix: Building, Installing and Running Software' whose synopsis starts 'It is common for a student or researcher to find a piece of software or to have one thrust upon them by a supervisor which they must then build, install and use. It is a myth that any of this requires system privilege.'

Many rpms (the RedHat / SuSE package format) are fully relocateable, and contain no interesting installation scripts. For these

```
rpm2cpio foo.rpm | cpio -id
```

suffices to extract / install them. Similar tricks apply to `.debs`, the Debian/Ubuntu package format.

Index

30 UNIX Commands

`a2ps` *file*	Print text file, two pages per sheet
`bc`	Fixed point calculator. See page 29.
`cal` *month year*	Show calendar (use four digit year!)
`cd`	Change current directory to home directory
`cd` *dir*	Change current directory to *dir*
`chmod go=` *file*	Prevent others from reading *file*
`cp` *file1 file2*	Copy *file1* to *file2*, overwriting *file2* if it exists
`cp` *files... dir*	Copy multiple files to a directory
`date`	Show date and time
`du -sk` *dirs...*	Show disk usage of *dirs*
`echo` *text*	Repeats its arguments
`env`	Display environment variable settings
`file` *file*	Guess file type
`kill` *pid*	Ask process to exit
`kill -KILL` *pid*	Cause process to be killed
`less` *file*	View a file, page at a time. See page 8.
`lpr` *file*	Print *file* (text or Postscript) to default printer
`lpr -P`*printer file*	Ditto, to named *printer*
`lpq`	List printer queue (can add `-P`)
`ls`	List contents of directory
`ls -ltr`	Ditto, with sizes etc., and sorted by modification time
`man` *command*	On-line manual for *command*
`man -k` *keyword*	Search on-line manual for *keyword*
`mkdir` *dir*	Make directory
`mv` *file1 file2*	Move (rename) *file1*, deleting *file2* if it exists
`mv` *files... dir*	Move multiple files to a directory
`nice -15 cmd args`	Run cmd at reduced priority
`passwd`	Change password
`ps / ps aux`	List processes / list all processes
`quota`	Show filespace quota
`rm` *files...*	Delete files
`rmdir` *dir*	Remove directory (if empty)
`status`	Show machine's status (TCM only)
`time` *command*	time a simple command
`top`	View process activity. Press q to quit
`wc` *file*	Count lines and words in text file
`xterm`	A command shell in a window

Useful keypresses:

{ctrl}A – begining of line	{ctrl}C – terminate command
{ctrl}D – terminate input	{ctrl}E – end of line
{ctrl}H – delete (if all else fails)	{ctrl}Q – resume after {ctrl}S
{ctrl}S – pause display	{ctrl}Z – suspend process