UNIX:
an Operating Environment

MJ Rutter
mjr19@cam.ac.uk

Lent 2004

Bibliography

Computer Architecture, A Qualitative Approach, 3rd Ed., Hennessy, JL and Patterson, DA, pub. Morgan Kaufmann, £37.
Operating Systems, Internals & Design Principles, 3rd Ed., Stallings, W, pub. Prentice Hall, £30.

Both are thick (1000 pages and 800 pages respectively), detailed, and quite technical. Both are pleasantly up-to-date.

The `man` command.

Introduction

This is not a book: it is a collection of overheads for a short lecture course. That much should be clear. It is also not really intended to teach, but rather to stimulate people into going off and learning on their own, or reading other books, or attending other courses.

References to UNIX include all sane versions of UNIX unless otherwise stated.

Most references to DOS include Windows 3.x and the Windows 95 series, those to WinNT include Win2000 and WinXP.

References to MacOS usually mean versions pre MacOS X. MacOS X is a sort of UNIX.

Various things may be registered as trademarks: AIX, DOS, Irix, MacOS, Motif, OS/2, Tru/64, UNIX, VMS, Windows and others.

Contents

History

History: to 1979

1951	Ferranti Mk I: first commercial computer
1953	EDSAC I ‘heavily used’ for science (Cambridge)
1954	Fortran I (IBM)
1961	Fortran IV
1963	CTSS: Timesharing (multitasking) OS
	Virtual memory & paging (Ferranti Atlas)
1964	First BASIC
1969	ARPAnet: wide area network
1971	UNIX appears within AT&T
	Pascal
	First email
1972	Fortran 66 standard published
1975	UNIX appears outside AT&T
1978	K&R C appears (AT&T)

History: the Thatcher years

1980	Fortran 77 standard published
1981	MS DOS version 1
1983	Internet defined to be TCP/IP only
1985	LaTeX2.09
	PostScript (Adobe)
	Ethernet formally standardised
	X10R1 (forerunner of X11) (MIT)
	C++
1987	X11R1 (MIT)
1989	ANSI C
1990	PostScript Level 2
1991	World Wide Web / HTTP
1992	OpenGL
1993	Windows NT

UNIX History

1971	UNIX appears within AT&T
1975	UNIX appears outside AT&T
1978	Berkeley starts work on UNIX (BSD)
	3BSD released, adding virtual memory
1983	System V released (AT&T)
1984	4.2BSD with csh and vi
	SunOS
1993	4.4BSD. Berkeley ceases development
1994	Linux 1.0
1995	Linux 1.2 (first version in TCM!)
1996	Linux 2.0

The major current commercial UNIXes (AIX, OSF/1, Irix) appeared around 1990.

UNIX today

Currently (2004) the following versions of UNIX are being actively developed and are in widespread use:

- AIX 5L (IBM)
- Free and Net BSD (public domain)
- HP-UX (HP)
- Irix 6.5 (SGI)
- Linux 2.6 (public domain)
- MacOS X (Apple)
- Solaris 9 (formerly SunOS) (Sun)

Other, now mostly dead, varients include UNICOS (Cray), Ultrix (DEC), and the ill-fated OSF/1 / Digital UNIX / Tru64 from DEC / Digital / Compaq / HP, killed by a surfeit of names and takeovers.

Schisms

The BSD (Berkeley) and System V (AT&T) branches of UNIX were developed in parallel for a long while, and evolved slightly differently. Most modern UNIXes have features of both, but are closer to one than the other.

BSD

Printing command is `lpr`, options to `ps` have no dash (e.g. `ps aux`), only root can use `chown`, `id` lists supplimentary groups, init script are per run-level, '`shutdown -h now`' halts system.

System V (et al.)

Printing command is `lp`, options to `ps` have a dash (e.g. `ps -elf`), users can 'give away' files, `id` lists just primary group by default, init scripts are per 'service', '`shutdown -y -g0 -i0`' halts system.

May the best man win

The sillier points above have mostly died. The `lp` printing system is mercifully rare, and the idea that users can give other users ownership of their files, and thus make a nonsense quota controls, is still rarer. Similarly the BSD-style init scripts have mostly disappeared.

However, differing syntaxes for `ps`, `id` and `shutdown` still persist, leaving most UNIXes floating between the extremes of BSD and System V.

Nowadays many different bodies have attempted to standardise various aspects of UNIX (and OSes in general). Hence one frequently sees references to standards such as POSIX and X/OPEN, as well as BSD and SVr4 (System V release 4).

Solaris and Irix are mostly System V, whereas SunOS and OSF/1 are mostly BSD. Linux is confused, particularly as different distributions can do different things (Slackware used to have BSD-style init scripts, but no other distribution did).

The genuine article

UNIX is a trademark of AT&T, and in order to use it one must demonstrate that one's OS passes a compatibility test, and pay AT&T some money.

Many things which are commonly called UNIX are not actual UNIX in this legalistic sense. However, the rest of this work ignores such legalistic niceties: if it looks mostly like UNIX, then it's UNIX.

Linux is probably one of the most popular 'UNIXes', but it is not actually UNIX in this sense, whereas SCO's UNIX is not very widespread, but is really UNIX.

Operating System Concepts

What isn’t an Operating System?

An operating system is not the thing which interacts with humans. Rather it is a layer between programs and the hardware of computer.

UNIX is UNIX whether it is hiding underneath a graphical interface or a text interface, and whether that graphical (or text) interface appears to have been written in the 1980s, 1990s or the 21st century.

What is an Operating System?

Resource allocation (CPU time, memory)

Fair allocation between competing processes is good.

File system

Disks store raw data. File names and directories are an invention of the OS.

Hardware abstraction

A program wants to see a generic graphics device or keyboard, without needing to know the precise details of the model attached.

Security

Program A should be kept from program B's memory, and user A from user B's files.

A Process

A process is a single copy of a program which is running or, in some sense, active.

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), and it has an *environment*. 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.

There are one or two simplifications above, some of which will be untangled later.

Environment

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=sprq1`

All these things a process will inherit from its parent at the instant of its birth. The inheritance is a copy of the original: after that instance either the *parent* or *child* process can change these attributes without affecting the other.

MS DOS (and hence Windows) is not significantly different. It, too, has an arbitrary set of inherited environment variables, and the three streams above.

Parents

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 *reparented* by a special process called `init`, whose PID is one, and which is responsible for many house-keeping tasks.

Death

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.

Children also cost money. The parent process is charged for the CPU time used by the child.

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 reparent 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 reparent and remove the zombies.

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 sucsess. 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.

Process Trees

As each process has a sole parent, and may have no, one, or multiple children, on can draw a ‘tree’ of processes showing their relationships.

```
> ps -e --forest
PID TTY          TIME CMD
725 ?        00:00:00 xdm
736 ?        00:00:02  \_ X
737 ?        00:00:00  \_ xdm
768 ?        00:00:00      \_ fvwm2
819 ?        00:00:00          \_ FvwmButtons
821 ?        00:00:00          \_ FvwmIconMan
822 ?        00:00:00          \_ FvwmPager
823 ?        00:00:00          \_ xclock
824 ?        00:00:00          \_ xload
825 ?        00:00:00          \_ xterm
827 pts/0    00:00:00              \_ bash
836 pts/0    00:00:01                  \_ emacs
854 pts/0    00:00:00                  \_ ps
```

The `forest` option is found only on Gnu `ps` commands.

Address spaces

Each process has its own independent virtual address space for accessing memory. The dynamic mechanism for mapping this to real, physical memory is an exciting lecture in itself.

The result is that one process has no mechanism for accessing another process's memory, for a unique virtual address includes both the address *and* the PID, and identical virtual addresses with different PIDs will map to completely different physical addresses.

Thus with 32 bit Linux, the standard memory map leaves the code of a program starting at address `0x08000000` and the stack at `0xC000000`, and this is what each and every program sees each and every time it is executed, regardless of what else is going on in physical memory.

DOS uses only physical addressing. MacOS Classic and 16 bit versions of Windows use a single virtual address space for all processes. UNIX and WindowsNT use a separate VAS for each process as described here.

SIGSEGV

The mapping from virtual to physical addresses exists only for *pages* allocated by the OS. If a process tries to access a random virtual address, the mapping to a physical address will simply not exist, so the instruction must fail.

This failure is called a ‘segmentation fault’ (or ‘violation’), and can also be caused by trying to write to a read-only variable.

(The mapping from virtual to physical addresses is done with a granularity of one page, typically 4K or 8K. The amount of memory ‘owned’ by a process is thus always a multiple of the page size.)

Signals

A simple way for processes to communicate is via signals. A signal can be sent to any process running with the same user id, or any process if sent by the root user. The receiving process knows only that it received a signal: it does not know who sent it. It may choose to ignore certain signals. Thus this means of communication is not very reliable. However, its use is common.

Default actions for signals are ignore, terminate, terminate and dump core. However, for most signals an arbitrary *signal handler* may be supplied by the program. Two signals may not be *caught* by a signal handler, but alway have their default action: KILL and STOP.

Signals may be sent manually using the `kill` command.

Signal Numbers

Name	Number	Circumstance
HUP	1	Terminal (stdin/out/err) has disappeared
INT	2	Control-C pressed
QUIT	3	Abort and dump core
ILL	4	Attempt to execute illegal/invalid instruction
FPE	8	1.0/0, sqrt(-1), etc.
KILL	9	You must abort
SEGV	11	Invalid memory access attempted
PIPE	13	The other end of a pipe has died
TERM	15	Default sent by `kill` command
CHLD	17	A child has exited
CONT	18	Continue after STP / TSTP
STOP	19	Suspend process
TSTP	20	Control-Z pressed
XCPU	24	CPU time limit exceeded

The numbers above refer to Linux. Utter nerds prefer the numbers to the mneumonics, and can presuably cope with the different standards for the numbering. Solaris and Irix believe that CHLD is 18, TSTP is 24, CONT 25 and XCPU is 30, and Tru/64 that CHLD is 20, TSTP 18 and CONT 19.

Common sense

If a program doesn't care if it is forced to abort suddenly, it will do nothing to change the default action of TERM, and that will cause it to exit. If it does care, because it wishes to delete some lock file before exiting (netscape), or perform an emergency save of current work (many editors), it will catch TERM, and do such tidying up before exiting.

If one simply tries '`kill -KILL`' the process is unable to do any last-minute cleaning up: it be dead before it knows what has hit it. So in general one should use '`kill -KILL`' as a last resort (and never so frequently that it is worth remembering that on any POSIX-conformant system one can save three keypresses by typing '`kill -9`').

When a system is shutdown, it will send all processes the TERM signal, followed by KILL to remaining processes 5-10 seconds later. A queueing system may first send XCPU, then several tens of seconds later TERM then KILL.

Pardon?

Asking ‘root’ to kill your own processes for you is silly. A process does not know from whom it received a signal, and if you cannot attract its attention, root will fair no better.

And processes can get stuck in states where they ignore all signals. A common example is a process waiting for certain forms of I/O (usually I/O involving a device which has just suffered a hardware failure). A signal sent to such a process will (usually) be queued until the process can be interrupted.

A process which is suspended will ignore many signals (though not CONT!).

A zombie is not really a process, and will not respond to any signal.

The `ps` command shows the usual I/O wait state as a status of ‘D’.

Kernels

The OS kernel is very different from a process. There can only be one of it, and it can address physical memory directly, address any hardware directly, and do anything to any process. Indeed, one part of the kernel, the *scheduler*, is responsible for giving the processes any CPU time at all.

A process wishing to access some hardware device must do so via the kernel, and cannot do so directly. The kernel is able to ensure that when the CPU is not executing kernel code it is unable to execute certain privileged instructions which might allow a process direct access to the hardware.

This clearly requires some support from the CPU. CPUs of the early 1980s (8086 in the IBM PC, Z80 in the Sinclair Spectrum, 6502 in the BBC B, and others) simply did not have sufficient functionality. The i386 was the first PC processor really capable of running a modern OS.

Disk Access

A process will usually access a disk drive in terms of files. The kernel will oblige, imposing any restrictions indicated by the filesystem as it does so.

The kernel also presents disk drives as *device files*. These can be used by a process to read and write raw data blocks directly from and to the disk without going via the filesystem. Any process which can do this can therefore bypass any access restrictions imposed by the filesystem.

This is still not the real, physical hardware. The process is still shielded from having to worry about whether it should be sending IDE, SCSI or floppy commands to the disk, about which PCI bus the controller is on, and which ID it has on that bus, etc. It is also prevented from sending commands other than reads and writes: not the harmless ‘identify yourself’ command, nor the harmful ‘update your firmware from me’ command.

Root Processes

A process run by root is little different from any other process. It still needs to call the kernel to access any hardware, and the access will still be indirect. The difference is that the kernel is more likely to say ‘yes.’ A root process can trivially read from, or write to, any regular file or device file, send a signal to any process, change any processes scheduling priority up or down, etc.

However, it still operates in its own virtual address space, and it will still die with a segmentation fault if it tries to access memory not allocated to it. It will also die if it tries to execute a CPU instruction reserved for kernel mode.

Accidents and Design

If a non-root process hits a bug and starts behaving randomly, it is extremely unlikely to have any adverse affect on anything, beyond perhaps wasting CPU time in an infinite loop, or filling a disk with an infinite file.

A root process is much more likely to cause trouble if it is buggy, but the expected outcome is still an uneventful death.

Triggering a bug in the kernel is very much more likely to cause trouble. A crash of the whole operating system is the expected outcome, and data loss is not unlikely.

Keeping the kernel small is therefore a good idea.

Malign Design

If a user process has malign intent and intelligence, it can probably crash the system, or at least make it unusable. Merely creating several dozen copies of itself, and then having each add zeros until they reach infinity should do the trick.

A malign root process can trivially do enormous damage: read and modify any files, intercept any data passing through the machine, install a new or modified OS, reboot the machine, etc.

Other Privilege Models

The UNIX privilege model is somewhere between simple and simplistic. There are pretty much three levels: unprivileged user, root, kernel, and the last two effectively have full control.

The world of VMS (and Windows NT) is different. It contains a long list of extra privileges a process might wish to have, such as read all files on local disks (a backup process), send ‘interesting’ network packets (see later), change user id, listen on privileged network ports, send signals to any process, etc.

This model is begining to creep into UNIX, particularly IRIX and Linux, in the form of ‘capabilities.’

It may seem more sophisticated, and therefore superior, but it does have significant pitfalls. The capabilities overlap considerably, so it is much harder to work out how much privilege one really has given a user or a process. E.g. the privilege of writing to any file allows one to change any part of the OS, and thus gives one full control. So would giving full access to the raw disk device, *or* to the disk controller, *or* to the bus the controller is on. Writing to any file not owned by the system would probably be sufficient if one is cunning, sending signals *and* listening on privileged ports would surely be enough, etc.

The traditional UNIX model is so simple that the Board can almost understand it, so mistakes are less likely.

Libraries

Although user programs can call the kernel directly, usually they don't. Kernel functions are usually referenced by unmemorable numbers, and have a calling sequence which requires one to write in assembler.

In UNIX, the C library (`libc`) provides wrappers for all useful kernel functions so that they can be called directly from C with sensible names, and it provides all other non-maths functions required by ANSI C. (Maths functions required by ANSI C (trig, logs, etc.) are traditionally in a separate library, `libm`.)

So `libc` contains functions which are little more than wrappers for kernel functions (`write()`), functions which do a lot of work, and then probably call a kernel function (`printf()`), and functions which will not call the kernel at all (`strlen()`). Most other libraries will call functions from `libc` rather than calling the kernel directly.

Hence Fortran compilers usually link against `libc` as well as their own Fortran-specific libraries.

ASCII

Computers are not very good with text: they prefer numbers. Text is stored by converting each character into a number, a nasty little number in the range 0-255 which thus takes just 8 bits, and is not processed very efficiently with a 32 bit computer. One mapping of characters to numbers is almost universal, and that is ASCII. It is also almost memorable:

Binary	Decimal	Characters
000 0000	0	control codes start
001 1111	31	control codes end
010 0000	32	{space}
011 0000	48	0
011 1001	57	9
100 0001	65	A
101 1010	90	Z
110 0001	97	a
111 1010	122	z

Punctuation fills the gaps.
Note:
The eighth bit is unused (all numbers are < 128).
One bit distinguished uppercase from lowercase.

Control

In the bad old days, the same communication channel would be used both for talking to humans and for controlling peripherals. Hence thirty-two non-printable control-codes above. These include:

Binary	Decimal	Representation	Purpose
0 0100	4	^D	end of data
0 0111	7	^G	sound bell
0 1000	8	^H	backspace (delete)
0 1001	9	^I	tab
0 1010	10	^J	line feed (enter)
0 1100	12	^L	form feed (new page)
0 1101	13	^M	carriage return
1 0001	17	^Q	XON (transmit on)
1 0011	19	^S	XOFF (transmitt off)
1 1011	27	^[	escape

The control codes from 1 to 26 can be produced by holding down the control key and typing the corresponding letter of the alphabet.

Forgotten, but not gone

Many interactive applications will exit if control-D is pressed, control-H works as a delete key at least as reliably as the key above enter, control-S will, in some circumstances, successfully request that the other end cease transmitting until it receives a control-Q, control-L is widely recognised as meaning ‘redraw screen,’ and control-G will cause a beep to be sounded.

Agreement over how lines should be ended has not yet been reached. DOS believes in carriage return and line feed, UNIX in line feed only, MacOS in carriage return only.

Old printers agreed with DOS: carriage return only caused overprinting – useful for generating bold or struck-out text, and line feed only produced ‘staircases’. However, a single character for end of line is much more convenient.

Mastering control

Entering a control character into text can be taxing. The usual convention (shells and vi) is to precede it with control-V. Emacs disagrees, and needs control-Q.

Viewing files containing them is also tedious. Simply using `cat` is a disaster, as each `^G` will cause the computer to beep, and most sequences starting with `^[` will cause one's terminal to do odd things. The program `less` does a very good job of handling these files without causing pain for one's terminal or one's ears. The convention for representing them is either reverse-video and the caret notation used here, or their hex code in angled brackets. The latter has to be used for non-printable characters with codes above 32.

For completeness, zero is represented by `^@` (as @ has ASCII code 64), and 27 to 31 by `^` and those characters whose ASCII codes are 91 to 95, namely `[\]^_`. So pressing the control key means 'subtract 64 from the ASCII code of the following symbol', just as pressing the shift key means 'add 32' in the case of letters.

Emacs is vile. It does not follow the conventions for `^C`, `^H`, `^S`, `^Q` and others besides, and thus needs a very forgiving terminal.

The Great Unknown

In the early days, the ‘unused’ (for text) ‘top’ bit of each byte was used as a parity bit to provide simple error detection: set to one if an odd number of bits in the rest of the byte were one. Although it has long since stopped being used thus, there is no universal standard on how it should be used. The ISO 8859/1 ‘Latin 1’ standard is the most widely used in Western Europe and the US. It adds important characters such as £, and enough squiggles over letters to amuse the French, Germans and Spanish. However, it contains no Greek at all.

ISO 8859/1 is not what DOS used, nor what HP printers default to, and whereas ISO 8859/1 believes that character number 163 is ‘£’, DOS tends to think it is ‘ú’, and ISO 8859/2 thinks it is ‘Ł’. All agree that character number 65 is ‘A’.

Thus if using ‘high ASCII’, it is important to specify which encoding one is following.

Clean or unclean

ASCII is divided into ‘printable’ (32 to 126) and ‘non-printable’ characters (character 127 is delete, character 8, ^H, is backspace).

The phrase ‘7 bit clean’ (ASCII) refers to a document which contains only printable characters, plus tab, line feed and carriage return. These documents are least likely to cause any surprises, and should be used where reasonable. Certainly program source files (including PostScript) and emails should fall into this category.

Data which describes itself as ‘8 bit clean’ adds all characters in the range 128 to 255, and 8 bit binary data contains simply any possible byte value.

A simple recipe for converting 8 bit data to 7 bit clean data would be to take each triplet of bytes (24 bits), split into four groups of six bits, each of which can be considered to be a number in the range 0-63, and then add 32 to each. This expands three bytes to four, but keeps each in the range 32 to 95. Thus the original `uuencode` program. Base64 encoding (as used by email for binary attachments) is similar, but avoids using space and much punctuation in its output. PostScript uses a marginally more efficient encoding, ASCII85, which expands four bytes of binary data to five, and also avoids relying on spaces being preserved.

Some people believe that codes 27 to 31 are acceptable in ‘clean’ ASCII.

Unicode

The future, if we are unlucky, is Unicode. This permits one to encode all conceivable characters in a scheme which has a single character occupying multiple bytes: at least two, and maybe more.

The good news is that it can cope with the Latin, Greek and Hebrew alphabets, as well as Near and Far Eastern scripts. And Microsoft has firmly embraced it in Windows.

The bad news is that for simple Latin text, it doubles the file size, and makes interpretation *much* harder, as characters can potentially be of variable length. However, in its simplest form it looks like ASCII with nuls (^@) added every alternate byte.

7 Bit ASCII

00	0x00	^@	32	0x20		64	0x40	@	96	0x60	‘
01	0x01	^A	33	0x21	!	65	0x41	A	97	0x61	a
02	0x02	^B	34	0x22	"	66	0x42	B	98	0x62	b
03	0x03	^C	35	0x23	#	67	0x43	C	99	0x63	c
04	0x04	^D	36	0x24	$	68	0x44	D	100	0x64	d
05	0x05	^E	37	0x25	%	69	0x45	E	101	0x65	e
06	0x06	^F	38	0x26	&	70	0x46	F	102	0x66	f
07	0x07	^G	39	0x27	’	71	0x47	G	103	0x67	g
08	0x08	^H	40	0x28	(	72	0x48	H	104	0x68	h
09	0x09	^I	41	0x29	)	73	0x49	I	105	0x69	i
10	0x0A	^J	42	0x2A	*	74	0x4A	J	106	0x6A	j
11	0x0B	^K	43	0x2B	+	75	0x4B	K	107	0x6B	k
12	0x0C	^L	44	0x2C	,	76	0x4C	L	108	0x6C	l
13	0x0D	^M	45	0x2D	-	77	0x4D	M	109	0x6D	m
14	0x0E	^N	46	0x2E	.	78	0x4E	N	110	0x6E	n
15	0x0F	^O	47	0x2F	/	79	0x4F	O	111	0x6F	o
16	0x10	^P	48	0x30	0	80	0x50	P	112	0x70	p
17	0x11	^Q	49	0x31	1	81	0x51	Q	113	0x71	q
18	0x12	^R	50	0x32	2	82	0x52	R	114	0x72	r
19	0x13	^S	51	0x33	3	83	0x53	S	115	0x73	s
20	0x14	^T	52	0x34	4	84	0x54	T	116	0x74	t
21	0x15	^U	53	0x35	5	85	0x55	U	117	0x75	u
22	0x16	^V	54	0x36	6	86	0x56	V	118	0x76	v
23	0x17	^W	55	0x37	7	87	0x57	W	119	0x77	w
24	0x18	^X	56	0x38	8	88	0x58	X	120	0x78	x
25	0x19	^Y	57	0x39	9	89	0x59	Y	121	0x79	y
26	0x1A	^Z	58	0x3A	:	90	0x5A	Z	122	0x7A	z
27	0x1B	^[	59	0x3B	;	91	0x5B	[	123	0x7B	{
28	0x1C	^\	60	0x3C	<	92	0x5C	\	124	0x7C	|
29	0x1D	^]	61	0x3D	=	93	0x5D	]	125	0x7D	}
30	0x1E	^^	62	0x3E	>	94	0x5E	^	126	0x7E	~
31	0x1F	^_	63	0x3F	?	95	0x5F	_	127	0x7F	

The full 7-bit ASCII set, with decimal and hex values for each character. Note that 32 is space, and 127 delete.

Shells

Shells

Most operating systems are provided with some form of command line interface or shell. This is not strictly part of the operating system, but rather a useful utility program.

DOS had `command.com`, Windows 2000 has `cmd.exe`, UNIX has the Bourne shell, and MacOS Classic demonstrates that one can have an operating system without a command line interface.

Sometimes the graphical interface of an OS is also referred to as a shell, particularly OS/2's 'workplace shell'.

The shell interacts with humans, and launches other programs for them. It itself is just a program too, so one can replace one's shell without changing one's operating system in any sense. The free shell `4DOS` existed for DOS, UNIX has a choice of half a dozen shells.

The shell is often called a command interpreter: it obeys commands, and interprets (rather than compiles) them.

UNIX shells

The first UNIX shell was Steve Bourne's, developed in 1974 at Bell Labs. It is not well suited to interactive use, although significant enhancements were made to it for the next decade.

The next, the C shell (`csh`), came from Berkeley in around 1978. It was rather better suited to interactive use, but its syntax was significantly incompatible with the Bourne shell.

Since then several improvements of these two basic shells have appeared.

Improved Shells

- Korn Shell (`ksh`) developed by David Korn in 1983. A superset of the Bourne shell with better interactive features.
- `ash` Another superset of the Bourne shell, loved by the BSD project.
- Z Shell (`zsh`) circa 1990. Yet another superset of the Bourne shell designed for interactive use.
- `bash` (Bo(u)rn(e) Again SHell, 1989). A great improvement on the above, whilst retaining compatibility with the Bourne shell. Loved by GNU and Linux.
- Tenex Shell (`tcsh`) – a C shell superset with good interactive functionality.

The `bash` and `tcsh` shells both allow command recall and editing with the cursor keys, whereas `sh` and `csh` do not.

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 one's 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.

Here the Bourne-like and C-like shells differ in syntax. To set a shell variable:

```
sh:  x=5
csh:  set x=5
```

To set an environment variable:

```
sh:  x=5 export x
csh:  setenv x 5
```
(N.B. no '=' sign)

In all cases, to see the result:

```
echo $x
```

Prompt Conventions

One shell variable sets the default prompt sting. 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.

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

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

which gives prompts such as

```
tcm30:~/talks>
```

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

This is all quite similar to DOS’s usual

```
set prompt=$p$g
```

However, in DOS all variables are environment variables, that is, they are inherited by child processes.

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. C-like shells build a complete table of all the commands which exist on the path when they are started, and never search the directories themselves.

C-like shells do rebuild their table should `$PATH` be modified.

If a command is added to a directory in the path, C-like shells *will not notice.* One must type '`rehash`' to cause them to rebuild their tables. 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’
> f77 -o test test.f
> test
> ./test
Hello
```

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

The dot heresies

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 his current directory, he clearly wants 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 they can booby-trap commands there:

```
> cd /tmp
> ls
Gotcha!
> /bin/ls -l
-rwxr-xr-x 1 spqr1 spqr1
           23 Dec 3 20:17 ls
> less /tmp/ls
#!/bin/sh
echo Gotcha!
```

And it could be much worse than that…

The last heresy

Others believe that ‘.’ is safe if last on the $PATH. If last the above trick will not work. However, seeding `/tmp` with common misspellings (`mroe`, `ks`, `xs`) will.

So ideally dot is not placed on the $PATH, and people learn to type a leading ’./’ if they wish to execute something from the current directory.

However, the default $PATH should include a user-specific directory, such as `$HOME/bin`, before all system directories so that the user can over-ride system commands if he is mad.

In TCM ‘dot’ is on the $PATH, for historical reasons. At least it is last. $HOME/bin is also on the $PATH.

Filename completion

The `csh`, `tcsh` and `bash` shells all offer *filename completion*. If, whilst typing a filename, one presses {TAB}, any extra characters which can be uniquely determined are filled in automatically. If there are multiple alternatives, pressing {TAB} twice in succession will list them.

Completion also works for command names:

```
$ ep{TAB}{TAB}
eps2gif      eps2ps       epstopdf
eps2eps      epsffit      epszip
```

For `tcsh`, alternatives are listed if {crtl}D is pressed and the cursor is at the end of the line.

For `csh`, only basic completion is available, the key is {ESC} not {TAB}, and the shell variable `filec` must be set.

Built-in commands

Most common UNIX commands are *external* to the shell: they are simple stand-alone programs, usually found in `/usr/bin`, and one runs precisely the same program independent of one's shell.

A few commands are, necessarily, *internal*: that is, they are part of the shell. These fall the follwing categories:

- flow control commands: loops, `if`, `case`, etc.
- commands altering the shell's internal state: `alias`, `pushd`, `popd`, `set`, `shift`, `history`, `(re)hash`.
- commands altering the shell's process status: `setenv/export`, `(u)limit`, `cd`, `umask`.
- job control: `fg`, `bg`, `exit`, `exec`
- there for efficiency: `echo`

An external `cd`

An external `cd` command exists on some systems as `/usr/bin/cd`, and it is a good example of near uselessness.

When run, it changes its current working directory and then exits. However, because the current working directory is held on a per-process basis, and a child cannot affect the state of its parent, the current working directory of the process which launched it is unchanged.

MS-DOS also has internal and external commands in the same fashion, although slightly more commands, such as `copy` and `dir`, are internal for efficiency. Many DOS commands, such as `format`, `xcopy` and `more` are external.

In and Out

The C programming language has the concept of three I/O channels: one for input, one for output, and one for error messages. These are called `stdin`, `stdout` and `stderr` respectively. By default a shell will run a program with all of these attached to the (pseudo) terminal the shell was using. However, they can be redirected.

The `>` character redirects `stdout`, and `<` `stdin`. Using `>>` causes `stdout` to be appended to a file, rather than over-writing it.

The C-like shells use `>&` and `>>&` to redirect `stdout` and `stderr` together, whereas the Bourne-like shells can redirect these separately:

```
$ ./a.out > output 2>errors
```

DOS has just > and <.

Experiments

```
$ 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
```

Note that when there is no output, the output file will get truncated to zero length.

Note too that the use of `cat` is not recommended. Using `less` is much safer if you accidentally hit a long, or binary, file.

Throwing things away

UNIX provides various special files, one of which can be very useful with redirection. The ‘file’ `/dev/null` simply discards anything given to it. Thus

```
> rm womble >& /dev/null
```

will delete and not give an error if the file does not exist (or if the file is undeletable).

If one reads from `/dev/null`, an immediate ‘end of file’ error is produced.

Pipes

One can also direct the output of one command into the input of another.

```
> ls -l
total 28
-rw-r--r-- 1 spqr1 spqr1  1024 Dec 18 19:59 magnum
-rw-r--r-- 1 spqr1 spqr1  2048 Dec 18 19:59 maior
-rw-r--r-- 1 spqr1 spqr1 10240 Dec 18 19:59 maximum
-rw-r--r-- 1 spqr1 spqr1     0 Dec 18 19:58 minimum
-rw-r--r-- 1 spqr1 spqr1     1 Dec 18 19:58 minor
-rw-r--r-- 1 spqr1 spqr1     2 Dec 18 19:58 parvum
> ls -l | sort -nr -k 5
-rw-r--r-- 1 spqr1 spqr1 10240 Dec 18 19:59 maximum
-rw-r--r-- 1 spqr1 spqr1  2048 Dec 18 19:59 maior
-rw-r--r-- 1 spqr1 spqr1  1024 Dec 18 19:59 magnum
-rw-r--r-- 1 spqr1 spqr1     2 Dec 18 19:58 parvum
-rw-r--r-- 1 spqr1 spqr1     1 Dec 18 19:58 minor
total 28
-rw-r--r-- 1 spqr1 spqr1     0 Dec 18 19:58 minimum
```

Valid options for the `sort` command vary enormously between different flavours of UNIX: use the `man` command to determine what is permitted.

Pipes in detail

DOS has pipes too, and, in DOS

```
C:\> dir | more
```

is precisely equivalent to

```
C:\> dir > tempfile
C:\> more < tempfile
C:\> del tempfile
```

This is not true in UNIX, where

```
> ls -l | less
```

causes the two processes `ls` and `less` to be launched ‘simultaneously’, and output to be transferred from one to the other, possibly (probably) after some buffering. This is much more useful: in the DOS world, one does not start to see any output until the first command has completed.

Launching programs

The actual launch of a program is done by the OS, not directly by the shell. The UNIX kernel understands just two calls related to process creation: `fork()` and `exec()`. The former creates a clone of the current process. The original keeps its own PID, and gets the PID of its child returned by the `fork()` call. The child is identical in every respect – it, too, appears to have run up to the same point as its parent, and to have just returned from the `fork()` call – except that it will have a new PID, and the value returned by `fork()` will be zero.

The latter, `exec()` replaces the current process with the process formed by executing the specified file. The PID is unchanged by `exec()`, as are the environment variables, but all the process's memory is obliterated by the new process.

A value of -1 is returned by `fork()` if it fails. In this case there is no child.

A process may `exec()` itself to restart, thus, for instance, re-reading its configuration files.

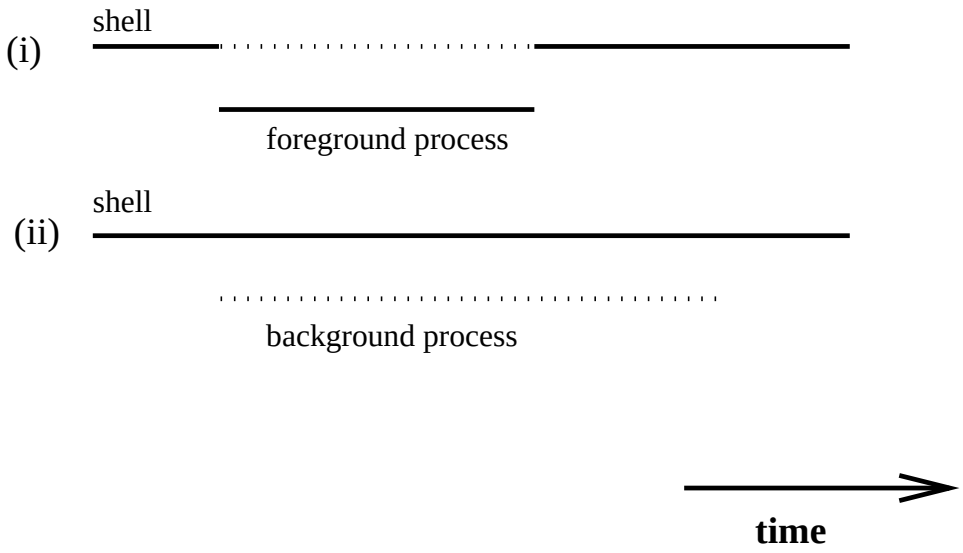
A team

The `fork()` and `exec()` calls are often used as a pair:

```
i=fork();
if (i>1){ /* I’m the parent */
}
else if (i==0){ /* I’m the child */
  exec("/some/program");
}
else{ /* Bother: fork() failed */
}
```

In the case of the shell, it, the parent, by default backgrounds itself and lets its child take control of the keyboard. However, if the command line ends with &, the child is left in the background, and the shell in the foreground.

Foreground and Background



The solid line represents the process which has control of the terminal, a *foreground process*, the dotted line a process running but not receiving input from the terminal, a *background process*.

Case `(i)` is the default, case `(ii)` is achieved simply by placing an ‘`&`’ at the end of the command line.

To make the last background process a foreground process, type ‘`fg`’, and to background a foreground process, suspend it by typing {ctrl}{Z}, then type ‘`bg`’.

Magic and exec

For `exec()` to work on a file, that file must have execute permission for the user calling `exec()`. What happens next depends on the contents of the file.

If the first two characters are ‘`#!`’ then it is a script. The word immediately after the ‘`#!`’ is the full path to the program to launch as the interpreter, any arguments after this are then passed to the interpeter, followed by the name of the script, and finally any command-line arguments.

The key signatures at the beginning of files which allow their type to be detected in this manner are called ‘magic numbers’.

More magic

If the first two characters are not ‘#!’, then the one hopes the file is a standard executable. Linux knows about several formats, the current one being ELF, which starts ‘<7f>ELF’ and goes on to specify what processor the file is intended for, etc, as the ELF file format is used by many UNIX variants.

Tru64 on Alphas uses a format called COFF, and its executables start ‘<01><83>’, and thus an Alpha running Tru64 can immediately spot if it is being asked to run a program compiled for Intel Linux, and vice versa, and both will refuse.

The command `file` will read the first few bytes of a file, and compare it with a known table of magic numbers, and report its deductions. The Gnu version on Linux has a particularly comprehensive table.

ELF: Enhanced Library Format
COFF: (un)Common Object Format File

Much more magic

Note that the concept of ‘magic numbers’ applies just as much to data files as to executables. The world of DOS and Windows may believe that the filename denotes the type, and MacOS may believe that such information is embedded in a fork, but that is not how the UNIX world works.

`%!`	Postscript or EPS file
`%PDF-`	PDF file
`BZ`	bzipped data
`GIF`	GIF graphics file
`MZ`	MS DOS (Windows or OS/2) executable
`0x1f8b`	gzipped data
`0xedabeedb`	RPM file (RedHat package)
`0xffd8`	JPEG graphics file

As usual, text given if the magic is printable, otherwise given in hex following ‘`0x`’. Thus ‘BZ’ could equivalently have been written ‘`0x425a`’.

Using ‘`less -U`’ to examine the beginning of a file is sane and safe.

UNIX compilers *do* use the name to determine type, and treat `.c`, `.o`, `.cc`, `.f` and `.F` files very differently.

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 it expects wildcard expansions, variable substitutions, and similar processing, to be done for it. This has one clear advantage: whereas in DOS some commands understand how to process wildcards 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`.

An example

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

We now have a simple shell script which will echo back its first two arguments.

Use your favourite editor to create the above script. If using `cat`, remember '`^D`' means type {ctrl}{D}. The `chmod` command makes this script as executable. As . is not necessarily on the default search path for commands, we shall use a preceding './' to execute this script.

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
```

Note the silent removal of excess spaces between words. What happens for `./args " hello world"`

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 ~ ~spqr1
The first  argument is: /home/mjr
The second argument is: /domus/spqr1
```

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

Hello, hello

```
> 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:
```

Note:
variables expanded first, then result split into words.
variables in double quotes are expanded.
variables in single quotes are not expanded.

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:
```

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, `csh` users must type '`source filename`' and Bourne shell users '`. filename`'.

```
tcmpc52:~> cat silly
#!/bin/sh
cd /
tcmpc52:~> ./silly
tcmpc52:~> source silly
tcmpc52:/>
```

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

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 `tcsh`, the sequence is:

`/etc/csh.cshrc`
`/etc/csh.login` (if a login shell)
`~/.tcshrc` or, if not found, `~/.cshrc`
`~/.login` (if a login shell)

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

The `csh` is as `tcsh` but without `~/.tcshrc`, and `sh` as `bash` but without files containing 'bash' in their name.

Startup logic

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.

Scripts read on login may produce output (‘Good morning, sir, you are 4KB below you disk quota again’). Scripts read by non-login shells may not.

Shell startup is depressingly common: many programs do it to expand wildcards, 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.

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.

Dot files are important

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.

Wildcards

Most people are familiar with the *wildcards* '*' (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
```

Order, order

It is clear to any sane computer that such sequences, and the natural order for textual things, is simply the numerical ordering of the ASCII codes, so digits before letters, all upper case before all lower case.

Meddling humans disagree, believing the correct collating sequence to run ‘AaBbCc…’, with ‘A’ and ‘a’ scoring identically, not ‘ABC…abc…’ Thus madness such as:

```
> ls
apple  Bill  pear
> LANG=C export LANG
> ls
Bill  apple  pear
```

It is important to ensure that the second form is the default, or there is a danger that ‘[A-Z]’ will match all lowercase letters too, as it will be interpretted as meaning any character which comes between ‘A’ and ‘Z’ in the collation sequence. Certain versions of RedHat Linux default to the ‘human-friendly’ collation sequence.

The French believe that ‘e’, ‘è’, ‘é’ and ‘ê’ rank equal for sorting. That is 101 = 231 = 232 = 233 as far as their codes in ISO 8859/1 are concerned, and that is before one considers capitalisation. The overhead of teaching computers such insanities can be significant.

Miscellaneous Commands

`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.

However, `find` is the cause of endless confusion.

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

First things first

The first argument to `find` is the directory to start searching from. It may not be omitted, so, if one wishes to start at the current directory, a dot must be given explicitly.

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

And *NEVER* try something like

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

because this will search through all remotely-mounted disks too. 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`, ...

Quote!

The most common use of `find` is dealing with half-remembered filenames:

```
find . -name *.eps -print
```

But this isn’t how it’s done, because the shell will expand that * and chaos will result. So

```
find . -name ’*.eps’ -print
```

is the answer.

Do what?

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.

```
find . -name '*.eps'
```

Gnu’s `find` will assume one meant `-print` at this point, but a traditional `find` will print nothing, whether or not anything is found. Some `find`s do not support `-ls`.

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

Living dangerously

One can cause `find` to launch commands on files found:

```
find . -name core -exec rm {}\;
```

Such automation is normally a guaranteed recipe for wholesale, unexpected destruction. It may be just about safe with `touch`, preferably in conjunction with `-xdev`.

The sequence '{}' is replaced by the full path of the file found, and the command to be launched by `-exec` must be terminated with `\;`. If you suffer from a `find` which does not support `-ls`, then `-exec ls -l {}\;` is the answer.

The `man` Command

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

Most UNIXes impliment the `man` command as described below, but, beware of AIX, which strongly prefers IBM's own 'info' system.

Man pages are stored in a simple typesetting language called 'troff'. The man command is responsible for finding the correct page, calling some troff variant in order to typeset it, and then displaying the result through a pager, usually `more`.

Manual Chapters

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

Man on Disk

There are usually several collections of man pages on a single computer. The commands in `/usr/bin` may be documented in `/usr/man`, whereas those in `/usr/local/bin` in `/usr/local/man`. Each collection has the following structure.

- whatis file containing index of all pages in this tree
- mann directory/ies containing troff source of pages
- catn directory/ies of preformated pages

There is no need for all possible chapters to be present, or for the preformatted pages to exist at all.

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 total line if more than one FILE is specified. With no FILE, or when FILE is –, read standard input.

–c, **– –bytes**

print the byte counts

–l, **– –lines**

print the newline counts

[etc.]

Another man page

wc(1) wc(1)

NAME

wc - Counts the lines, words, characters, and bytes in a file

SYNOPSIS

wc [**–c | –m**] [**–lm**] [file. . .]
The **wc** command counts the lines, words, and characters characters or bytes in a file, or in the standard input if you do not specify any files, and writes the results to standard output.
It also keeps a total count for all named files.

OPTIONS

–c Counts bytes in the input.
–l Counts lines in the input.
–m Counts charcters in the input.
–l Counts words in the input.

DESCRIPTION

[etc.]

EXAMPLES

[etc.]

The Similarities

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 these 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 similarities

The Synopsis section should give a brief summary of the syntax for the command. Things included in [ ] 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. It should also give a c. one paragraph summary of what the command does.

This should be followed by a (usually alphabetical) list of options, and a description of what they do.

Towards the end one should see examples, lists of standards to which the command conforms, and, finally, a list of related man pages.

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. Tru64 is usually significantly better, and Solaris better still.

Search Order

Sometimes man pages appear in multiple sections. Examples include `exit`, `printf`, `mkdir`, `cvs`, `crypt` and many others.

The `whatis` command will display all appropriate summaries, whereas the `man` command may display just one, using a precidence order which is non-obvious (1,8,2–7 for RedHat, other orders for Tru64 and Solaris). To specify a precise page, one must specify the section too, as in

```
man 3 printf
```

or

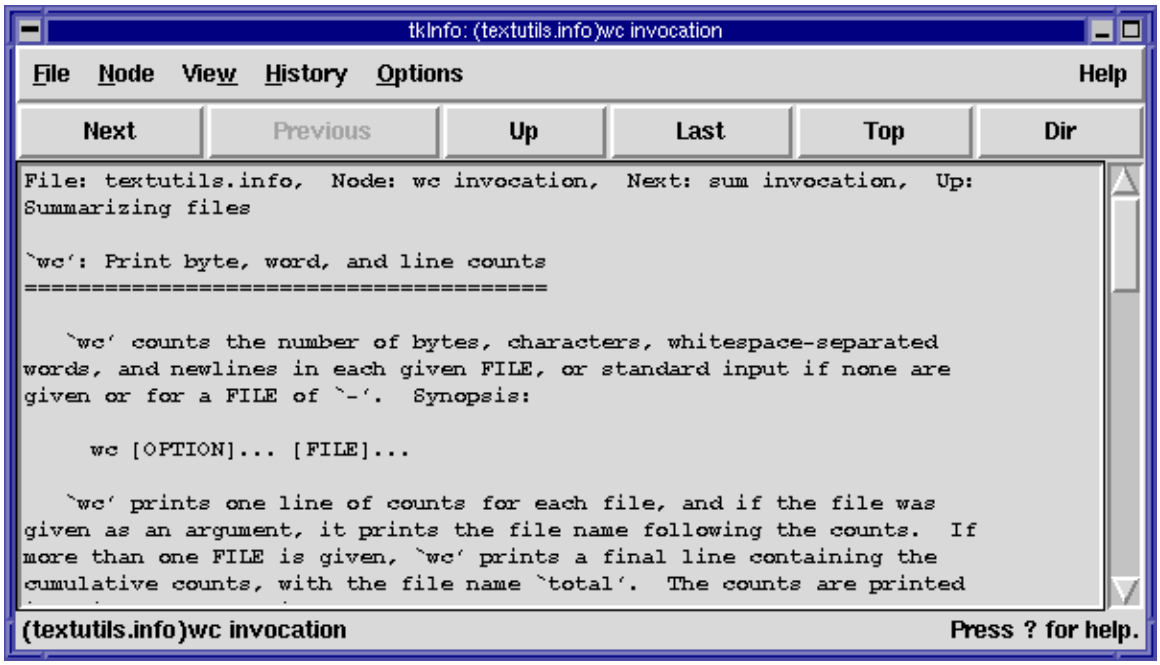
```
man -s 3 printf
```

the latter being Solaris's syntax.

If the whatis database is missing, or out-of-date, then `man -k` cannot work correctly, although `man` itself will find pages not in the whatis database.

Other sources of documentation

The Gnu project is hopelessly fond of its info system for documentation. Keen `emacs` users will be familiar with it, and will also like the text-based `info` command. For the rest of us, `tkinfo` provides a friendly graphical interface to this system.



The `more` command

The `more` command is used to display a text file one screenful, or page, at a time. It is often known as a *pager*, and some programs will use the environment variable `$PAGER` to indicate which pager is prefered.

Most will be familiar with the following keypresses:

{space}	next page
{enter}	next line
d	scroll about half a page
/text	next occurance of `text`
n	repeat previous search
v	start `vi`
q	quit
{ctrl}L	redraw screen

More enhanced

Many versions of `more` offer considerably greater functionality, including:

b	previous page
j	previous line
u	reverse scroll c. half a page
?text	previous occurrance of `text`
G	goto end of file
numG	goto line number `num` (1G for beginning)
{ctrl}G	display current position in file
:i	toggles case-sensitivity of searches

A very fully-featured variant of `more` is Gnu's `less` (a horrible pun).

`man` **and** `more`

The `man` command uses an external pager, often `more` or `less`. Hence the above keystrokes should work when viewing man pages.

RedHat Linux uses `less` with searches defaulting to case-insensitive. Solaris uses a version of `more` which does not support moving backwards at all, and thus one should consider setting the environment variable `$PAGER` to '`less -isr`' to encourage man not to use this pager. Tru64 uses `more`, but a rather better version than Solaris's, and it does support the 'b' and 'G' commands.

All for less

Of course, these pagers can be used with *any* command, by using a pipe to pipe the output of one command into another. E.g.

```
> ls -l | less
> du -sk * | less
> du -sk * | sort -nr | less
```

This works because although `more` and `less` will display a file if a filename is given, if no filename is given they simply display `stdin` page at a time.

Users of DOS will find that DOS's `more` command is similar, except that it does not accept filenames: it only reads from stdin. So, to display a file page at a time one types:

```
c:\> more < autoexec.bat
```

Cool and Calculating

It is always depressing to see someone sitting in front of a computer ask for a pocket calculator. Computers can do arithmetic too!

The program `xcalc` is a well-known GUI calculator which one can drive with a mouse. It has basic scientific operations, and one can cut and paste from (but not to) it. (‘`xcalc -rpn`’ gives a Reverse Polish version.)

The program `bc` is less well-known, but more common, and is a text-mode programmable calculator. It defaults to be an arbitrary-precision integer calculator, with just the `sqrt` function defined.

```
> bc
6*7
42
2^10
1024
7/6
1
```

More calculating

Envoking `bc` as `bc -l` sets the scale (the number of decimal places) to 20 and predefines the following functions: `a(x)` (atan), `c(x)` (cos), `e(x)` (exp), `j(n,x)` (`Bessel`) and `l(x)` (ln).

```
> bc -l
7/6
1.16666666666666666666
scale=30
4*a(1)
3.141592653589793238462643383276
define f(x){
  auto i,s;
  s=1; for(i=1;i<=x;i++) s=s*i
  return(s)
}
f(4)
24
f(24)
620448401733239439360000
```

Gnu's version of `bc` is more accommodating than those found in most commerical UNIXes, happily calculating 1000 digits of π. However, `bc` can be very slow compared to other high-precision packages.

Finding things

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
 TOTAL ENERGY IS           -789.4217177
 TOTAL ENERGY IS           -790.2230024
 TOTAL ENERGY IS           -790.3021778
 TOTAL ENERGY IS           -790.3107729
```

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

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

Quoting and counting

When using `grep`, it is important to remember which characters need quoting from the shell, and safest simply to enclose the string one is seaching for in 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
```

More complexity

Most people are familiar with the shell wildcards ‘*’ and ‘?’ used for filename ‘globbing’. However, the general syntax for wildcards 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/dict/words
independent
```

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

```
>  wc -l /usr/dict/words
   25143 /usr/dict/words
```

A beginning and an end

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

```
> grep 'pret$' /usr/dict/words
interpret
> grep '^pret' /usr/dict/words
pretend
pretense
pretension
pretentious
pretext
pretty
```

Such regular expressions are called *anchored*.

Repeats

The character ‘*’ means any number (including zero) of the preceeding character.

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

Thus ‘.*’ is the equivalent of ‘*’ as a shell wildcard.

Ranges

Square brackets denote ranges, just as for shell wildcards.

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

Yes, that really is 4,974 proper nouns and abbreviations.

Extensions

Some `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
```

More extensions

One can also specify multiple expressions to match for used extended regexps:

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

Negated Ranges

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

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

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

More ideas

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.

Polylingualism

Sometimes it is useful to be able to exchange text files with people using DOS's or MacOS's odd conventions. Usually one's editor will cope, but otherwise the '`tr`' program is good at single-character translations to change the different end-of-line codes.

DOS to UNIX (CRLF to LF)

Simply delete all carriage returns.

```
> tr -d '\r' <file.dos >file.txt
```

Mac to UNIX (CR to LF)

```
> tr '\r' '\n' <file.mac >file.txt
> tr '\n' '\r' <file.txt >file.mac
```

The `tr` command can also do multiple substitutions. The most well-known example is to impliment the 'ROT13' code, which moves each letter forward 13 places in the alphabet, and for which encoding and decoding are equivalent.

```
> echo 'Hello World' | tr '[A-Za-z]' '[N-ZA-Mn-za-m]'
Uryyb Jbeyq
> echo 'Uryyb Jbeyq' | tr '[A-Za-z]' '[N-ZA-Mn-za-m]'
Hello World
```

sed

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

UNIX to DOS (LF to CRLF)

```
> sed 's/$/^M/' <file.txt >file.dos
```

This finds end-of-line characters, and adds `^M` before them.

For this simple example, using `sed`'s substitute command (s), the regular expression to search for is between the first two '/' characters, and the replacement string between the second two. Only the first occurance on each line will be replaced.

Remember that you may have to type `^M` as {ctrl}V{ctrl}M

More sed

Code uglification

Remove indentation:

```
> sed 's/^ *//' < code.old > code.new
```

Remove C++ comments:

```
> sed 's%//.*%%' < code.old > code.new
```

Remove F90 comments:

```
> sed 's/!.*//' < code.old > code.new
```

Remove F77 comments:

```
> sed '/^[cC]/d' < code.old > code.new
```

(on any line matching the regexp '`^[cC]`' perform the operation 'd' (delete line))

Code conversion

Convert C++ comments to C comments:

```
>sed 's%//\(.*\)%/*\1 */%' < old > new
```

For those unfamiliar with the above languages:

```
> echo 'foo // bar' | sed 's%//\(.*\)%/*\1 */%'
foo /* bar */
```

WARNING: the above are merely examples: they do not deal correctly with comment characters in strings, etc.

`tar` and backups

The traditional UNIX backup command is `tar`: Tape ARchive. It takes one or more files or directory trees, and bundles them up as a single stream of data suitable for placing on a tape. It can also perform the reverse operation. There are several, marginally incompatible tar formats, and all impose some restrictions on filename length, file sizes, and similar tedious aspects.

The `tar` command does not compress the data. If compression is required, it must be requested explicitly (Gnu's tar), or obtained by using a pipe and another program.

To create an archive:

```
> tar -cf castep.tar castep
```

or

```
> tar -cf - castep | gzip > castep.tgz
```

or

```
> tar -czf castep.tgz castep
```

The '`c`' option specifies 'create'. The '`f`' option the file to create: otherwise the default is the tape drive on the local machine! As is the case for many commands, specifying an output file of '`-`' means `stdout`. On Tru64, '`z`' means 'position tape after EOF marker', not 'compress'. One can use '`gnutar`' instead.

`tar` tricks

List files in a bzipped archive:

```
> bunzip2 -c x.tar.bz2 | tar -tvf -
```

Copy a directory from one computer to another:

```
> tar -cf - castep | ssh spqr1@remote cd somewhere \; tar -xf -
```

(Note the backslash before the semicolon: we do not want this shell to interpret the semicolon as a command separator. It will remove the backslash, so the remote computer sees a plain semicolon and does interpret it as a command separator. Note too that one can pipe via `ssh`, and that '`-`' as an input file means `stdin`.)

If the network is very slow, one may wish to use compression with the above. If the network is even close to 100MBit/s, don't bother.

If speed is an issue, one may wish to consider `rsh` not `ssh`, but take care.

If you are confused about semicolons, try:

```
> ssh spqr1@remote uname -a \; uname -a
```

and

```
> ssh spqr1@remote uname -a ; uname -a
```

If backing up to CDs, `tar` is not the answer.

Heads or tails?

Two very simple commands: `head` displays the first few lines of a text file, and `tail` the last few.

Display 20 most recently modified files:

```
> ls -last | head -20
```

Display last 40 lines of output:

```
> tail -40 output
```

Lose the summary (first) line from `ls -l`:

```
> ls -l | tail +2
```

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.

Shell Scripts

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 choice is really no choice, for shells derived from the C shell do not allow one to define functions, whereas the Bourne shell introduced functions in 1984. Worse, there is no formal standard describing what the C shell does, whereas POSIX has standardised a Bourne shell. Finally, `/bin/sh` is guaranteed to exist, and no other shell is.

So clearly the C shell loses for portability, it loses for being poorly defined, and it loses for scripts of more than a couple of dozen lines due to its lack of functions.

The result is that no sane person scripts in the C shell, and therefore there is no corpus of decent scripts for one to learn from by example. Also, being little used for scripting, C shells are likely to be more buggy.

Other scripts

Although the C shell is almost never the correct language to use, the Bourne shell is not invariably correct either. 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 derivates 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` (like `csh`) 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
```

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.

Testing times

The command `test` allows one to test most aspects of file existance, 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 `]` preceeded 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 comparisions can be negated with a leading '!', such as `[ ! -d /temp ]`

See '`man test`' for more.

Expressing oneself

The `expr` command performs simple arithmetic and string matching functions. It is fussy about spaces.

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

At this point 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.

Again, see ‘`man expr`’ for more.

Rotating

Log rotation usually consists of taking a log, renaming with a ‘.1’ suffix and compressing it, and also moving the previous version, ‘.1.gz’ to ‘.2.gz’, etc.

```
#!/bin/sh
keep=6
[ -f $1.$keep.gz ] && rm $1.$keep.gz
x=$keep
y=‘expr $x - 1‘
while [ $y -gt 0 ]
do
  [ -f $1.$y.gz ] && mv $1.$y.gz $1.$x.gz
  x=$y
  y=‘expr $x - 1‘
done
mv $1 $1.1
touch $1
gzip $1.1
```

Just like the C language, the shell always *short circuits* the conditionals and (&&) and or (||). That is, for and the second expression is evaluated only if the first is true, and for or only if it is false.

For loops

```
#!/bin/sh
for f in 3 6 9 10 13 33
do
  rsh tcm$f status
done
```

should display the status of TCM's XP1000 machines.

The `csh` equivalent is

```
#!/bin/csh
foreach f ( 3 6 9 10 13 33 )
  rsh tcm$f status
end
```

Try using both in an interactive shell (`bash` and `tcsh` respectively), and then press cursor up, and see which you prefer…

Remote printing

```
#!/bin/sh
target=tcm0.phy.cam.ac.uk

usage(){
  echo ’Usage:’
  echo ’ slpr [-Pprinter] [filename]’
  exit
}

[ "$1" = -h ] && usage
printer=
if expr "$1" : -P. > /dev/null
then
  printer=$1
  shift
fi
if [ -n "$1" ]
then
  if [ ! -r $1 ]
  then
    echo "Error: unable to read file $1"
    usage
  fi
fi
cat $1 | ssh $target lpr $printer
```

A different angle

```
#!/bin/sh
target=tcm0.phy.cam.ac.uk
printer=

if [ $# -gt 0 ]
then
  case $1 in
  -h)
    echo ’Usage:’
    echo ’ slpr [-Pprinter] [filename]’
    exit
    ;;
  -P*)
    printer=$1
    shift
    ;;
  esac
fi
if [ -n "$1" -a ! -r "$1" ]
then
  echo "Error: unable to read file $1" 1>&2
  exit 1
fi

cat $1 | ssh $target lpr $printer
```

Other languages

```
> less unix2dos
#!/bin/sed -f
s/$/^M/
> ./unix2dos file.unix > file.dos
```

Recall how #! is interpretted. The above command is equivalent to

```
> /bin/sed -f ./unix2dos file.unix > file.dos
```

Whereas the shell will accept a script file merely as its first argument, `sed` expects scripts to be preceeded by '`-f`'. Any further arguments are then passed after the #! magic has first added the name of the script file.

Other scripts

Many more ideas can be found from:

The contents of `/etc/init.d` (see later)

The `xon` command (usually a script)

```
> cd /usr/bin; file * | grep script
```

(and similarly other directories on the `$PATH`)

Beware! Some will have errors…

Numquid potest cæcus cæcum ducere?
Nonne ambo in foveam cadent? (Luc. VI xxxix)

Filesystems

Filesystems

One very important feature of an OS is that of producing the concept of a filing system from a disk drive.

Disk drives do not store files. They store blocks of data. The blocks are typically 512 bytes, and the commands between the computer and disk drive look like:
Give me block number 43578
Write these 512 bytes to block 1473

A disk drive has no concept of a ‘file’.

Different operating systems conjure files out of disk drives in different ways, depending partly on perceived requirements.

Files: the requirements

The following items are almost essential for any filing system:

- a concept of a ‘file’ as an ordered set of disk blocks.
- a way of refering to a file by a textual name.
- a way of keeping track of free space on the disk.
- a concept of subdirectories.

The data which describes the layout of the files on disk is called ‘metadata’, as opposed to the plain data which the files contain.

Files: the options

The following items are useful, but not so essential:

- last modification timestamp
- last access timestamp
- ownership
- access permissions
- symbolic links
- quotas
- identication of file type

Files: the limits

Inevitably limits creep in when one tries to construct a filing system. Such limits might include:

- largest possible filesystem
- largest possible filesize
- longest possible filename component
- longest possible complete path
- smallest allocation unit
- characters which may not occur in a filename
- maximum number of files per directory
- maximum number of files in total

FAT

An old and simple filesystem is that from DOS: FAT. It is still widely used, and, in its VFAT form, is used by almost all floppy disks and USB mass storage devices.

From the above list of options, it supports only a last modification timestamp, and a simple read-only flag. It has no concept of multiple users or ownership.

Plain FAT uses upper-case only filenames which are a maximum of eight characters long, followed by a dot, and then an extension of up to three characters. The extension is meant to indicate file type. The legacy of this naming convention is still widespread: `.htm` for `.html`, `.jpg` for `.jpeg`, `.mpg` for `.mpeg`, `.tif` for `.tiff`, `.tgz` for `.tar.gz` are all attempts to get within the three character extension limit.

HTML = HyperText Markup Language; JPEG = Joint Photographic Experts' Group;
MPEG = Motion Picture Experts' Group; TIFF = Tagged Image File Format

VFAT

VFAT (introduced in Windows 95) adds one important extension to the FAT system: filenames can now be mixed-case and 255 characters long.

Another late introduction was (V)FAT 32. Previous forms of FAT had divided the disk into 2^{12} (FAT 12) and 2^{16} (FAT 16) allocation units, and FAT 32 makes the obvious extension and permits filesystems of more than 2GB.

Floppy disks use FAT 12, whereas anything with a capacity of more than 2MB uses FAT 16 or FAT 32.

UFS

The UNIX File System exists more on paper than in reality. Most vendors implement something which is mostly compatible with the specification, but which differs in detail between vendors. So although Tru64, Solaris and Linux all support a filesystem which is effectively UFS, one will not be able to read a disk made with the other.

Unlike FAT, UFS does store file ownership and last access times. It also has a more flexible, case-sensitive, naming system. However, the '/' character is the directory separator, and can never appear in a UFS filename.

Many of FAT's problems arise from its use of fixed-length directory entries (32 bytes each) which store not only the name (11 bytes!), but also the starting point of the file on the disk, the last modification time, the file size, and therefore most of the file's metadata.

Almost all filing systems do not permit their directory separator to appear in a filename. Hence FAT bans \, UFS /, and MacOS :. Most also ban ^@ (nul), C's end of string marker. FAT bans many more too (:<>?*|).

Index nodes

UFS uses index nodes for a file's metadata. When a filesystem is created, a certain number of index nodes are created in a table, and this for ever limits the number of files which can be stored on the disk. Each *inode* entry is typically 128 bytes, and stores all the file's metadata *apart from its name.* These metadata include:

File length
File ownership (user and group)
File 'creation', modification and last access times
File access permissions
Number of directory entries pointing at this file (link count)
A list of the first ten clusters occupied by the file
Three pointers to clusters containing details of further clusters used

Directories

A directory entry under UFS is exceptionally simple. It consists of a name, and the corresponding inode number. With most implimentations of UFS, the directory entry is of variable size, in order to allow long filenames efficiently.

The directory itself is stored on disk as though it were a normal file (it itself has a corresponding inode). The only distinction is that a single bit in the inode entry is set to indicate that the inode refers to a directory, not a data file.

The ‘..’ entry is explicitly stored in the directory, and points to its parent. Likewise ‘.’ points to itself.

FAT is similar in this respect: subdirectories are effectively files with one bit set in their own directory entries.

Fun with inodes

With UFS, directory entries are mostly for the enjoyment of humans. When a human names a file he wants opened, the directory structure is used to convert the name to an inode number, and everything else is done via the inode.

The `mv` command, when possible, moves files by changing directory entries but leaving the inode number unchanged. Thus it is often possible to use `mv` on an open file without the application which has it open noticing.

Two directory entries could point to the same inode, and this is permitted by the *link count* in the inode. Deletion decrements the link count, but the inode is only marked free, and likewise the blocks occupied by the associated file, when the link count reaches zero.

If the `mv` command is used to move between different filesystems, then it falls back to a copy followed by a delete.

The consequences of choice

Because FAT stores so much metadata in the directory, all forms of `ls` (`dir` in the DOS world) are as fast as each other: all simply require the directory to be read.

With UFS, a simple `ls` is still fast and efficient, but even `ls -F` (or a ‘colour’ `ls`), which indicates the difference between files and directories, is much slower, for not only does the directory need to be read, but, for each entry, the corresponding inode needs to be read in order to determine whether the entry relates to a file or a directory.

Progressing to `ls -l`, one not only needs the inode for type, link-count and size, but also a user-id lookup to convert the numeric user-ids stored in the inodes to a textual form. Thus `ls -l` can be *much* slower than a simple `ls`.

Permissions

The standard read, write and execute permissions for owner, group and others are familiar. A few subtleties are sometimes missed.

If the process's uid matches the file's, the user permissions are applied; else if one of the process's groups matches the file's, the group permissions are applied; else others. This is true even if the permissions for 'others' are more generous than the first match.

To make a script executable, it needs `rx` permissions, whereas a binary file needs just `x`.

To access a file or subdirectory, execute permission on (all) parent directory(s) is necessary. To list the contents of a directory, read permission for that directory is required.

If one knows the name of a file, execute permission on its directory is sufficient.

An open and shut case

Most operating systems have the concept of opening and closing files. A file must be opened before it can be read or written to, and should be closed when no longer required (and will be closed when the application exits). On opening a file, it is possible to specify whether one wishes to read or write to the file.

With UFS, opening is one of the few operations to involve the directory entry, and closing does not. With FAT, closing may need to write updated file length and modification times to the directory entry.

The kernel can keep a count of how many applications have a certain file open simultaneously. ‘Real’ file deletion, the freeing of space occupied, occurs only if this count is zero as well as the link count in the inode.

A file held open by an application, but deleted from all directories, will not be really removed until the last application closes it. However, it is very hard for another application to open it, as it has no corresponding directory entry.

Locks

Bad things can happen if two applications try to write to a file at once. Even one reading and one writing can be bad. Thus applications can request that the OS gives them exclusive access to a file. UNIX calls this ‘locking’, and Windows, being positive, ‘sharing’.

If an application which has locked a file dies, its lock(s) are immediately freed. If it simply goes into an unresponsive sulk, then the locks are not freed. . .

Locks exist in the kernel’s memory, not on the physical disk, and therfore they do not survive reboots.

Consider what will happen if I change the title of this document to reflect that published in ‘The Reporter’. By altering the length of a line near the begining, the text editor may feel forced to truncate the file at that point, and write out again everything following. If a second program tries reading the file the instant the truncation occurs, it will get a rather odd result.

Free space

So that space for new, or growing, files, can quickly be found, all filesystems maintain a simple structure indicating which blocks are free. This also enables commands which report the total free space to work efficiently, although they may be aided by another entry simply giving this value.

The UNIX command `df` indicates the free space available on a filesystem:

```
>  df -k .
Filesystem 1k-blocks    Used Available Use%
/dev/hda2    1548096 1357860    111520  93%
```

The numbers do not add up ($1357860 + 111520 = 1469380$) because some space is usually reserved for root only with UFS.

That `df` command in full:

```
>  df -k .
Filesystem 1k-blocks    Used Available Use% Mounted on
/dev/hda2    1548096 1357860    111520  93% /
```

Consistency

A consistent UFS filesystem has the following properties:

- Each inode's link count corresponds to the number of directory entries pointing to it
- Each block marked 'in use' in the free block table is associated with an inode
- No block marked 'free' in the free block table is associated with an inode
- The file length in each inode is consistent with the number of blocks allocated to the file
- Each subdirectory's '..' entry points to its parent
- Each subdirectory's '.' entry point to itself

Inconsistency

A filesystem cannot hope to remain consistent. Simply deleting a file requires a directory entry to be deleted, the inode to be marked free, and the blocks occupied by the file to be marked free. These actions cannot happen simultaneously, so the filesystem may be consistent before and after, but no during, a deletion. The same is true for many other operations.

UNIX assumes the worst. It marks a filesystem as being *dirty* (potentially inconsistent) when it first considers writing to it. If the system is shut down in a controlled fashion, it ensures all operations have completed, and then makes the filesystem unavailable and marks the filesystem as being *clean* again. If the system crashes (or suffers a power loss), when it is next booted the dirty flag will still be set, and the system will check the filesystem's consistency.

Modern versions of Windows behave in a similar fashion. However DOS and early versions of Windows fail to detect improper shutdowns

Automatic repair

Windows has `scandisk` and UNIX `fsck` which can detect and repair filesystem inconsistencies. It is worth pointing out that consistency and correctness are different: formatting a disk also reduces its filesystem to a consistent state, but in a slightly unhelpful manner.

It is also quite possible to get a filesystem into a state which these utilities cannot cope with, or, indeed, where they make matters worse.

Both are quite good at finding ‘lost’ files: files with no directory entries, but still with space allocated to them. In the UNIX world, these are retrieved to the corresponding `lost+found` directory.

fsck = File System CHeck

Fragmentation

When accessing a disk drive, reading consecutive blocks of data is much faster than reading blocks scattered randomly over the physical surface of the disk. It typically takes around 5ms to move the disk heads to a new location, and a further 2ms for even a 15,000rpm drive to complete the typical half-revolution necessary to find a piece of data. By contrast, consecutive disk blocks will be read at around 100 blocks/ms.

The filesystem will usually record in 4K or 8K blocks, not the 512 byte blocks which the disk may well suppport, but it is still very helpful to keep blocks which are consecutive within a file consecutive on disk. UFS tries to do this, whereas DOS-based systems default to an allocation scheme which is extremely poor, as it always takes the first free block regardless of where any other blocks in the same file are.

The `fsck` and `scandisk` programs usually report fragmentation as they run, each fragment of a file being a run of consecutive blocks on disk.

Journalling filesystems

Because checking filesystem consistency is painful on large fileservers – it can often take over an hour – various filesystems which never need a full consistency check have been developed.

They all work by keeping a log, or journal, of operations which they are about to do. Deleting a UNIX file might be broken down as:

```
write to journal ‘I am about to remove this
  directory entry, free this inode, and mark
  these clusters as free.’
do the above
remove the journal entry
```

After a crash, the journal is scanned and those entries which have not been completed are finished.

A journalling filesystem must flush the journal from cache to disk before attempting the updates described by the journal.

Digital UNIX has AdvFS as a journalled filesystem, Irix has xfs, AIX has jfs, Linux has ext3, Solaris ufs (with logging),and WinNT has NTFS. SGI and IBM have donated xfs and jfs to the Linux project.

Journal problems

Journalling produces a significant performance penalty, as every write is turned into two: one to the journal, and one to the real file. For this reason most journalled filesystems only journal metadata.

Journalling metadata can ensure that the filesystem remains consistent, and guards against the type of errors which can cause whole directories to vanish. The contents of files can still be corrupted by crashes.

Journalling data as well as metadata is a serious performance penalty, and requires a much bigger area for the journal. Many journalling filesystems do not support data journalling at all.

The final problem with journalling is that hardware errors or bugs in the OS can still cause a journalled filesystem to become inconsistent. Because the recovery tools for journalled filesystems are used less frequently, they tend to be less tested and less effective.

Linux's ext3 and Solaris's UFS support journalling and still use the same layout as the older, non-journalled, filesystem they are based on. Hence the old recovery tools are valid.

Remote files

It is often convenient to use files physically located on a remote computer as though they were stored locally. This UNIX, MacOS and Windows can all do, and you do every time you use TCM's computers, for your UNIX home directory is physically on one machine, and your Windows home directory another. Neither of these two servers do you ever touch directly.

That UNIX, MacOS and Windows use three completely incompatible protocols for this will be no surprise.

In all cases there is a speed and reliability penalty to pay compared to local disk access, but the increase in convenience can be great.

Disk drives not only typically have a higher bandwidth than networks, but also a lower latency, especially once the overheads of going through the networking protocols is considered.

Remote trouble

An obvious performance increase for local disks can be achieved throught *caching*. The kernel knows that only it can modify the contents of the disk, so it can store recently accessed data, and much of the metadata, in RAM, and use that fast copy for most purposes, writing changes to the disk only occasionally, and never reading the disk just to check that the disk agrees with a copy of some data which the kernel already had in its cache.

All OSes do this: UNIX, Windows, even DOS, and simply caching the most important metadata can make a significant difference to performance, as the physical latency of a disk drive is around 10ms, and of the memory system around 100ns. Although software overheads reduces this difference to a mere factor of 1,000 or so, it is still huge.

But caching fails with network drives. The client has no way of knowing if the server, or another client, has modified some data, and therefore must assume that the worst has happened.

Remote confusion

Losing all caching is too big a penalty, so most clients will cache metadata from remote drives for a few seconds for reads, and not at all for writes. This can cause amusement when caches become stale.

Locking is a bigger problem. With a local process and a local drive, if the process dies the kernel will know, and will be able to free the lock. With a process and a remote drive, the local kernel might not even know that the lock has been requested, as the process was, in some sense, communicating directly with the remote server. Hence when the process dies the local kernel is unware it should return some remote locks. Or it might know, but a network problem might prevent it.

Or perhaps the server suddenly reboots. The local process can survive this, but how can it be told, reliably, that it has lost its lock? With entirely local access, if the computer reboots, processes don’t survive.

NFS

The UNIX protocol for remote file access is called NFS (Network Filing System). It isn't quite the same as UFS. Important differences include:

NFS v1 and v2 have a 2GB filesize limit.

NFS has no real concept of a file being 'open', so a file is deleted as soon as its link count becomes zero, regardless of whether some clients think they have it open still.

NFS's attempt at providing locking is interesting and fraught.

The good thing is that applications are usually unware of whether they are using a local UFS filesystem, or a remote NFS one.

NFS's unencrypted, IP-based, client-side authentication 'security' model is a complete joke. It is not, in general, suitable for storing passwords or anything else vaguely sensitive.

NFS quirks

Listing all the infelicities of NFS would take forever. However, just one for amusement. Suppose an NFS client has a file open, and deletes it while it is open. What should it do?

Given that the same machine has the file open and is trying to delete it, it can be expected to notice the problem. Standard UNIX semantics say that the delete should succeed in removing the directory entry, but that subsequent access to the open file via its inode should also succeed.

What the client actually does is that it causes the delete operation to rename the file to the a name starting '`.nfs`' and ending with a 'random' number. Humans are content, as they see the file disappear. Applications are content, because they can still use their open file in safety. The world is not content, because these `.nfs` files are not deleted when the original file is closed, and simply accumulate.

Special files

The basic programming interface to files: opening, seeking to a given position, and reading or writing, is so simple and convenient that it is useful to be able to apply it to other things too. This UNIX does.

The directory `/dev` contains *device files*, each of which corresponds to (usually) a hardware device, and each of which obeys standard file permissions. Noteable entries include each disk drive (for reading raw blocks), tape drives, serial, USB and parallel ports, sound cards, and physical memory. Most can be read only by root.

The directory `/proc` contains information about processes, and the machine in general. These files have no physical existance on any disk, but the kernel interprets attempts to read from them as requests for certain information, and supplies same.

One should be very careful if one attempts to backup files in `/dev` or `/proc`. One, `/dev/zero`, contains an infinite number of zero bytes, and backing it up by opening it and reading its contents will take a very long time indeed.

`/proc` on Linux

Linux has a comprehensive set of files under `/proc`, almost all of which can be safely prodded with the `less` command. Being slightly unreal, many report zero size to `ls -l`, although they contain data.

`/proc/cpuinfo`
CPU type and speed.

`/proc/meminfo`
Memory size and utilisation (used by `free`).

`/proc/PID/cwd`
Symbolic link to process's current working directory.

`/proc/PID/exe`
Symbolic link to process's executable file.

`/proc/PID/cmdline`
Process's command line.

`/proc/PID/status`
Process's memory use, uids, gids, ppid, and signal info.

`/proc/PID/maps`
Process's memory map.

Multiple filesystems

DOS, Windows and MacOS present each filesystem to the user as a separate ‘disk drive.’ With DOS, they are called friendly things like `C:`, `D:` and `E:`, whereas MacOS pops up icons with configurable textual names.

UNIX does things rather differently. It presents a single directory tree with a single root directory. Different filesystems are then grafted on to that tree using the `mount` command. On a typical TCM Alpha, there are three filesystems resident on local disks: `/`, `/usr` and `/temp`. There are also several remote filesystems including `/u/tcmsf1` (where the home directories reside) and `/usr/local/shared` (where many applications are to be found).

The joins between these filesystems are almost invisible to the user.

‘`df -k .`’ will tell you where you really are.

‘`umount`’ unmounts a filesystem, and ensures that all changes are flushed from the caches to the disk, and that the filesystem is marked ‘clean’ (consistent).

The Internet

The Internet

The Internet is a global network based on the IP protocol and to which computers in TCM are attached.

Many different types of machine are connected to the Internet, and the network itself travels over various media (copper, fibre and wireless) using many protocols (ethernet, ATM, modems, etc.)

The design is a flexible layered design, with each layer specifying as little as possible about the others.

Ethernet

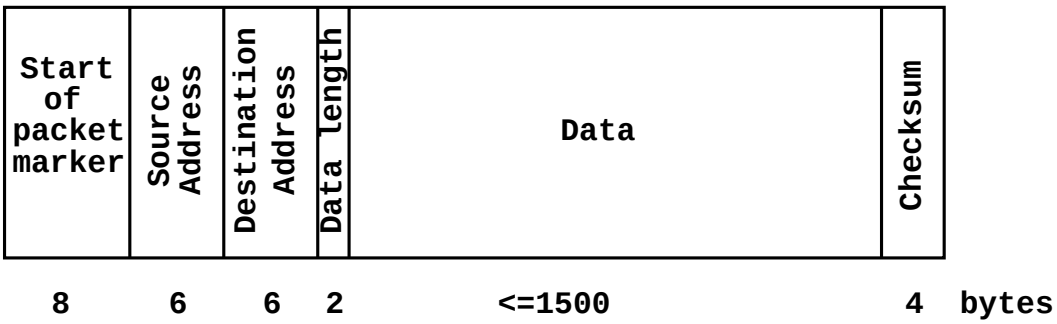
Ethernet is the protocol on which the TCM network, and almost every network in Cambridge, is based. It runs over both fibre and copper at several different speeds, from 10 MBit/s to 1 GBit/s.

It is the lowest level of the protocols used, and very few programs use it directly. It is useless for communication beyond TCM.

Like most networking protocols (from carrier pigeons onwards), it is based on the transferal of packets of data.

Other possible network protocols which are alternatives to ethernet include ATM and TokenRing. The latter is firmly dead, but ATM has advantages over ethernet when long distances are involved.

An Ethernet Packet



The addresses refered to are the ‘unique’ 6 byte MAC addresses which every ethernet card must have, not the IP address.

There is no protocol for acknowledging receipt of a packet, and the network is quite likely to throw the packet away before it reaches its destination anyway.

The destination address must be on the same physical network as the source: long-distance routing is impossible.

A useful system for a few hundred computers, but thereafter increasingly unworkable.

An IP packet

The next protocol level is IP (Internet Protocol). Such a packet has a header of 20 bytes, the important fields of which are:

Source address (as IP address) (4 bytes)
Destination address (as IP address) (4 bytes)
Packet Length (2 bytes)
Checksum of header (2 bytes)
Protocol (2 bytes)
Time To Live (2 bytes)

The TTL field gets decremented by one every time the packet passes through a router, and the packet gets discarded if it reaches zero. This prevents packets wandering aimlessly around the world for ever.

When an IP packet is sent over an ethernet network, it is enclosed in the data section of an ethernet packet. Other ethernet packets on the same network may be transporting other protocols, such as DECNet, NetBEUI or Ethertalk.

Local or remote?

It is useful to be able to reach beyond one's local network.

A correctly configured computer will know which IP addresses correspond to machines on its local network, and, for those which don't, the IP address of a *router* or *gateway* that will forward the packets on.

Thus a computer is always sending out packets either directly to another computer on the same physical network, or to a router on its physical network which will know where to send the packets next.

The local physical network stops at the nearest router.

A router will usually cope if it is sent a packet unnecessarily: that is, it will simple return the packet to the same network it came from, but addressed to the correct computer this time.

Some routers (including TCM's) also look for packets lost on the wrong network and rescue them. Details to follow…

IP or MAC Address

An important extra step is the translation of a numeric IP address to an ethernet (MAC) address. The protocol for resolving this, called ‘ARP’, is depressingly simple.

A computer sends out a broadcast ethernet packet containing the question ‘who is 131.111.62.175?’ (Broadcast: to all machines on the local network, specified as an address of all ones.)

The first reply it receives reading ‘I am 131.111.62.175, my MAC address is 00:06:2b:00:ca:b7’ is believed.

Security? None. All computers were explicitly sent the query: any could have chosen to reply.

This is the point where a helpful router can see a request saying ‘who is 131.111.8.12?’ and think ‘no-one around here, you fool: give it to me, I’ll handle it.’ It thus replies to the ARP request on behalf of 131.111.8.12 giving its own MAC address so that it will get sent the real packets. Such a reply is called a *proxy ARP* for obvious reasons.

Ethernet cards will respond to packets sent to their own address (e.g. 00:06:2b:00:ca:b7) or to the broadcast address (ff:ff:ff:ff:ff:ff). Broadcasts are inefficient as they must be transmitted to every machine. Unicast packets can be kept away from machines which do not need them.

Routing

One routes packets, one routs armies.

Yes, dear American friends, these words are pronounced differently.

A router connects to a small number of networks. If it is sent an IP packet, it knows to which network to send it so that it can move towards its destination. Sending it on will involved re-encapsulating it, maybe in an ethernet packet with different source and destination addresses, maybe in (several) ATM packets, etc.

The router will decrement the TTL field of the IP packet, sending back an error and not forwarding the packet if it reaches zero. It may also return an error if it does not know how to reach the requested address.

Addresses are allocated in blocks. Hence anything of the form 131.111.xx.yy can be sent to this University for further routing, and anything of the form 131.111.62.zz will end up in the Cavendish. So no, your College IP address will not work in TCM!

Telephone systems work in a similar fashion, but on a larger scale. Unlike ’phone systems there is no national code: 131.112.2.zz is in Japan!

Networks

Specifying an individual IP address is simple: 131.111.62.129.

To specify a contiguous block of addresses, that is to say a network, in the bad old days one used classes, depending on which bytes were ignored, so 131.111.62.zz would be called a ‘class C’ network, and have 256 addresses, and 131.111.xx.yy a ‘class B’ network with 65,536 addresses. Class As are rare.

These can also be specified using a *netmask*, as 131.111.62.0/255.255.255.0. This is more flexible, as the netmask, in binary, has ones for the bits which are constant in the network, and zeros for those which can change. One is now permitted *classless* networks such as 131.111.62.128/255.255.255.128, which is half a class C.

As the netmask must have the form, in binary, (bunch of ones)(bunch of zeros), it is quicker to specify just the number of ones. Hence 131.111.62.128/25 or 131.111.0.0/16.

Public and Private Addresses

There are $2^{32} \approx 10^9$ possible internet addresses. Most of these are routed around the world in a conventional fashion. Cambridge has three class B networks.

There are also addresses which are defined to be private to an institution. That is, an institution (in this case the University) may use them on its private network, but they should never appear outside, and, indeed, are meaningless outside as different institutions may well be using identical addresses. Such addresses are fine for printers and other things which really do not need an address which enables them to be contacted from the other side of the world.

These addresses are sometimes called *RFC 1918* addresses, after the standard which defines them. They include 10.0.0.0/8, 172.16.0.0/12 and 192.168.0.0/16.

Private addresses in Cambridge

The University (and JANet) have divided the RFC 1918 addresses as follows:

- reserved: 10.128.0.0/9
- routed across the CUDN: 172.16.0.0/12 except 172.31.0.0/16
- not routed anywhere: 10.0.0.0/9, 172.31.0.0/16, 192.168.0.0/16

This greatly increases the number of addresses available: for every public IP address the University has five CUDN-routed private addresses and over forty unrouted ones.

CUDN: Cambridge University Data Network – the internet within the University.
JANet: Joint Academic Network, the UK's govt funded academic network.

Addresses in TCM

We have one public range: 131.111.62.128/25.
We have one CUDN-routed private range: 172.24.25.0/24.
We, with BioPhysics, have all the ‘unrouted’ ranges on the previous slide.

Thus the routing table for a TCM machine might specify 131.111.62.128/25 and 172.24.25.0/24 as local networks, and 131.111.62.190 (or 172.24.25.62) as a *default gateway* which knows what to do with all other destinations.

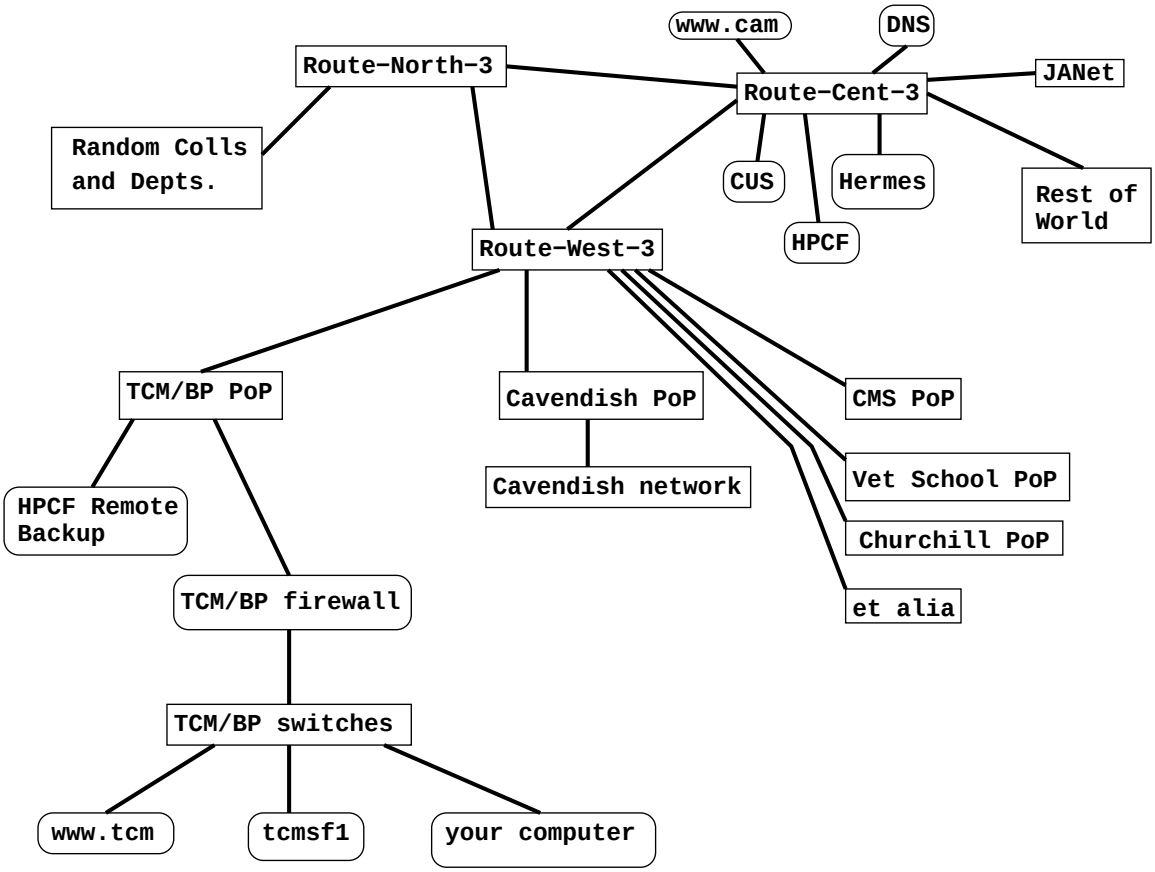
Note that the gateway must be within (one of) the local network(s), otherwise it is unclear how to reach the gateway…

BioPhysics has 131.111.75.192/26 and 172.24.125.0/25 as its public and private ranges.

The terms ‘gateway’, ‘default gateway’ and ‘default route’ are used interchangeably.

The command ‘`netstat -nr`’ displays a machine’s routing table, and `route` modifies it.

The World



Various other central CS routers are not shown, e.g. `route-sidg-3` (Sidgwick site), `route-down-3` (Downing site) and `route-south-3`.

A ‘PoP’ is a Computing Service Point of Presence: a physical network switch which it manages and an institution plugs into. Most of these are omitted from the above diagram.

The CS routers are named by their own physical location. Hence Wolfson College is connected directly to the Sidgwick router, and Hughes Hall to the Downing router.

`Route-West-3` lives physically just beside Reception at the Cavendish.

Helpfulness

If, in TCM, 131.111.62.129 wants to talk to 172.24.25.1, it should just broadcast an ARP request on the local network, and talk to it directly. If it doesn't realise that its target is on the same network, it will send the packet to its gateway instead. The gateway is sufficiently intelligent to realise that the packet needs returning to the same network from which it received it (but with a different ethernet destination address) and will do so. However, bouncing traffic off the gateway like this is inefficient, as the gateway will surely be a bottle-neck.

The converse is 131.111.62.129 wanting to talk to 131.111.8.1, and simply sending an ARP request to the local network. The broadcast cannot possibly reach the intended target sitting on a quite distinct network, but the gateway will see the broadcast, notice the error, and reply on behalf of 131.111.8.1. Thus the gateway gets sent the traffic, which is what should have happened anyway.

This form of 'fake' ARP reply is called *proxy ARP*.

Say that again?

Yes, the previous paragaph does read as follows:

131.111.62.129: could 131.111.8.1 please give me its ethernet address, so that I can send data to it?
131.111.62.190: here, this is my ethernet address, I am 131.111.8.1.
131.111.62.129: yipee! I've found my friend: now I'll send him that data.

The silly fool called 131.111.62.129 is simply unaware that the machine claiming to be 131.111.8.1 isn't. If one has access to the local network, intercepting IP traffic in this fashion is incredibly easy.

Naturally anyone caught playing such tricks can expect to suffer an involuntary change of employer or place of work.

Caching ARP

It would be extremely inefficient to have to send out an ARP request for every IP packet, particularly as the ARP request is *broadcast* to all machines, rather than being *unicast* to a single machine. Thus when a machine receives an ARP reply, it will remember the result for a while. Occassionally (minutes) it will check that nothing has changed (if it is still using that IP address) by sending another ARP request, initially just unicast to the address which it expects will respond ‘That’s me!’

The contents of a computer’s arp cache can be listed with the command

```
> arp -a
```

If an IP address changes its ethernet address (e.g. replacement after ahrdware failure), it can take some machines an hour or so to time-out the incorrect entry in their cache and to go looking for the correct one again.

ICMP

A simple IP protocol is ICMP: Internet Control Meta Packet. It is used for diagnosing faults on the internet and for keeping the internet running smoothly.

A common use of ICMP is by the program '`ping`'. This sends a packet with the request that it is echoed straight back. One can thus see how far packets are getting, how long they are taking, and how many are being lost.

One should not make excessive use of `ping`, but, when network problems are apparent, it can be useful to see if one can 'ping' a machine on the local network (should give a time of 1-2ms and no losses), one's gateway (ditto), a slightly more distant machine (e.g. `cus`), something in another town, etc.

Many commercial networks block 'ping' packets. In Cambridge, ITS Rules require that connected machines respond to ping packets from 131.111.8.0/25.

TCM's gateway is 131.111.62.190.

Most versions of ping (but not Solaris's) run for ever. Press control-C to stop them.

TCP and UDP

TCP and UDP are the two protocols which most widely used. UDP is the simpler, and is mainly used by NFS, DNS and some logging facilities. It has a header of just eight bytes, containing four two-byte fields: source port, destination port, length and a checksum.

The port number, used by both TCP and UDP, is used by the machines at either end of the connection to determine which program should handle the incoming packet. It is a 16 bit integer. Common values include:

21/TCP	ftp
22/TCP	ssh
23/TCP	telnet
25/TCP	SMTP (mail server)
79/TCP	finger
80/TCP	WWW server
123/UDP	NTP (clock synchronisation)
161/UDP	SNMP
515/TCP	lpd (printing)
993/TCP	IMAP/SSL (secure email)
6000/TCP	X11

TCP

TCP has a twenty byte header, again containing source and destination IP addresses and ports, but also sequence and acknowledgment numbers, and various flags.

Ethernet, IP and UDP are all *connectionless* protocols. A packet is sent, it has no explicit relationship to any other packet, it is not acknowledged, it might not even arrive.

TCP is a protocol which establishes a two-way connection over an unreliable medium. Packets do have sequence numbers within an established connection, so the receiver can tell if one is missing or if some arrived out of order, and all packets are acknowledged so that the sender knows if they have been received.

As TCP creates a point-to-point connection, it has no concept of broadcasting. UDP and ICMP can broadcast by specifying an IP address which has all bits in the network-variable part set to one (e.g. 131.111.62.255 for TCM's public address range).

UNIX uses broadcasts rarely, and usually as a result of poor configuration. Windows uses broadcasts much more heavily, particularly heavily in early versions.

Funny handshakes

To establish a TCP connection from A to B, the following exchange occurs:

Computer A sends a TCP packet with the SYN flag set to the relevant port on the remote machine and a random initial sequence number.

Computer B replies either with the RST flag (no program is listening on this port: go away), or with both the SYN and ACK flags set.

Computer A replies with a packet with ACK set to acknowledge B's packet, and with the sequence number incremented. This is the first packet which may contain real data.

Each packet will be resent a small (3-4) number of times if no reply is received after a reasonable time.

ACK: acknowledge (previous packet)
RST: reset (connection)
SYN: synchronise (sequence numbers)

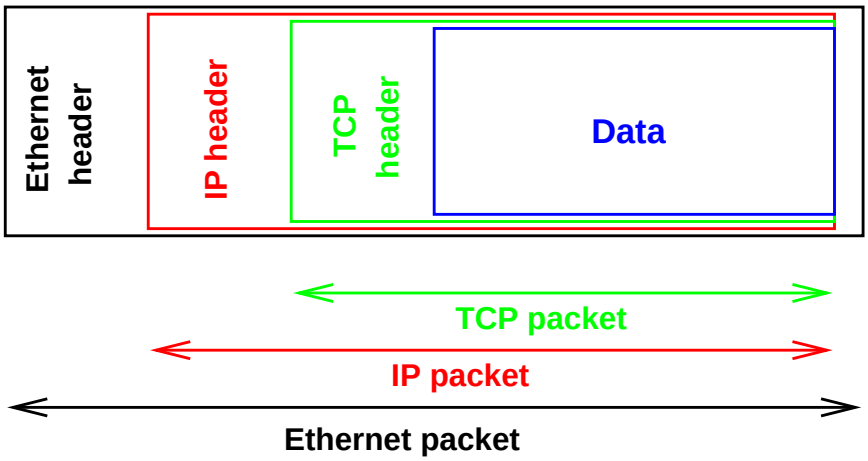
Saying Goodbye

Having established a connection, it can be closed either if one end stops responding for about a minute, or, more cleanly by one end requesting that the connection be closed, and then the other end acknowledging the closure.

If a connection is transmitting no data, neither machine will notice if something breaks. Thus one has the concept of ‘keep alive’ packets: packets sent intermittently (about once an hour or so) which carry no data by require the remote machine to acknowledge them and thus confirm that all is well.

The command ‘`netstat -a`’ lists open TCP connections.

A TCP packet



Remember that the data section of an ethernet packet cannot be larger than 1500 bytes. It is permissible to fragment an IP packet across several ethernet packets, but it can be messy.

Trust

Don’t – a TCP connection could come from anywhere.

Always demand that a password be supplied with each connection, unless anonymous connections are acceptable.

Thus `telnet`, `ftp` and most `imap` and `pop` clients. Pity that none bothers to permit the passwords demanded to be encrypted.

This form of authentication is called *server-side authentication*, for it is the remote server which handles the authentication, and is responsible for verifying that the connection is from the user it claims to be from.

Strictly `imap` does permit encrypted passwords, but this extension is rarely implemented.

Be Trusting

And reduce the number of unencrypted passwords flying around.

Base trust on IP address and user details that client program sends.

One needs to believe the IP address spoofing cannot happen, and one needs to trust the client program.

Thus `rsh`, `rcp`, `xon`, `lpr` and NFS.

And this form of authentication is called *client-side authentication*, for it is the local client which handles the authentication, and the server trusts that the client is behaving.

Be very trusting

Require no authentication: offer service to anyone who requests it.

In many ways ideal: no false sense of security!

Thus http (WWW), SMTP (email(!)), finger, daytime, anon. ftp…

Packets and Processes

UNIX requires processes to use the kernel for all interactions with hardware devices, including network cards. The kernel permits:

Outgoing UDP/TCP with a source port ≥ 1024
Listening for UDP/TCP on a free port ≥ 1024

Anything else (ICMP, sending from ports < 1024, raw ethernet packets) can be done by root only.

UNIX does have a mechanism for giving users root privilege. An executable program can be '*setuid root*', which means, when run, it gains all the privileges of root. This is how the `ping` program is able to send ICMP for non-root users.

There are no restrictions on the port numbers one can connect to, only on those one can connect from or listen on.

```
> ls -l /bin/ping
-rwsr-xr-x  1 root  root  22620 Jan 16  2001 /bin/ping
```

the 's' in the user permissions shows that the program is executable and setuid to the user who owns the file.

Privilege

When `rsh` on machine A contacts machine B and says ‘this is `spqr1` from A,’ how can B tell that `rsh` is really being run by `spqr1`, not `a.n.other`?

Certainly `a.n.other` can compile a version of rsh which will always (falsely) claim to be being run by `spqr1`. But one hurdle remains. It cannot send *from* TCP ports in the range 0-1023. The modified user-installed version of `rsh` will have to use a port number in the range 1024-65535. Machine B will refuse all rsh connections from such ‘non-privileged’ ports.

The real `rsh` is installed setuid root, so is able to connect from a privileged port, and might be trusted by the remote machine.

Neither Windows nor MacOS follow this convention, so user data supplied by `rsh` running on such machines is untrustable.

Cryptography

Cryptography is a marvellous thing. It allows information to be exchanged in such a way that third parties cannot eavesdrop, *a*nd it allows remote machines to prove their identity.

Unfortunately, very few programs use it. Windows NT, ssh and ssl all do a good job though.

Without cryptography, when typing a password you know neither who is eavesdropping, nor that the machine to which you are sending the password is really what it claims to be, unless you fully understand and trust the management of all the networks between the two machines.

The sensibly encrypted protocols include `ssh`, `https` and `imaps`. The last two are simply `http` and `imap` sent though an encrypted *tunnel* using ssl.

Firewalls

Most firewalls, whether implimented on the machine they are protecting, or as a separate machine through which all traffic to a network must pass, are very simplistic. They examine the IP header of each IP packet, and make decisions about whether to forward or reject the packet based on the sender's and recipients IP addresses and port numbers, and the TCP flags.

Possible rules include:
Reject all incoming TCP packets with SYN set and ACK not set (rejects all TCP connections initiated from outside)
Reject all incoming TCP packets to port 6000 (rejects all attempts to contact X server from outside)
Reject all outgoing TCP packets going to port 80 (stop people browsing the WWW)
Reject all incoming packets except TCP to port 22 (allow just `ssh`)

One does need to think hard about what one intends to achieve with a firewall, and which protocols use 'random' ports, before going too far though.

Packets and Hardware

Most network devices examine some part of the packet in order to optimise where they send it next. The list is roughly:

Hubs/repeaters: none
Switches: ethernet destination
Routers: IP destination
Firewalls: IP source and destination, TCP and UDP headers

As one moves down the list, the problem becomes more complex, and the price goes up. However, so does the largest network size one can support.

X11

X11

The X Windowing system is one area of UNIX which is frequently badly misunderstood.

X is a system for the display of graphical applications. It was developed at MIT in the 1980s as a research exercise, not as a application intended for general use.

It divides the problem of displaying graphics into two distinct parts. Firstly there is the X server, which is responsible for getting the images onto the screen, and for reading keyboard and mouse events. Secondly there is the X client, which is the program which sends requests to the X server, and receives information about keypresses and mouse movements from the X server.

The X server

The X server needs to know precisely how to interact with the particular video hardware and keyboard and mouse attached to the computer on which it is running. Once it has initialised the graphics hardware, it just sits listening for requests to display things and requests to be sent keyboard and mouse information. These requests come encoded in the 'X protocol.'

The server really does listen: it listens on TCP port number 6000. Anyone connecting to this TCP port can talk to the server, and in the bad old days there was no further security whatsoever.

A computer will usually run a single X server, usually run as root to permit more direct access to the graphics hardware, and usually written by the hardware manufacturer: hence `Xdec`, `Xsun`, etc.

The X client

The X client is any of the multitude of processes which will use the X server in order to be displayed: your window manager, an xterm, an xclock, `gv`, `xdvi`, `acroread`, `xcalc` are all X clients.

The client will speak to the server using the X protocol, but, because the X protocol is tedious to program, every X client in fact uses the X11 library (libX11.so) which offers simple C functions one can call in order to open windows, select fonts, write text, write bitmaps, draw rectangles and arcs, and do the other simple operations.

The X client knows which X server to contact by looking at the environment variable `$DISPLAY`, whose syntax is `[host.domain]:offset.head`
The optional `host.domain` part specifies which computer to connect to (if absent, the local machine is assumed). The offset specifies the TCP port number, as an offset from 6000. It is usually zero unless one is using `ssh`. The head part is (usually) non-zero only for multi-headed displays.

Toolkits

The standard X library is rather limitted. It does not provide for the standard scroll-bars, menu bars, drop-down menus and other similar items that one might expect. True, one can construct them all out of the primitive functions it does provide, but this is tedious.

Hence various ‘toolkits’ or ‘widget sets’ exist, which are pre-packaged collections of such useful things. Two very common ones are ‘Xt’ (the X toolkit, really rather basic) and ‘Xaw’ (the Athena widgets set) which ship freely with X11. Commerical libraries exist too, such as Motif (libXm). It is often possible to guess which widget set(s) an application uses just by looking at it.

Extending X

The X protocol is usefully extensible, and most X servers support far more than the basic operations of the original X11. The command `xdpyinfo` will return a list of extensions supported by the server. Notably ones include:

MIT-SHM - permit clients to send information to the server via shared memory segments. Increases speed, but works only if client and server are running on the same computer.

DPMS - enables clients to request that the monitor switch to a low-power state (Display Power Management System).

SHAPE - permits non-rectangular windows.

GLX - see next page.

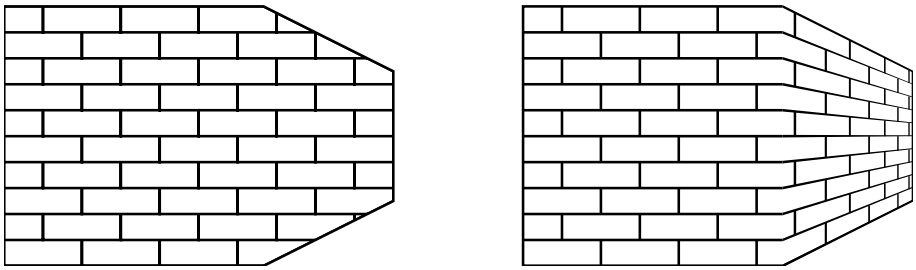
GLX

GLX, or OpenGL, is a set of library routines for doing common operations in 3D graphics, particularly filling triangles with the correct shade of colour given their natural colour and reflectivity and the prevailing lighting conditions, mapping textures onto triangles, and dealing with fog. All this can be done in software, but usually these operations are done in hardware by the video card: the library simply sends a request to the video card, and a processor on the card does the work leaving the computer's CPU free to do other things.

Video cards with this sort of computing power became widely and cheaply available in around 2000. Before then, hardware GL was extremely expensive. Now it is ubiquitous, and driven by the mass games market. Games running under both MS Windows and X11 can use GL for rendering their 3D scenes.

GL surprises

Because the calculations required by the GL operations are done by the video card, and because approximations are permitted, indeed, almost encouraged in order to achieve decent speed, the same program running on machines identical apart from their video cards can display different images. The differences are usually too slight to be worth worrying about.



A brick texture mapped without (left) and with (right) regard to perspective. A GL-supporting graphics card can do either mapping itself.

Window Managers

The window manager is a rather special X client. It is responsible for adding the *decorations* to a window: the border and title bar, and for controlling move and resize operations using them. It is also responsible for any menus which appear when one clicks on the bare background of the X server (the *root window*), for controlling the *focus* policy (which window gets keyboard input) and iconisation.

There are many different window mangers, including `twm` (one of the first), to `fvwm`, `dtwm` (commercial, part of the CDE environment), `mwm` (Motif) and `wmaker`. No more than one application may register itself with the X server as the window-manager at any one time. It is possible (just) to run without a window manager, but do not expect to be able to move or resize windows, or to raise or lower them, or have anything other than the default focus-follows-mouse policy.

xterm

Most people claim to be familiar with `xterm`. It is a simple GUI wrapper for a text application, which, by default, runs one's preferred shell. It can run any text application:

```
> xterm -e tail -f output.dat &
> xterm -e vi talk.tex &
> xterm -e pine &
```

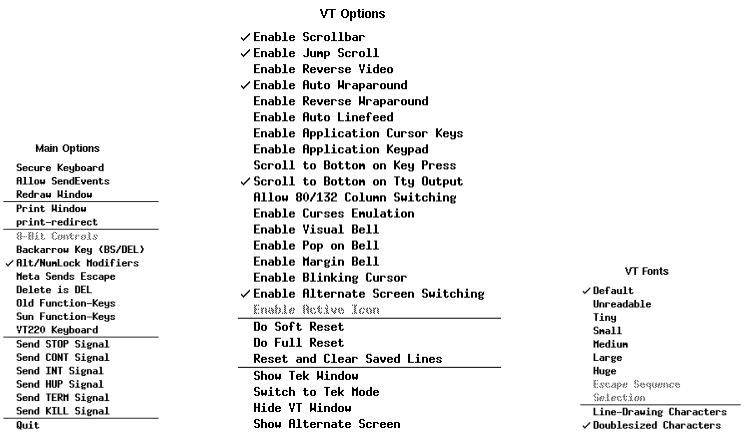
or one's preferred shell *as a login shell*:

```
> xterm -ls &
```

One can also set the number of lines remembered in the scroll-back region:

```
> xterm -sb -sl 500 &
```

It has menus, accessed by {ctrl} and the mouse buttons:



fvwm2

The window manager most people in TCM use on the Linux PCs is `fvwm2`. The Alphas tend to use `dtwm`, and some people prefer `wmaker` on the PCs.

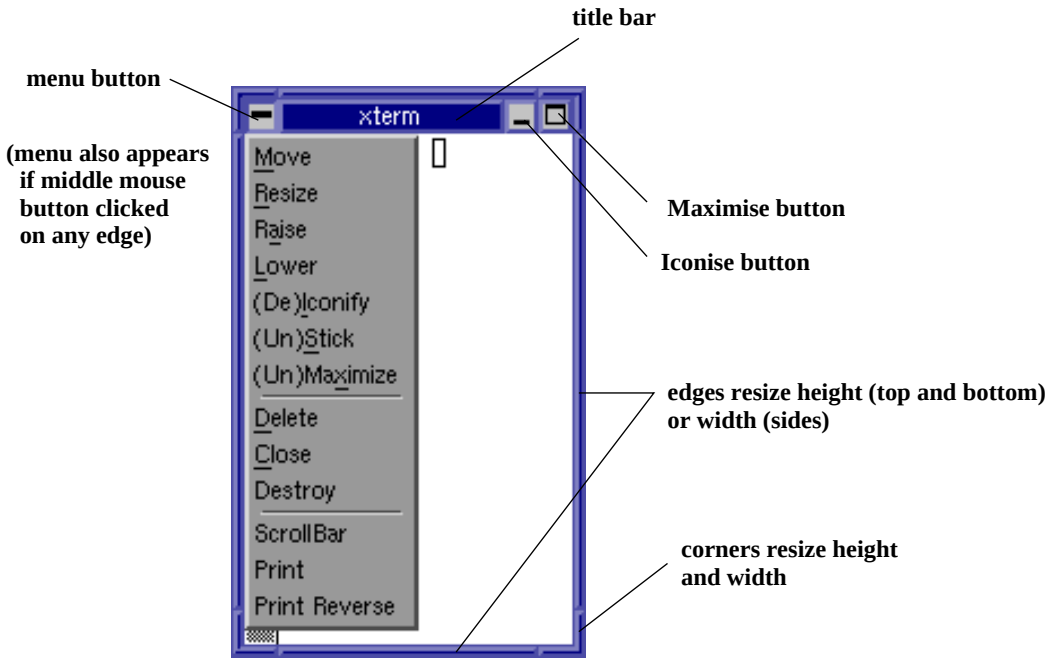
A good window manager is small, configurable, and easy to configure sanely. The above come close. . .

Fvwm2 is smallish, and reads its configuration from `~/.fvwm2rc`, or a system-wide file if that does not exist. It can be configured to support the common focus policies (focus follows mouse, click to focus, lazy focus), it supports multiple desktops, switching between them using the mouse, keypresses and / or a pager, it allows windows to be moved or resized from any part of their frame. It supports iconisation, and configurable pop-up menus when one clicks on the root window, and the closing of errant windows.

Lazy focus: follows mouse, but never switches to root window

fvwm2 continued

‘It does?’ I hear some of you say. Well, in TCM it does.



‘Raise’ and ‘Lower’ will move the window up and down a stack of overlapping windows. ’Stick’ makes the window appear on all virtual desktops.‘Close’ will close an errant window, and ‘Destroy’ ditto but overly forcibly. A double-click on the top left button is equivalent to selecting ‘close.’

Bad window managers, bad clients

Features of some window managers are silly. For instance, resizing from the bottom corners only (useless when the bottom of the window is off the screen), moving the window via the title bar only (similar), using excessive numbers of colours and thus crippling older computers, and having single click to close an application (particularly on a button close to the iconise button).

The worst client is the GUI file manager. Whereas `ls` does what a human expects: it lists the current state of a directory, a GUI filemanager does not: it shows the state when the window was opened.

Updates from other clients, or even programs on the same client, qickly cause the GUI's view to date. So *really* bad GUI filemanagers refresh their displays periodically automatically. This does not guarantee that they are correct when a human looks at them, but does add nicely to the server load.

Does your GUI filemanager distinguish between directories with subdirectories, and those without? To do so, it may need to open each subdirectory and check its contents: much more work than `ls` does. Is it obvious that one should press {F5} to cause Windows to update its filemanager?

(MacOS and Win9x have much to answer for, and are someway behind Windows 3.x and `mwm`.)

More clients

The ability to take a snapshot of the display (or part of it) is clearly useful, and X provides this functionality. The programs `xwd`, `xv` and `gimp`, amongst others, can all do so.

The ability to follow the mouse, even when it is over other windows, can be useful. The program `xeyes` can do this.

The ability to received copies of all keypresses, even when they are destined for other windows, certainly has its uses, and such a program is simple to write.

Security?

Having anyone anywhere in the world being able to contact the X server one is using and request a screen-dump, or a copy of all keystrokes, is a disaster. But this project was designed to demonstrate the feasibility of remote graphics display on a trusted network, not to be routinely used, nor to be anywhere near an untrusted network.

The first improvement was host-based authentication: `xhost`. This restricts access to the X server to a list of machines. This is useful if the machines listed are single-user machines and cannot readily be spoofed. Thus it is fine if the list contains the single entry ‘localhost’, and the machine in question is running Windows. In most other situations it is dodgy.

Magic Cookies

The next improvement was the use of *MIT magic cookies*. These are 128 bit random numbers which the X server generates when it is started or reset. Any connection presenting the currently-valid cookie is accepted.

The cookies, along with the name of the display for which they are valid, are stored in a file called ‘`.Xauthority`’ in a user’s home directory. An X application knows which display to connect to from the value of the `$DISPLAY` variable, and it can then look at the file of cookies to see if there is a corresponding cookie to offer. The command `xauth` can manipulate the `.Xauthority` file. In particular, `xauth list` lists all cookies currently held.

One problem with cookies is that they get transmitted unencrypted every time a new connection to an X server is made. However, if one logs out frequently, they quickly become useless and thus any hacker has a small window of time (hours, not weeks) in which to trap and exploit one.

Failed Magic

These cookies need to be stored in a file in one's home directory, one which is readable only by its owner. However, if one's home directory is unavailable (server half dead), or unwritable (over quota), there is a problem. This normally results in one being unable to log in.

The solution, on Linux machines, is to press {CTRL} {ALT}{F1} for a text-mode login, and see what has happened. {CTRL}{ALT}{F7} usually restores X: if not, try F6 and F8.

On machines running CDE, one can choose a text-mode login from the login screen.

Sometimes running ssh whilst over quota results in one's .Xauthority file being deleted (or truncated to zero length). Then one cannot open any more windows, and must solve the quota problem, then log out and in again.

In general, text mode logins are better at reporting what error occured.

X and ssh

The ssh program does some useful tricks with X. It sets up a dummy X server on the remote machine, usually on port 6010, and sets $DISPLAY to remotehost:10.0 at the remote end of the connection. This dummy server accepts X traffic, and passes it over the encrypted ssh link back to the originating host. The X traffic is then represented to the originating host's X server (actually, whatever $DISPLAY specified on the machine from which ssh was used). Magic cookies are magically changed during this process.

This is ideal: everything happens automatically, and neither the cookies nor anything else are not transmitted uncrypted. However, graphics applications (particularly animated ones) do produce a lot of X traffic, and there may be a noticeable overhead associated with encryption and decryption.

Between machines which trust each other closely, it may be better simply to use xon or, if home directories are shared, to set $DISPLAY manually, if one is worried about performance.

A Process and its Memory

The world of a process

A processes exists in its own private virtual address space. This address space contains different regions or *segments*:

Text
(The machine instructions themselves, and read-only data.)

Data (Initialised data.)

BSS (Uninitialised static data.)

Heap (Dynamically allocated data.)

Shared Libraries (optional)

Stack

Executable files and heaps

The executable file contains the whole of the text and data segments, and a note of the size of the bss segment. As the bss segment contains data which does not need initialising, there is no need for the executable file to contain any real data for it.

The command `size` will show the sizes of these three segments, and totals in both hex and decimal.

The heap is (conventionally) where storage requested with C's `malloc()` function or Fortran's `allocate()` is found. Thus it can grow (and shrink) during program execution, and its maximum size is unknown when the program starts. 'Old' F77 programs make no use of this sort of dynamic storage.

Shared Libraries

Non-shared libraries, e.g. `libc.a`, are added to a program's text segment when the program is linked. Shared libraries, e.g. `libc.so`, are not, nor are they stored in the executable: they are linked dynamically at run-time.

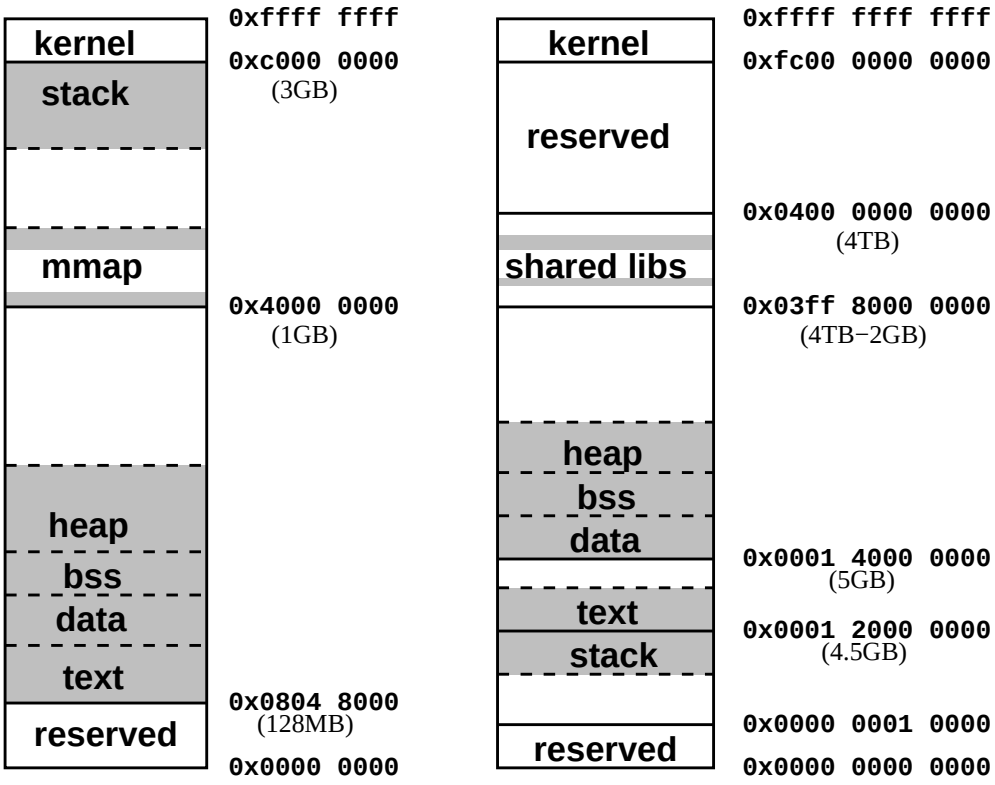
If multiple programs use the same shared library (certainly true with `libc.so`) only one copy will occupy physical memory, with multiple virtual addresses pointing to it.

A shared library loaded at runtime need not be the same as that checked at compile time, or even that used last time the same executable was run. It might be newer, less buggy, and specifically optimised for the given machine. However, there is usually a slight overhead on each call to a shared library function, so sometimes one links statically (i.e. non shared) for speed. The `ldd` command show which shared libraries an executable will use.

Windows uses shared libraries extensively too, calling them `.dll` files.

`.dll` dynamic linked library; `.so` shared object.

Memory maps



The left shows IA32 Linux 2.4, the right Tru64 UNIX.

Linux uses the mmap area for shared libraries, and for large allocations which would conventionally go onto the heap.

Tru64 does not really have a 64 bit virtual address space, but merely a 43 bit one. One could equally draw the map with the kernel directly above the shared libraries, and the highest address being `0x0800 0000 0000`.

Note the very different scales: Linux gives a user process just under 3GB of virtual address space, Tru64 almost 4TB.

Memory maps in action

It is easy to examine the memory map of a running process on most modern UNIX systems. Linux is simplest, particularly if one recalls that the shell interprets '$$' as its own PID.

```
> less /proc/$$/maps
08048000-080c0000 r-xp 00000000 03:02 115456 /bin/bash
080c0000-080c6000 rw-p 00077000 03:02 115456 /bin/bash
080c6000-0811e000 rwxp 00000000 00:00 0
40000000-40016000 r-xp 00000000 03:02 82360  /lib/ld-2.2.2.so
40016000-40017000 rw-p 00015000 03:02 82360  /lib/ld-2.2.2.so
...
4019e000-401b1000 r-xp 00000000 03:02 82378  /lib/libnsl-2.2.2.so
401b1000-401b3000 rw-p 00012000 03:02 82378  /lib/libnsl-2.2.2.so
401b3000-401b5000 rw-p 00000000 00:00 0
bfffb000-c0000000 rwxp ffffc000 00:00 0
```

Text segment from `0x0804 8000` to `0x080c0 0000`, marked read-only.
Data segment next (initialised from `/bin/bash`, but writable).
Then bss and heap together, not associated with any file and writable.
A large gap, and then the first shared library at `0x4000 0000`: just its text and data segments are shown.
After many more shared libraries, the last here having text, data and bss segments, a gap followed by the stack growing down from `0xc000 0000`.

In Solaris similar information can be obtained using '`pmap $$`'. As a TCM-special, Tru64 has a `pmap` command giving somewhat more limited information.

Stack for functions

The stack is a simple storage area which (traditionally) grows downwards in memory, and which has a register, the stack pointer, dedicated to marking the lowest address in it in use. Changing the size of the stack is thus trivial: one changes the value of the stack pointer.

When a function is called, the return address (the next instruction in the calling function) is placed on the stack, and the stack pointer adjusted appropriately, and then a jump made to the address of the function. To return, the function reads the return address from the stack, and again the stack pointer is adjusted.

The portion of the stack used by a function is called its *stack frame*. The stack is a sequence of these frames which reflects the calling sequence which the code followed to reach the current function.

Tradition (and DOS!) has a downwards growing stack, an upwards growing heap, and they can keep growing until they collide. Linux tends to have shared libraries in the middle causing further confusion.

Stack for variables

The stack is also a convenient place to store small local variables. (Local: destroyed as soon as the function in which they are defined exits.)

Variables whose values are saved between function calls (Fortran: `save`, C: `static`) cannot be placed on the stack in this manner, and are usually placed in the uninitialised data segment.

The stack is also used as temporary scratch space for saving intermediate values of long expressions, or saving register values.

The IA32 architecture has relatively few registers: eight floating-point, and eight general purpose. This means that it is much more likely to run out of registers for holding intermediate results and be required to resort to the stack than a typical RISC processor with 32 floating point and 32 integer registers.

Arguments on the stack

Finally the stack is the placed used for transferring arguments to functions, and returning results from them. If the number of arguments is small, they may be passed in the CPUs registers, and similarly for return values.

Tru64 is capable of passing up to six integer or floating-point numbers to a function using just registers: the seventh argument will go onto the stack. IA32 Linux passes everything via the stack, and can return just one integer or one floating-point value in registers.

Tru64 is marginally better on returning values, in that it can return a complex number (two floating-point values) in registers. It also passes the return address in a register, thus avoiding any use of the stack at all for trivial functions.

Fortran traditionally passes everything with pointers, not by value. Hence for `sin(0.7)` C will pass a double, Fortran a pointer to a real.

The stack frame

Address	Contents	Frame Owner
	. . .	calling
	2nd argument	function
%ebp+8	1st argument	
%ebp+4	return address	
%ebp	previous %ebp	
	local	current
	variables	function
	etc.	
%esp	end of stack	

The above diagram is based on IA32 Linux.

When the current function exits, it will set the `%esp` register to the current value of the `%ebp` register, and restore the old value of `%ebp`. Being IA32, and thus highly CISC, it will do this using a single, one-byte instruction.

Naturally the local variables of an exited function are lost.

The Alpha uses register $30 as the stack pointer, $26 for the return address, and $15 for the frame pointer only if the frame is of variable size.

Big stack problems

```
a=matmul(b,matmul(c,d))
```

Is `matmul(c,d)` a suitable object for placing on the stack, or is it too big?

```
allocate a(n,n)
...
call wibble(a(1:m,1:m))
```

Where will `a(1:m,1:m)` be constructed?

In the world of F77 and C, it is unusual for any large object to end up on the stack, and historically limits on stack sizes have been low. In the world of F90 and C++, this is not so.

Process limits

Limits, both hard (absolute) and soft (raisable) are placed on a process's use of various parts of memory:

```
% limit
datasize        unlimited
stacksize       8192 kbytes
coredumpsize    1000000 kbytes
memoryuse       unlimited
$ ulimit -a
core file size (blocks)     1000000
data seg size (kbytes)      unlimited
max memory size (kbytes)    unlimited
stack size (kbytes)         8192
virtual memory (kbytes)     unlimited
```

The first form is `csh/tcsh`, the second `bash`.

`memoryuse` or `max memory size` is the maximum physical memory a process may use, whereas the virtual memory controls how many pages of virtual memory it may use.

Core files

A `core` file is produced in response to certain forms of signal, if not prevented by the process's resource limits.

It contains a copy of the values in all the registers at the time the core was created, the signal number which caused the core to be created, the name of the executable being run, and a complete dump of the virtual address space.

This complete dump does not contain the text segment: as that is read-only, it can be obtained perfectly from the original executable file. Nor does it contain the read-only segments of shared libraries, for similar reasons.

Core files can be prevented with
`ulimit -c 0` (Bourne-like shells)
`limit coredumpsize 0` (C-like shells)

Debuggers

A debugger will work either with a core file together with the original executable, or by examining the address space of a running process directly. The information it gives about the function call tree, and function arguments, is obtained by walking back through the stack frame from the current value of `%ebp`.

If `%ebp` does not point to the start of the current stack frame (e.g. the code was compiled with `gcc` and the `-fomit-frame-pointer` option), this is impossible.

If the cause of the crash was that a variable stored on the stack overflowed (array index too large, C string too long) and overwrote part of the stack, a debugger will display nonsense. This is a great weakness of this stack model: a local variable overflowing can destroy the return address of that function, causing a crash as soon as the function exits, and destroy the stack frame sufficiently for a debugger to have little idea how the code got to where it did.

A 64 bit world

Arguments to functions are usually just pushed onto a stack. The function knows what type of argument to expect, so it can read them off sensibly: there is nothing to tell it where one argument stops and another begins. Similarly when a caller reads the return value of a function.

With IA32 Linux, most arguments relating to pointers or the size of objects are 32 bits: the return from `malloc()`, the argument to `fseek()` (move to a certain offset in a file), the file length field returned from `fstat()`, `ftell()` (the current offset in a file) etc.

This is unfortunate. With 32 bits one can address no more than 4GB, as there are only (approx) 4×10^9 combinations for 32 binary values. The limit usually bites at 2GB, for functions such as `fseek()` need to be able to move forward and backwards from the current point, so take signed values.

Tru64

Tru64 UNIX has always been 64 bit. All the above system calls have always returned 64 bit values, and thus there is no problem dealing with object bigger than 2GB.

Moving IA32 Linux towards 64 bits is hard. There is no reason why some of the file I/O commands should not support 64 bits, even if there is no hope of 64 bit memory addressing as the CPU does not support it. However, old applications will expect `ftell()` to return a 32 bit value, and `fseek()` to require one. Adding an `fseek64()` and `ftell64()` is easy, but one needs to change one's source to gain any advantage.

The current solution is to have both sets of functions, with `gcc` compiler magically changing to the 64 bit versions if the preprocessor symbol `_FILE_OFFSET_BITS` is defined and set equal to 64.

Life under Irix is much messier, as Irix was wholly 32 bit, and is now wholly 64 bit. So there are two versions of every library, one expecting 32 bit pointers, and one 64 bit pointers. Fortunately the linker prevents any accidental mix-and-match approaches. AIX and Solaris have similar problems.

What should a 32 bit version of `fstat()` do if called on a file which is larger than 4GB?

Not 64 bits

Floating point data has been 64 bit for double precision and 32 bit for single precision on all major architectures for about two decades. The IBM PC has supported 64 bit double-precision floating-point arithmetic since it first gained the 8087 maths co-processor in the mid 1980s, even though the main processor was mostly 16 bit.

In the past such terms have been used to describe not the address space of a computer, but its data bus width. The ZX Spectrum and BBC model B were both called ‘8 bit computers’, yet both could address 64KB of memory (i.e. 16 bits worth) not 256 bytes (8 bits worth). However, both had 8 bit data buses.

All IA32 processors since the Pentium have had 64 bit buses, but no-one has bothered to call them 64 bit CPUs.

Graphics cards are even more confusing in their use of the term ‘bits’. Sometimes it refers to the width of their data buses (S3 Trio64, ATI Radeon 128), and sometimes to the number of colours which can be displayed simultaneously: 256 for 8 bit cards, 16 million for 24 or 32 bit cards.

The Boot Sequence

POST

When first turned on, a computer begins a Power On Self Test. Some of this progresses at the hardware level: until the PSU has managed to achieve a stable voltage, power will simply not be applied to the CPU or to other important parts of the computer.

The majority of the POST is performed by the processor, which, at this point, is running a program from a ROM on the motherboard.

PCs typically look for some form of graphics card very early in their POST, and then give a running commentary. If they are unable to find a graphics card, or are facing some other major problem (no RAM, or even no CPU), they will make plaintive beeping noises, with the number and duration of the beeps giving a clue as to the problem, provided one has the correct handbook to decode them (every motherboard is different).

Real computers do not mind if there is no graphics card, and communicate instead via their serial port.

Fast or Slow?

Much of the POST is optional. It is necessary for a computer to detect how much memory it has, but not necessary for it to test every byte each time it is turned on. Thus many computers offer a choice of a ‘fast’ or ‘slow’ boot, depending on whether extensive memory tests are done.

However, whilst the POST progresses, one has the option of pressing some key (usually {F1} or {Del}) to enter the BIOS setup screens for changing boot devices etc. If the POST is too fast, it is quite hard to get at these screens. . .

Options, options

A PC will also scan its expansion cards for ROMs at this point, to see if any peripherals have code that they want executed at boot time. Usually the video card will be eager to display its manufacturer's logo, and the SCSI card will want to explain to the computer's BIOS that it can provide a valid device from which the system can be booted. (A standard PC BIOS knows nothing about SCSI, and can cope with IDE disks only.)

It will also go looking for interesting peripherals, such as floppy disk drives, hard drives and CD ROMs. If it finds something it doesn't recognise, it will cheerfully ignore it at this point.

BIOS: Basic Input Output System, something which provides very basic low-level access to disk drives etc.

The Boot Device

Having worked out which bootable devices are connected, the PC will then try to boot from them, in the user-specified order which can be changed in the BIOS setup screens. The original order was first floppy drive, first hard disk drive, and then fail. Today CD ROMs, Network cards and USB devices are often options.

For the drives, the BIOS understands very little about how to operate them. It can check to see if the drive exists, has media (if a floppy), whether the media is bootable, and, if so, it loads the ‘boot sector’ into memory and executes it.

The boot sector is tiny – just 512 bytes including the partition table on a PC – and will usually immediately load a rather larger boot loader.

The Boot Loader

Common boot loaders in the Linux world are LILO and Grub. Windows uses its own, and there are many others. These can be quite large programs, offering the user a choice of what to do next, and interacting happily with the user. The larger boot loaders may have a reasonable knowledge of how to read a filesystem (Grub does), or they may prefer to read a fixed sequence of sectors from a disk and hope.

Either way, they need to get an OS kernel into memory. To keep Linux happy, the minimum requirement is to load a compressed kernel from disk, decompress it, and also pass to it some arguments, telling it where it should find the rest of the OS.

Which boot loader are you running?

```
# dd if=/dev/hda bs=512 count=1 | strings -4
```

may well reveal the string ‘GRUB’ or ‘LILO’.

The Kernel

The kernel now needs to take care of loading the rest of the OS. But note that the BIOS loads the bootloader, and the bootloader the kernel. It is quite possible for a kernel to be unable to detect the disk it was loaded from: afterall, it didn't load itself, it was loaded by someone else. This is awkward if one believes in *modular* kernels which have most hardware drivers as separate files, rather than integrated into a single, massive *monolythic* kernel. One might have the correct driver for the kernel to be able to access the disk, but it's on that disk which cannot be accessed until that driver is loaded…

The Linux solution involves an initrd: an initial RAM disk. The boot loader loads a compressed image of a (tiny) disk, decompresses that too into memory, and the kernel is able to access this RAM disk to find any drivers it needs before it accesses the real disk. If the real disk is 'standard', the initrd may be unnecessary.

Windows has similar issues: I have seen a Windows installation disk complain that it could not find the CD drive it had just booted from.

The (Linux) computer I am typing this on does not use an initrd.

Hardware detection again

Just as the original ROM code needed to do hardware detection to work out what it could do next, now the kernel needs to perform the same trick, as the bootloader may have loaded it, but it will not have told it much. Processor type, amount of memory, what to use as a console, what PCI buses and cards exist, what disk drives and network cards exist, and how to assign IRQs are all very interesting questions that the kernel needs to resolve.

However, it does not need to score 100% in this test: if it misses some items because they are “unknown”, it is not a problem, provided that the CPU, memory and the disk drive to boot from are all detected.

The next stage

The kernel will understand the filesystem perfectly, and will mount its root filesystem, read-only, as `/`. It will also set up all the data structures it needs to act as an OS kernel. This done, it starts one process, `/sbin/init`, and its work (for booting the machine) is over.

The kernel will, of course, be heavily used by every process until the machine is shutdown. It is responsible for creating and destroying processes, afterall, and all disk and network access.

`init`

The `init` process, after setting itself up, runs the script `/etc/rc.sysinit`. This is just a simple shell script being run by with the privileges of root.

It configures the keyboard, checks that the filesystems were unmounted cleanly, runs `fsck` if not, remounts `/` read-write, starts software RAID devices, adds swap space, starts loggers (note nothing can be logged to the disks until they are mounted read-write: earlier messages are logged to a buffer in memory and then flushed to disk when possible later) and loads drivers for the more unusual bits of hardware.

This is just one big, ugly script, and changing it is awkward. So the parts that one might want to change are elsewhere.

Run-levels

A UNIX system can run at different `run-levels`. These vary between vendors, but usually include *single user mode* (designed to be used by root only), *multi user mode*, the same with X, reboot and halt.

The `init` process is responsible for booting into the default (or the specified) run-level, and for changing the run-level of a running system. Hence a reboot is just a change of run-level under the control of `init`.

The default run-level is specified in the file `/etc/inittab`, and is usually multi-user with X. For some servers it would be without X. The main use of single user mode is, like Windows' "safe" mode, for recovering from surprises: it tries to do as little as possible whilst booting, so it is more likely to succeed.

Tru64's single user mode leaves `/` mounted read-only, and `/usr` not mounted at all.

/etc/init.d

The /etc/init.d directory contains scripts which are always called with a single parameter, either start or stop, and which each start (and stop) a service or a set of services.

Hence there is a script for starting the network, a script for starting the print spooler, one for sshd, one for a mailserver (configured off, *please*), and so on.

The order of calling these scripts matters, and the calling sequence is determined as follows by the runlevel required as follows.

Tru64 uses /sbin/init.d, some UNIX flavours use /etc/rc.d/init.d.

rc?.d

Each run-level is associated with a directory called `/etc/rc.d/rcN.d` for runlevel N. This directory is full of scripts, named so that the first character is either ‘K’ or ‘S’, the next two characters are digits, and the rest is arbitrary. Tradition says that the final part of the name is the name of the corresponding script in the `/etc/init.d` directory, and these scripts are simply symbolic links to the scripts in `/etc/init.d`.

On entering a new runlevel, first the scripts starting with a ‘K’ are run sequentially, in numerical order, with the single parameter ‘`stop`’. The the scripts starting with an ‘S’ are run in order with the parameter ‘`start`’. (Immediately after a reboot the kill scripts are usually omitted.)

Tru64 uses `/sbin/rcN.d`, some UNIX flavours use `/etc/rcN.d`.

Simplicity

```
for script in /etc/rc.d/rc${runlevel}.d/K*
do
  $script stop
done

for script in /etc/rc.d/rc${runlevel}.d/S*
do
  $script start
done
```

The above really is pretty much the code used to change to a new runlevel. For the runlevels corresponding to halt and reboot, there will be no start scripts.

No standards

Runlevel	Effect
0	Halt (Irix: halt and power off)
1 or s	Single user
2	Linux/Irix: multiuser, no NFS; Tru64/Solaris: multiuser, no networking
3	Multiuser
4	
5	Linux: Multiuser with X11; Solaris: halt and power off; Irix: halt
6	Reboot

No X

Tradition dictates that X itself is not started by an init script, but rather by a direct entry in `/etc/inittab`. This is partly because entries in `/etc/inittab` can specify what action should be taken when the script or service started exits. In most cases the answer is nothing, in the case of `xdm`, the thing responsible for starting X, the answer is to restart it. Thus if/when `xdm` crashes or is killed, `init` will restart it.

Sensible inits (which includes current Linux distributions) notice how frequently they are having to restart a service, and, once a threshold is reached, they give up for a short while (5-10 minutes).

Basic Configuration

The order for mounting filesystems is, as mentioned, `/` read-only, and then, much later, after running `fsck` as necessary, `/` read-write, and other filesystems for the first time (usually read-write immediately).

Two facts should be obvious. Firstly, until `/` is mounted read-write, nothing can be written to it, including logs. If the boot fails before this point, nothing will be logged to disk.

Secondly, everything which needs doing before mounting and in order to mount the other filesystems must be doable from the data in `/`.

Filesystem layout

The directories `/bin`, `/sbin`, `/etc` and `/lib` are always on the same device as `/`. The directory tree starting at `/usr` need not be (and, by default, is not with Tru64). Thus the basic system utilities must work in the absence of `/usr`.

Specifically, kernel modules are found under `/lib`, not `/usr/lib`, the basic shells and the shared libraries they require (if any) in `/bin` and `/lib`, and most configuration files are in `/etc`.

Non-critical things, such as X11, compilers, numerical libraries, and most convenience applications live under `/usr`.

In English, ‘`bin`’ and ‘`lib`’ are pronounced as spelt. In American they are pronounced as the first syllable of ‘binary’ and ‘library’ respectively.

Configuration files in `/etc`

fstab
List of filesystems to mount at boot (`filesystems` on AIX, `vfstab` on Solaris).

hostname
The name of this machine (`nodename` on Solaris).

hosts
Hostname to IP address lookup without a DNS.

passwd
Password file.

printcap
Printer configuration (`printers.conf` on Solaris).

syslog.conf
System logger configuration file.

Daemons

A *daemon* is the name given to a program which detaches itself from all terminals, loses its parent (so its parent is `init`), and sits in the background doing some function. Often daemons listen for connections from the network: WWW servers, mail servers, `sshd` and many others.

Some daemons perform more local tasks, such as logging (`syslogd`), running periodic tasks (`crond`), or tasks at specific times (`atd`).

Having one daemon for every network service would be excessive, so there is a daemon called `inetd` (configured by `inetd.conf`) which is responsible for listening on many ports, and launching the correct program to service requests as and when required. Servers with a high start-up cost (such as WWW and email) are not usually run this way, but `rshd` and `rlogind` are almost always run thus.

Daemons produce the final ‘d’ in the names of many processes.

Logging

Everyone who runs a UNIX system connected to the internet needs to be familiar with `syslogd` and its logs. For only by reading logs will one see attempted hacks, and warnings of hardware failure or misconfiguration.

The program `syslogd` starts early in the boot sequence, reads its configuration from `/etc/syslogd.conf`, and logs messages to (usually) either `/var/log/messages` or `/var/adm/messages`. This log file will grow linearly, so something needs to save old copies and delete very old copies occasionally: a procedure called *log rotation*.

Silly computers default to throwing away most of their logs: someone has just made a thousand unsuccessful login attempts from Korea? Surely no-one wants to know this? A memory chip has needed its data corrected a dozen times in the last ten minutes? That's life.

Such computers need shooting.

Logging in

The program which responds to an attempt to login (from the network or the keyboard), must do the following things:

Satisfy itself that you are who you say you are. It will interpret password file enties itself if it expects a password.

Launch a new process for you.

That process must then change its group memberships to yours, then its uid to yours. It must previously have been running as root in order to be able to change its groups and uid.

Finally, `exec()` your prefered shell as a login shell (or launch an X session, or whatever).

Silly: each application (`rlogind`, `sshd`, `ftpd`, `xdm`, etc.) interprets the password file on its own, so all had better agree on its precise format.

(Many other subtleties exist, such as making a new process group, and worrying over `tty` permissions.)