UNIX: System administration

A Concise Guide

By Rudolf Cardinal

Revision dated 18th August 1995

INTRODUCTION 7

WHAT IS UNIX? 7

BASICS OF THE UNIX COMMAND LINE 7

Finding files: `find` 7

Examining a directory: `ls` 7

Deleting, copying and renaming files 7

Creating and deleting directories 8

Moving around directories 8

Viewing and editing files 8

Pattern searching: `grep` 8

Editing files: a brief guide to `vi` 10
Cursor movement 10
Altering the file: basics 11
Rearranging and duplicating text 11
Miscellaneous 12
Commands preceded by a colon (`ed` commands) 12

BOOTING A UNIX MACHINE 13

STOPPING A UNIX MACHINE 13

FLUSHING THE CACHE 13

BROADCASTING MESSAGES TO PEOPLE 13

WHICH MACHINE AM I ON? 13

HOW UNIX STARTS 14

UNIX SECURITY; USERS, GROUPS AND OWNERSHIP 14

The superuser, root 14

The `su` command 14

THE UNIX FILE SYSTEM 15

Filenames and Wildcards 15

File ownership; output of `ls -al` 15

Changing the mode (flags) of a file 15

Changing the ownership of a file 16

Drives – concept, mounting and dismounting 17

`/etc/fstab` 18

What's CHKDSK in UNIX? 18

Links 18

NFS – beware 19

A little look at system files and directories 20

MANAGING USER ACCOUNTS 21

Adding users 21

Deleting users 21

Adding groups 21

Removing groups 21

Changing passwords 21

The `/etc/passwd` file 21

The `/etc/group` file 22

Getting information 22

Advanced security 22

Login banners 23

Message of the day 23

Trusted path 23

Disabling login 23

MANAGING PROCESSES. SNOOPING AND KILLING ERRANT TASKS. 24

First, some theory. 24
- What is a process? 24
- Signals 24
- Pipes 24
- Sockets 24
- Forking 25

Swapping out 25

Finding out about processes 26
`w` – what are people doing? 26
`ps` – process status 26

Sending signals to processes; how to kill processes 28

Be nice 28

Monitor thyself for evil 29

Other status commands 29

DEVICES 30

Making devices 30

Null 30

Memory 30

Disks 31

Tapes 31

Terminals 31

LAT configuration 33

Printers 34

DAEMONS 37

Concept 37

A brief summary of common daemons 37

CRON: SCHEDULING PROCESSES, SUCH AS BACKUPS 38

Format of `/etc/crontab` 38

The `at` and `batch` commands 38

PRINTING 40

The `lpr` command – print files 40

The `lprm` command – remove jobs from printer queue 40

The `lpq` command – examine spool queue 40

The `lpstat` command – printer status information 40

The `lpc` command – line printer control **41**

USING TAPE DRIVES 42

Magnetic tape manipulation: `mt` **42**

Backing up data: `dump` **42**

Restoring data: `restore` **43**

Archive manipulation: `dd`, `cpio`, `tar` **45**
`dd` 45
`cpio` 45
`tar` 46

NETWORKING 48

Introduction **48**
TCP/IP: addressing 48
LANs and beyond: address resolution, routing and complex services 49
Internet addresses for humans 49

Configuring UNIX **51**
The simple way: using `netsetup` 51
Essential files 51
Interface configuration: `ifconfig` 52
The Internet daemon, `inetd` 53
Routers 54
NFS – the Network File System 55
How a typical network starts 61
Remote booting – the bootp protocol 62
MOP file retrieval – `mop_mom` 62

Some important client programs for users and administrators **63**
`ftp` (requires `ftpd`) 63
`ping` (administrative) 66
`telnet` (requires `telnetd`) 66
`finger` (requires `fingerd`) 66
`rlogin` (requires `rlogind`) 66
`rsh` (requires `rshd`) 66
`netstat` (administrative) 67
`ruptime` (administrative) (requires `rwhod`) 67

REBUILDING THE KERNEL 68

Editing the configuration file **68**

Generating the kernel and activating it **71**

SOFTWARE SUBSETS 72

SHELLS AND SHELL SCRIPTS 73

What is a shell? 73

Simple and background commands 73

Standard input, output, error. Redirection and pipes. 73

Paths and environment variables 74

Shell scripts 75

The sh command language in brief 76

Invoking shells, login scripts and restricted shells 78

Two lines about csh 79

ACCOUNTING 80

Login accounting 80

Command usage accounting 80

Printer accounting 81

ERROR LOGS 82

MAIL 84

OTHER HANDY COMMANDS: THINGS LEFT OVER 85

GETTING HELP: WHERE TO GO FROM HERE 87

The online manual, man 87

About the text manuals 87
The ULTRIX manuals and their abbreviations 87

Introduction

This is a guide for system administrators. It assumes reasonable familiarity with syntactic definitions and command-line operating systems in general, and some skill with the basics of UNIX (cataloguing disks, editing files and so forth). It also assumes you have full authority over your system. I don't usually mention when superuser authority is required for a particular command: in general, anything that affects other users, their processes or their data requires `root` authority.

I have based this guide on ULTRIX from Digital; this is a BSD UNIX clone.

This is primarily a reference guide, to look things up in and not to read from cover to cover.

What is UNIX?

UNIX is a multiuser operating system. It is organised into a kernel, the main "program" that is the operating system, and a set of utility programs found on disk. It provides facilities for many users to run programs simultaneously, and to keep files on the system, with no impact on each other aside from the system's apparent speed. In order to administer UNIX there is a superuser, "`root`", with complete authority over all aspects of the system. That's you, that is.

Basics of the UNIX command line

I don't give full details of these commands, just the most useful options. See *Getting Help* for details of the manuals.

Finding files: `find`

Simplified syntax:

`find` ***startdirectory*** **`-name`** ***filename*** **`-print`**

Without the `-print` command, you don't see the result.
Examples:

```
find / -name rc.local -print
find /usr -name '*.c' -print
```

Examining a directory: `ls`

`ls`	Basic catalogue
`ls -al`	Full details
`ls` ***filespec***	About a given file/group of files.
`ls -al | grep '^d'`	List all directories.

Deleting, copying and renaming files

`rm` ***filename***	Deletes.
`cp` ***source dest***	Copies. The `-r` option allows recursive copying.
`mv` ***source dest***	Moves or renames. See `mv`(1).

These can take the parameters – ("everything that follows is a filename", so you can use filenames starting with `-`); `-f` (force); `-i` (interactive mode) and `-r` (recurse subdirectories). The `mv` command cannot take `-r`. The `cp` command can also take `-p` (preserve file dates/times/modes). Possibly the worst thing you can do to UNIX is to issue the command "`rm -r *`" from the root directory while you are superuser.

Creating and deleting directories

mkdir ***directory***	Makes a directory
rmdir ***directory***	Removes a directory

Moving around directories

cd [***directory*** **]**	Change to *directory*. If no directory is specified, the directory specified in the environment variable `$HOME` (the user's home directory) is used instead.
pwd	Print working directory.

Viewing and editing files

cat ***filename***	Same as `type` in DOS.
more ***filename***	Same as `more` in DOS; equivalent to `cat` *filename* `|` `more`, which also works. Also equivalent to `more <` *filename*. Space for next page, `q` to quit. If you use more as "`more` *filename*", you can also press `b` to move back a page; the other forms of the command use piped input and `b` doesn't work.
head ***filename***	Looks at the top of a file. Can use as in `head –30` *filename*, to look at the top 30 lines.
tail ***filename***	Looks at the end of a file. Can specify line count as for head.

Pattern searching: `grep`

`grep` stands for "get regular expression". `grep` can be used, like `more`, as a filter (`command | grep` *options*), a place to route input (`grep` *options* `< file`) or as a straight command (`grep` *options file*).

Syntax:

`grep [` ***options*** **]** ***expression*** **[** ***file*** **]**

When specifying a pattern ("expression") to match, there are many special characters and wildcards: see `grep`(1) for details of these and all the other options. The following patterns are the most useful:

Pattern	Matches
`^`	beginning of a line
`$`	end of a line
`.`	any character
normal character	that character
`[`*string*`]`	any character from *string*; you can use ranges as in `[a-z0-9]`
`*`	zero or more characters

Examples:

`ls -al | grep '^d'`	Catalogues all directories in the current directory by searching for lines in the output of "`ls -al`" that begin with a `d`.
`grep fish tree.c`	Looks for the word "`fish`" in the file `tree.c`.
`grep execute *.h`	Looks for "`execute`" in all files ending "`.h`". If more than one file satisfies this criterion, its name will be shown in the output from `grep` so you know where to look.
`grep fish < tree.c`	Same as "`grep fish tree.c`"
`ps -aux | grep '^oracle' | more`	Gives process status information on all processes owned by `oracle`, pausing between pages.

For more complex pattern-matching, use `egrep`(1) or `fgrep`(1).

Editing files: a brief guide to `vi`

Fire it up with **`vi`** ***filename***.
For read-only access, use **`view`** ***filename***.

Pronounced "vee-eye", `vi` exists in two states: edit mode and command mode. You begin in command mode. At any time, you can return to command mode by pressing Escape. (If you're on a VT terminal and you get a back quote, you can either go into the terminal's keyboard setup and make that key send ESC, or you can use Ctrl-`[` instead of Escape. Escape and Ctrl-`[` both send character number 27, which is what you're after.) If you were in command mode anyway, it beeps at you.

Now for some commands – note that these are all case-sensitive! By the way, `^X` is a convention for Ctrl-X.

Cursor movement

SPACE	Advance the cursor one position
^B	**Move backward a page. (A count specifies repetition.)**
^D	Scrolls down. (A count specifies number of lines and is remembered for future ^D and ^U commands. Default is a half-page.)
^E	Exposes another line at the bottom
^F	**Move forward a page. (A count specifies repetition.)**
^H	Backspace the cursor
^J	Move cursor down (same as down arrow)
^M	A carriage return advances to the next line at the first non-white character. Given a count, it advances that many lines (as in 5^M). During an insert, it causes the insert to continue onto another line.
^N	Next line, same column (same as down arrow)
^P	Previous line, same column (same as up arrow)
^U	Scrolls up (see ^D)
^Y	Exposes another line at the top
0	**Go to start of line**
^	First non-white on line
$	**End of line**
%	Finds matching bracket, brace or box. Useful for programming!
)	Forward sentence
}	Forward paragraph
]]	Forward section
(	Backward sentence
{	Backward paragraph
[[	Backward section
+	Next line, at the beginning
–	Previous line, at the beginning
/	**Scan for a string (that follows the /), forwards**
?	Scan backwards
B	Back a word, ignoring punctuation
H	Home screen line
M	Middle screen line
<line>G	**Go to a particular line. So 1G goes to the top of the document; 100G goes to line 100. If you type G on its own, you go to the end of the file.**
L	Last screen line
W	Forward a word, ignoring punctuation
b	Back a word
e	End of current word
h	Move left

j	Move up
k	Move down. If you're lucky (!) the arrow keys will also work.
l	Move right
n	**Scan for next instance of the pattern specified with / or ?**
w	Word after this word

Altering the file: basics

i	**Insert text before the cursor**
I	Insert text at start of line
a	**Append text after cursor**
A	Append text at end of line
d	**Deletes the object you specify. Examples:** **d0 Delete to start of line** **d$ Delete to end of line** **dd Delete line**
^@	Not a command character. If typed as the first character of an insertion it is replaced with the last text inserted (up to 128 characters), and the insert terminates. A ^@ (ASCII 0) cannot be part of the file.
^I	Inserts a tab, during insert.
^Q	Not a command character. In input mode, ^Q quotes the next character (same as ^V), except some terminal drivers 'eat' ^Q so the editor never sees it.
^T	Not a command character. During an insert, with autoindent set and at the beginning of a line, inserts shiftwidth white space.[1]
^V	Not a command character. In input mode, quotes the next character so that it is possible to insert non-printing and special characters into the file.
^W	Erase a word during an insert (deleted characters remain on the display).
^Z	If supported by the Unix system, stops the editor.
^[(esc)	**Cancels partially formed commands; terminates input on the last line; ends insertions; if editor was already in command mode, rings the bell.**
erase	(Usually ^H or #) Erases a character during an insert
kill	(Usually @, ^X or ^U) Kills the insert on this line
O	Opens and inputs new lines, above the current
o	Opens and inputs new lines, below the current
U	Undoes the changes you made to the current line
u	Undoes the last change
c	Changes the object you specify to the following text

Rearranging and duplicating text

f*x*	Find *x* forward in line
p	Put text back, after cursor or below current line
y	Yank operator, for copies and moves
t*x*	Up to *x* forward, for operators
F*x*	f backward in line
P	Put text back, before cursor or above current line
T*x*	t backward in line

[1] vi looks at an environment variable called EXINIT for these options (though you can also type :set *option* while in vi). For example, to have vi move the cursor to a bracket's pair for a second when you type a bracket, you can type ":set showmatch" in vi, or "EXINIT='set showmatch'; export EXINIT" in your .profile (executed when you log in).

Miscellaneous

^G	File statistics, including how many lines there are and what line you're at.
^L	Clears and redraws screen.
^R	Redraws the screen, eliminating logical lines not corresponding to physical lines (lines with only an @ on them).
.	Repeats the last command that changed the buffer (the text)
ZZ	**Saves and exits.**

Commands preceded by a colon (ed commands)

The other sort of commands in vi are those preceded by a colon (:). When you type a colon, the cursor hops to the bottom line where you can type in commands. Press enter to execute the command.

:w	**Write the file**
:q	**Quit. If the file has changed, it won't let you.**
:q!	**Quit, even if the file has changed (losing the changes).**
:wq	Writes, then quits.
:x	Write (if necessary), then quit (same as ZZ).
:e *file*	Edit file *file* (losing changes)
:e!	Reedit, discarding changes
:e + *file*	Edit, starting at end
:e + *n*	Edit, starting at line *n*
:w *name*	Write file *name*
:w! *name*	Overwrite file *name*
:*x*,*y*w *file*	Writes lines *x* through *y* to *file*
:r *name*	Read file *name* into buffer
:r !*cmd*	Read output of *cmd* into buffer
:n	Edit next file in argument list
:n!	Edit next file, discarding changes
:n *args*	Specify new argument list
:ta *tag*	Edit file containing tag *tag*, at *tag*
:shell	Fires up a shell. Press ^D to return to vi.

Booting a UNIX machine

This depends for the most part upon the hardware. Turn on your UNIX box and after testing itself it will give you a console prompt (>> on the DEC machines).

The machine knows about the devices (SCSI, Ethernet, etc.) attached to it, because it just asked them what they were. If you want to know too, you have to ask the machine: on the DECs this is a command such as `conf` (DECsystem 5100) or `test -c` (DECstation 3100). If in doubt, type `?`.

The objective is to load a file called `vmunix` from one of those devices. Normally, the machine will have been set up with an environment variable in its CMOS RAM. So… **try `boot` or `auto`**.

Failing that, you have to specify which device to boot from, as you do when you install UNIX. This command varies between machines, but you must specify SCSI controller number, SCSI device ID and device type. On a DECstation 3100, to boot from SCSI tape (*tz*) with SCSI ID #5 on controller #0, you type `boot -f tz(0,5)`. Check with the machine manual, or better, the UNIX installation guide.

Stopping a UNIX machine

There are many ways to do this:

`shutdown -h now`	Shuts down now, halts and returns to the machine's console prompt.
`shutdown -r now`	Shuts down now and reboots.
`reboot`	Same as the above.

You can also use `shutdown` to specify a time for shutdown, but this is of limited use.

Flushing the cache

If you ever have to turn off a system running UNIX, it is essential to flush the cache.
UNIX uses write-behind caching: you might think you wrote a file to disk, but chances are it's still in RAM. Until we get laser-addressed non-volatile protein memory – you heard it here first, courtesy of *Scientific American* – this means switching off the power makes a mess of UNIX.

Type `sync`. Wait.
When the prompt returns, assuming no other processes (i.e. users) write to disk, you can switch off in relative safety.

Broadcasting messages to people

If you want to shut down and there are people on the system, it is courteous to tell them.

`wall`	Short for "write all". You need to be root (superuser). Type in your message, press `^D` to finish (or `^C` to abort).
`write` ***user***	The same, but to a particular user.
`cat > /dev/tty`*xx*	The rude and amusing way to do it. If you know what terminal they're logged on at (see *Managing Processes and Snooping* below), and you are superuser, you can write directly to their terminal. That way you don't get the beep and the message saying "Message from …". It can really confuse people `:-)`

Which machine am I on?

Type `hostname`. If it's not what you expect, don't shut it down!

How UNIX starts

1. `/vmunix` loads. The operating system itself. This runs…
2. `/bin/init`. If the reboot 'fails' or multi-user mode is not set up, `init` leaves the system in single-user mode (talking to the console, with superuser privileges). If the reboot succeeds (or when the superuser presses `^D` in single-user mode), init begins multiuser operation and runs…
3. The shell script `/etc/rc` executes. This brings up the file systems (running `fsck`, file-system check, to ensure their integrity) and performs other somewhat essential tasks (bringing up system daemons). As part of its execution, it runs…
4. The shell script `/etc/rc.local` executes. This contains machine-dependent stuff, like the machine's name: "commands pertinent only to a specific site", according to the manual. The "`.local`" suffix indicates the idea behind this: `rc` contains stuff that any UNIX system will need; `rc.local` contains stuff specific to this system.

UNIX security; users, groups and ownership

Everything in UNIX, be it memory, a tape drive or a directory, is owned. This is to prevent processes from reading from or writing to things they shouldn't. We needn't be overly concerned with process ownership here. File ownership is covered under *File systems*.

The superuser, root

This user has read rights to everything and the ability to change the owner and mode (flags) of anything.

The su command

`su` stands for "**s**ubstitute **u**ser ID". The syntax is

`su [ – | -f ] [` ***username*** **`]`**

If *username* is omitted, '`root`' is assumed, and after a correct password is entered a # prompt is substituted for the `$` "to remind the superuser of his responsibilities". If you were superuser to begin with, you need enter no password.

Normally, `su` changes no part of the user environment except the variables `HOME` and `SHELL`. If you use "`su –` *username*", a full login is simulated (so all environment varibles will be set). The `–f` parameter prevents `csh` (the C shell) from executing `.cshrc`, making `su` start faster. It isn't relevant in a system using `sh`.

Since the shell (see *Shells and Shell Scripts*) can take the parameter `-c` *filename* to read commands from *filename*, you can issue the command

`su –` *user* `-c` *shellscript*

to run *shellscript* as *user*. This form of the command is very useful in scripts run automatically by `cron` (see *Cron*).

The UNIX file system

Filenames and Wildcards

- UNIX filenames are longer than you need them to be. They cannot contain some characters (`?`, `<`, `>`, `$`, that sort of thing). They can contain dots (`.`) *but* they don't have file types or extensions like DOS. They can contain more than one dot, too (e.g. `tree.c.backup`).
- A file whose name *begins* with a dot is invisible on a normal `ls` display (`ls -a` shows all files).
- A `?` is a wildcard for any single character. `*` is a wildcard for any (null or greater) group of characters.

File ownership; output of `ls -al`

All files are owned by one user (and one group). When you type `ls -al`, a username appears by each file; this is the owner. Let us analyse the output from this command and see how ownership is relevant.

```
Flags          Links  Owner        Size  Date/time      Name
drwxr-x--x   3 rudolf          512 Jun 30 15:26 .
drwxr-xr-x  20 root            512 Jun 30 15:26 ..
-rwxr-x--x   1 rudolf          261 Jun 30 10:03 .cshrc
-rwxr-x--x   1 rudolf          234 Jun 30 10:03 .login
-rwxr-x--x   1 rudolf          143 Jun 30 10:04 .profile
-rwxr-xr-x   1 rudolf        38180 Jun 30 09:42 a.out
drwxr-x--x   2 rudolf          512 Jun 30 10:03 bin
drwxr-x--x   1 rudolf        53428 Jun 30 10:03 core
-rwxrwxrwx   1 rudolf           17 Jun 30 13:14 tree
-rw-r--r--   1 rudolf         3335 Jun 30 13:46 tree.c
```

Flags: First, `d` for directory. Then, three groups of `r`,`w`,`x`: these stand for read, write and execute permission. The first group is for the owner, the second for the group the owner is in, and the third for everyone else: "user", "group" and "world" in UNIX slang. So, to take an example: `tree.c` is owned by `rudolf`, who can read it and write to it, but not execute it (it's not a program). Other members of `rudolf`'s group can only read it, as can the rest of the world.

Links: Number of hard links to the file. See *Links* below.

Owner: The name (or number if UNIX can't look up the name) of the user who owns the file. The owner can change the flags for the file (see below).

Size, date, time: fairly obvious. The size is in bytes.

Name: the filename (see *Filenames and Wildcards*)

Changing the mode (flags) of a file

This is the task of chmod. A nasty command to learn, it requires some thought.

`chmod [-R]` ***mode file***

The complicated bit is *mode*, which can be specified in two ways.

1. As an absolute octal value. I think this is the easiest. Here, mode is a three-digit octal number (digits 0-7). The first digit represents user; the second group; the third world. Each digit is made up as follows: take 4 for read, 2 for write, 1 for execute. Add them up.

   So, for example, let us say I have a file called `tree.c`, and I want to set the flags as `rwxr-xr--`. Digit one is 4+2+1; digit two is 4+0+1; digit three is 4+0+0. My number is 754. I issue

the command `chmod 754 tree.c`.

Extra-complicated bit. If you specify a *four*-digit octal number, the extra digit in front is composed of the following bits:

4	Set user ID on execution (applies only to executables)
2	Set group ID on execution (applies only to executables)
1	Set sticky bit. Only the superuser can do this. If the sticky bit is set on an executable, nd the file is set up for sharing (the default), the system will not abandon the swap-space image of the program-text (non-data) part of the file when its last user terminates. (This means the file cannot be written or deleted, though directory entries can be removed if one link remains). To replace a sticky file, clear the sticky bit and execute the program to flush the swapped copy. Write the file (impossible if others are using it). If the sticky bit is set on a directory, an unprivileged user cannot delete or rename files of other users in that directory. This is useful for directories such as `/tmp` that must be publicly writable but which should deny users the possibility of arbitrarily deleting or renaming each others' files.

The set-UID and set-GID bits give the process created by running an executable the user/group ID of the owning user/group (typically so it may access privileged data). Note also that the set-UID and set-GID bits are automatically turned off when a file is written or its owner changed, for obvious security reasons.

2. In a symbolic fashion. Here, *mode* is

[who] op permission [op permission] …

without spaces. ***who*** is `u` ("user", the owner), `g` (group) or `o` (others) or a combination of the three. The letter `a` (all) can be used instead of "`ugo`". ***op*** is `+` to assign permission, `-` to revoke permission or `=` to assign permission in an absolute fashion. ***permission*** is any combination of `r` (read), `w` (write), `x` (execute), `s` (set owner or group ID – can only be assigned to `u` or `g`) and `t` ("save text", i.e. sticky). Alternatively, you can use `u`, `g` or `o` as a permission, to set the permission for *who* to be the same as that for `u`/`g`/`o`.

Some examples will help.

`chmod g+x filea`	Gives group execute permission.
`chmod g=x fileb`	Gives group only execute permission.
`chmod g=u filea`	Gives the group the same permissions as currently exist for user.
`chmod o= fileb`	Revokes all permissions for others.
`chmod u+w,g=u filea`	Gives write permission to user, then assigns all current permissions for user to group.

The `-R` flag recursively descends the directory heirarchy – often useful. (However, `chmod` does not change the mode of any symbolic links it encounters, and does not traverse the path associated with the link.)

Changing the ownership of a file

`chown [ -R ]` ***username/number*[.*groupname/number*]** ***filename***
`chgrp [ -R ]` ***group file***

These commands are easy. For example:

```
chown rudolf tree.c
chown root /etc/oodle
chown n-thorpe.oracle random.file
```

Only the superuser can change the ownership of a file; users can change the group of a file they own to another group to which they belong. However, `/etc/chown` isn't usually on users' paths; discourage casual use. In both `chown` and `chgrp`, the `-R` flag recurses subdirectories.

Drives – concept, mounting and dismounting

At the lowest level, a drive is a SCSI device. At the next level in the heirarchy, UNIX sees it as a device (whose file is kept in `/dev`!) that UNIX can talk to with chunks of data called blocks. This is distinct from other devices – "character" devices – that are talked to one byte at a time. There is a system by which UNIX maps the special files kept in `/dev` to the hardware (see *Devices*). These devices have data on them that are organised in a structure that UNIX recognises as a file system. The boot procedure gets the root (`/`) file system up and running, together with the swap space. It then mounts all the partitions.

A disk is mounted under the root file system by mapping it to a directory that is otherwise empty. The syntax is

`/etc/mount` [*device*] [*directory*]

and the drive is dismounted using

`/etc/umount` [*device*] [*directory*]

You can omit either *device* or *directory* and then the system looks up the missing data in `/etc/fstab` (file-system table). If you issue the command `mount -a`, mount looks up all the devices in `/etc/fstab` and tries to mount them all. The boot procedure does this.

An example might help. Let's say you have a disk drive which corresponds to device rz1c (SCSI bus 0, device 1, partition c which is the whole disk) that you want to mount in your empty directory `/programs`. You say

```
/etc/mount /dev/rz1c /programs
```

Here is the full syntax of the mount command as it applies to UFS (local file systems). For details of NFS-specific options, see *NFS* under *Networking*.

`/etc/mount` [*options*] [*device*] [*directory*]

Options:

(none)	Without arguments, `mount` prints the list of mounted file systems.
`-a`	Reads `/etc/fstab` and mounts (or unmounts) all file systems listed there.
`-f`	Fast unmount (NFS only).
`-o` *options*	Passes *options* to the specific file system's mount routine in the kernel. Not for everyday use.
`-r`	Mount read-only. To share a disk, each host must mount it read-only.
`-t` *type*	Specifies the type of file system being mounted. When used with `-a`, all file systems of that type that are in `/etc/fstab` are mounted.
`-v`	Verbose.

The umount command has the syntax:

`/etc/umount` **[** ***options*** **]** **[** ***device*** **]** **[** ***directory*** **]**

Options:

`-a`	Unmounts all mounted file systems. It may be necessary to run this twice.
`-v`	Verbose.

Note:

1. **Mounting corrupted file systems will crash the system** – run `fsck` first!
2. If the directory on which a file system is to be mounted is a symbolic link, the file system is mounted on top of the directory to which the link refers, not the link itself.

`/etc/fstab`

What exactly is the format of `fstab`? Here's one I found lying around.

```
/dev/rz0a:/:rw:1:!:ufs::
/dev/rz1c:/usr:rw:1:2:ufs::
/dev/rz3a:/var:rw:1:4:ufs::
/dev/rz3g:/usr/users:rw:1:6:ufs::
/dev/rz6g:/database:rw:1:8:ufs::
/dev/rz6a:/tmp:rw:1:3:ufs::
/usr/users@pythagoras:/pythagoras_users:ro:0:0:nfs:soft,bg,nosuid
```

The fields are as follows:

1. Name of the block special device on which the file system resides. It can also be a network name for NFS (the network file system), such as `/@discovery`.
2. The pathname of the directory on which the file system is to be mounted.
3. How the file system is mounted:
   - `rw` – read/write
   - `ro` – read only
   - `rq` – read/write with quotas
   - `sw` – make the special file part of the swap space
   - `xx` – ignore the entry
4. The frequency (in days) with which the `dump` command dumps the `rw`, `ro` and `rq` file systems.
5. The order in which the `fsck` command checks the `rw`, `ro` and `rq` file systems at reboot time.
6. The name of the file system type.
   - `ufs` – ULTRIX file system
   - `nfs` – SUN Network File System
7. Options: an arbitrary string that applies to that particular file system. In the NFS entry above, NFS-specific options are listed.

What's CHKDSK in UNIX?

`fsck`(8). With no options, it checks all file systems in `/etc/fstab`. This is an important part of the boot procedure (`fsck` is invoked from `/etc/rc`) as mounting corrupted file systems will crash the system.

Links

A link is a directory entry referring to a file. A file, together with its size and all its protection information, may have several links to it. There are two types of link: hard and symbolic.

A **hard link** to a file is *indistinguishable* from the file itself, and must be on the same file system (i.e. same physical device) as the original file. Hard links cannot refer to directories. There is always at least one hard link to every file: this is its directory entry. (This implies that files are distinct from their directory entries, yet referenced by them, and this is exactly the case. UNIX deletes file by *unlinking* their entries.) If you create a new hard link, you get another directory entry (somewhere, under some name) for the same file. If you modify a file via one of its hard links, it is modified as referenced by any other. Clear? It's the same file.

A **symbolic link** is much the same, except it can span file systems and refer to directories. Furthermore, it can have a mode (see "Changing the mode of a file") different from that of the file to which it is linked. Symlinks are useful mainly as references to directories; for example, the directory `/sys` is a symlink (at least on the machines I'm using) to `/usr/sys`. If you type `cd /sys` followed by `pwd` (print working directory) you will see `/usr/sys`.

Creating links. Use the command

`ln [-f] [-i] [-s]` ***filename linkname***

Options:

`-f`	Forces overwrite of any files that exist.
`-i`	Interactive: prompts if any files already exist.
`-s`	Symlink: make the link symbolic.

If *linkname* is omitted, the link has the same name as *filename*. The current directory is assumed, but *linkname* can also be a directory to put the link in. Fairly obvious when you use it.

Detecting links. If you execute the command `ls -al`, you get information that can help. If the first letter of the flags (the first column, looking like `lrwxr-xr-x`) is an `l`, the entry is a symlink. The last column gives the name of the symlink and what it is linked to (e.g. `sys -> usr/sys`). Hard links, of course, are indistinguishable from other files! However, the second column gives the number of hard links to a file. If this is more than 1, there's another hard link somewhere! A utility like Tree (plug, plug) will allow you to find it, though not with ease: the hard links have the same i-node and device numbers.

NFS – beware

I'd like to warn you about NFS. It's very useful, but think twice before mounting any network drive read-write, as opposed to read-only. Something nasty nearly happened to us: the machine `hubble` (a 'test' machine) NFS-mounted a database drive from `discovery` (a live system) in order to copy some data over. It turned out that `hubble` was running its database from `discovery`'s drive: had anyone chosen to wipe the 'test' database, we'd have had problems.

A little look at system files and directories

Here's a quick summary of a default UNIX installation.

Directory	What's in it
`/`	• Boot file (`ultrixboot` in our case) • Kernel (`vmunix`) • Files used at root's login (`.cshrc`, `.login`, `.profile`)
`/bin`	• Programs that are absolutely part of the core of UNIX (`cp`, `rm`, `mv`, `sh`, `ls`, `mount`, `shutdown`…)
`/dev`	• Device special files. • The `MAKEDEV` device-special-file-making script
`/etc`	• Programs that aren't quite so central to UNIX (`adduser`, `chown`, `lpc`, `ping`…). These are management tools: ordinary users don't have `/etc` on their path. • Many configuration files (`rc`, `rc.local`, `fstab`, `crontab`, `disktab`, `inetd.conf`, `hosts`…)
`/lost+found`	• There's one of these directories at the top of every file system (so there's always `/lost+found`, and often `/usr/lost+found`, `/var/lost+found`…). The `fsck` program saves the UNIX equivalent of "lost clusters" under DOS (i.e. files that are allocated but unreferenced) into this directory.
`/tmp`	• Temporary storage space
`/usr`	• Nothing by itself; `/usr` contains lots of other administrative programs and files, and often everything else that happens on the system. A few important subdirectories are listed below.
`/usr/bin`	• 'User' programs (such as `nice`, `passwd`, `sort`, `touch`)
`/usr/dict`	• Dictionary
`/usr/diskless`	• Files for diskless workstations
`/usr/etc`	• In the same vein as `/etc`
`/usr/examples`	• Programming examples
`/usr/include`	• Header files for C
`/usr/man`	• Manual pages
`/usr/skel`	• Default files for new users: `.cshrc`, `.login`, `.profile` and others for XWindows &c.
`/usr/sys`	• System header files (`./h`), configuration scripts for making new kernels (`./conf`)… all sorts of stuff you hope you never need but probably will.
`/usr/ucb`	• Programs by the University of California, Berkeley. Things like `vi`, `man`, `whoami`, `tail`…
`/usr/users`	• Users' home directories
`/var`	• All sorts of relatively unimportant administrative stuff, including printer spooling (`/var/spool`), XWindows (`/var/X11`), UUCP (`/var/uucp`), system logs (`/var/adm`)

Managing user accounts

Adding users

Run `/etc/adduser` by typing **`adduser`**. You are asked for a username, user ID (use the default supplied!), full name, the login group (usually "`users`", although you can make a new group at this point) and any other groups you want the user to be a member of, the location of the home directory, the default shell and a password. `adduser` sets up a home directory and copies default `.cshrc`, `.login` and `.profile` files into it from `/usr/skel`. The username can contain only lower case ASCII characters ('`a`' – '`z`') and digits (`0` – `9`).

Deleting users

Run `/etc/removeuser` by typing **`removeuser`**. You are asked for the name of the user you wish to remove, and whether you wish to destroy the home directory. Make sure there's nothing you want in there first!

Adding groups

Run `/etc/addgroup` by typing **`addgroup`**. You are asked for the group's name and number.

Removing groups

Edit `/etc/group` manually!

Changing passwords

Run `/bin/passwd` by typing **`passwd [ -afs ] [`** ***name*** **`]`**. If no *name* is supplied, `passwd` operates on your user (if you have `su`'d, the user you now appear to be). With no other options, `passwd` asks you for an old password (unless you are `root`) and a new one.

Options:

`-a`	Supply a list of system-generated passwords. Use this if you want real security
`-f`	Change the finger information (real name, phone number etc.), not the password. This is equivalent to the `chfn` command.
`-s`	Change the login shell, not the password.

Security note: anyone who uses an English word or a name connected to them doesn't care about real security or doesn't understand it. The best passwords are random, like `bx23H5sj` – remember UNIX is case-sensitive. The next best are system-generated; the system generates pseudo-words that are a little easier to remember, such as `kuboit`. Other good passwords are mis-spelled words, such as `oppised`. Words in a dictionary are vulnerable to a dictionary search, and this is a trivial problem for any modern personal computer.

The `/etc/passwd` file

Use **`vipw`** to edit `/etc/passwd`, not "`vi /etc/passwd`". You get the benefit of better locking, database synchronisation and a check that you haven't trashed the root user before it saves.

This file is an ASCII file that contains the following information for each user:

Login name
Encrypted password
User ID
(Primary) Group ID
Real name, office, extension, home phone
Initial working directory
Login shell

Fields are separated by a colon; entries are separated by a new line. If the password field is blank, no password is asked for. If the password field is "`Nologin`" or "`PASSWORD HERE`", that user won't be able to log in. If the shell field is blank, `/bin/sh` is used. The "real name" field can contain an ampersand (&) to stand for the login name; the name and telephone numbers, if present, are separated by commas. Example entries:

```
root:UnoHkGYv74KO.:0:1:System PRIVILEGED Accounts,,,:/:/bin/sh
accounts:1ZuMQiDIiEEA:272:15:Accounts User:/usr/users/accounts:
```

The `/etc/passwd` file can be read by all users. This is superficially secure as the passwords are encrypted by a one-way encryption system, and a reasonable one at that. Part of the security of the algorithm stemmed from the difficulty of obtaining it. This applies less today: programs are readily available which will encrypt every word of a dictionary and compare the encrypted version to an entry in `/etc/passwd`. Some details of the encryption system are listed in the manual under `crypt`(3) (see *Getting Help…*), for those that are interested. Since `/etc/passwd` is obtainable by any user with the most rudimentary file access (certainly any with shell access), most UNIX systems in the academic and corporate sectors are vulnerable to dictionary check hacking. Choose secure passwords!

The `/etc/group` file

This is also accessible to all users, but that's not a problem (except to find out which accounts are worth breaking into). It contains entries with the following fields:

Group name
Encrypted password
Group ID number
Comma-separated list of all users allowed in the group

No more than 200 users are allowed in any group. Also, no more than 1820 characters are allowed on one line of the file. Don't put a user in more than 8 groups: it causes problems with NFS-mounting from old versions of UNIX. If the password field is null, no password is demanded.

Getting information

Type **`id`** to find out who you are and your primary group. If you have `su`'d, you get information about the user you appear to be. Type **`groups`** to find out which groups you are a member of.

Advanced security

Run **`/usr/etc/sec/secsetup`** to change your system's security level. This may involve rebuilding the kernel. You can enable security auditing, trusted path and enhanced login in any combination. See `secsetup`(8) for details. It modifies the mandatory configuration file `/etc/svc.conf`, which you can also edit yourself, but bear in mind that `secsetup` knows when to modify the kernel and you probably don't. The `/etc/svc.conf` file does contain password lengths and expiration time; I guess that modifying these variables takes effect at the next reboot without needing to rebuild the kernel.

With UPGRADE-level password security, if the password entry in `/etc/passwd` is "`*`", the password stored in the `auth` database is used instead. With ENHANCED-level security, the password field in `/etc/passwd` is always ignored. The `auth` database can only be read by the superuser, alleviating most of the vulnerability of `/etc/passwd`.

The `auth` database contains a user-ID key, then the password, the time the password was last modified, the minimum password lifetime, the maximum password lifetime, the account mask (account enabled? can the user change his/her password? is the user allowed to make up a password him/herself?), login failure count, audit ID, audit control and audit mask. See `auth`(5) for details.

You may edit a user's `auth` entry using `/usr/etc/sec/edauth` *username*. However, the editor used is `ed`, which must vie (pun intended) for the title of "Most Unfriendly Editor Ever". I don't expect you to need this command.

Login banners

You can edit `/etc/gettytab` to add a banner message.
The banner message is the `im` field in the `default` entry. Normally it gives the UNIX version. Edit it, then `kill -HUP 1`. Note that `rc.local` (ours, at least) tries to put the version number back in whenever UNIX boots, by searching for "ULTRIX" – if it's not there, the file might come to grief. I suggest that your banner should go on a different line to the UNIX version message, or on the same line but before the version message. Alternatively, edit `rc.local`!

Message of the day

The file `/etc/motd` is displayed immediately after a successful login. This is the normal place to put announcements and instructions.

Trusted path

This is a security system designed to assure a user that the login prompt is genuine and not a Trojan horse trying to capture passwords. If trusted path is running (configured by the script `/usr/etc/sec/secsetup`), pressing the BREAK key followed by RETURN causes the trusted path system to kill all processes on that terminal and return to the login prompt.

Trusted path is not supported for pseudo-terminals. You may need to reconfigure your terminal server's "attention" key to something other than BREAK.

Disabling login

If the file `/etc/nologin` exists, no account other than `root` can log in. The file is displayed to those who try.

Managing processes. Snooping and killing errant tasks.

First, some theory.

What is a process?

A process, or task, is a program. Whereas DOS runs only one program at any time, UNIX is a multi-tasking operating system and runs many. A special program called the scheduler divides the processor's time between processes – "time-slicing" – so they appear to run simultaneously. This is the whole point of UNIX.

So what's to prevent my process from spying on your process's memory, or writing random information to it, or suddenly redirecting your highly confidential information to my screen? UNIX prevents any process from reading or writing another process's memory directly. Indeed, the most common programming error is to read or write to a "floating pointer"; when such a violation occurs, UNIX will return an error, and unless the program traps that error signal it will crash and "dump core" – write the state of the process to a file called `core` that in theory can be used for debugging. Inter-process communication is handles through *signals*, *pipes* and *sockets*.

Signals

If you want a process to do something, you can't write to its memory; you must sent it a signal. UNIX has a set of valid signals that processes may send to each other ("hang up", "interrupt", "quit", "kill", "illegal instruction", "user signal"...) Programs can arrange for parts of themselved to be called when their process receives a certain signal. Some signals cannot be trapped in this way, notably "kill".

Pipes

Signals enable processes to transfer simple information. For data transfer, a more complex method is needed: the pipe. A pipe is a "channel" between two processes. A process allocates two file descriptors for this purpose, then forks (see *Forking* below). The two child processes thus created can read using one file descriptor and write to the other, and cooperating in this manner can transfer data.

You probably use pipes under DOS and UNIX all the time. They're the means by which output from one command can be turned into input to another. When you type `ls -al | more`, the shell creates a pipe, then forks (see *Forking* below). One of the child processes runs "`ls -al`" after making the standard output channel (`stdout`) a copy of the pipe's write channel. The other runs "`more`" after making the standard input channel (`stdin`) a copy of the pipe's read channel. The result is a transfer of data directly from one process to another.

Sockets

Sockets are like pipes, but the information is carried in a different way. If you want your process in Kent to talk to another in Dallas, pipes won't do. A socket can provide reliable two-way communication between processes on the same machine, or between processes anywhere on the planet. If you're an Windows Internet user, you might have heard of WINSOCK: this is a program that provides sockets to programs and connects these sockets to the TCP/IP communications protocol for transfer around the world. Sockets come in four types:

Socket type	Purpose
`SOCK_STREAM`	sequenced, reliable, 2-way communication byte stream with a mechanism for out-of-band transmission
`SOCK_DGRAM`	datagrams (connectionless, unreliable messages of a fixed maximum length, typically small)
`SOCK_RAW`	provide access to internal network interfaces (superuser only)
`SOCK_SEQPACKET`	for DECnet communications; ignore

Forking

I've mentioned forking a couple of times now without explaining it. It's fun and central to UNIX. Each process has a reference number, a process ID. There is a function called `fork()`, and when a process calls this UNIX makes *another copy* of the process. This copy is called a *child*; the original is now a *parent*. The child is an exact copy of the parent, aside from having a different process ID and parent process ID (the process ID of the parent). The `fork()` call returns 0 to the child and 1 to the parent, so they can distinguish themselves.

You may think this rather erudite, but it happens all the time. Consider the shell: this must run programs. It runs programs as separate processes and must not itself be destroyed in the process (no pun intended). There is no low-level command in UNIX to start a separate process like this. Odd, you might think, but how would you implement the system? What the shell does is this. First, it forks. The child process *transforms itself* into the program to be executed (destroying itself in the… process), using the `execve()` function. The parent process then waits, using the `wait()` function, for its child to terminate. Of course, there are functions to do just this, such as `system()`, but it's informative to know what's happening "under the bonnet".

The practical benefit of this system is that you have the option of *not* waiting for the child to terminate. If you append an ampersand (`&`) to a command, the shell reports its child's process ID (in case you want to kill it, or whatever) and returns immediately. The output from the background process still goes to your terminal, unless you redirect it, but (obviously) there is no input ("the default standard input for the command is the empty file `/dev/null`") unless you direct some to it from a file. The **`wait`** command can be used to wait for all child processes to terminate.

How to crash UNIX

Do not try this on a system that someone cares about! Make sure all data is saved and **sync**ed beforehand. Compile and run this C program, and your system will crash:

```
main()
{
  while (1) fork();
}
```

This code does nothing but fork. Each `fork()` makes two copies of the process, each of which forks... the system becomes unresponsive within seconds (if you're using a workstation, you'll notice that the mouse is fine, because it runs on hardware interrupts, but the CapsLock light responds about two minutes after you press the key). You'll have to turn it off – killing the process won't work. Although killing a process kills its children too, UNIX is time-slicing: while the system is removing processes from the "top", at the "bottom", processes are being created.

The alarming thing about this code is that any user has the authority to crash the system. UNIX never forbids a process to fork on that basis of who owns it, only on the basis of having run out of process space (which is what happens when you run this code – only it doesn't care, it keeps trying). If you give users access to a command line, they can crash the system. Consequently, this is information to be carefully controlled. I justify its inclusion in this guide because you, the readers, are administrators and should be aware of the danger, and because if you want to try it you can find a spare machine. If you tell a user without the responsibility of running a system, you risk your data.

Swapping out

UNIX implements virtual memory properly. If it hasn't got any RAM free to give a process, it takes another process and writes its image (the "pseudo-computer" that the process it, including memory, CPU register values, open files, current directory and so on) to an area of the file system known as *swap space*, freeing up the RAM in the process. When the process that is swapped out needs to run, UNIX shuffles its memory around and gets the process back from disk.

UNIX says it needs a swap space about three times the RAM size. This means we waste 600Mb of disk space on our main system alone. Heigh ho. For details on managing swap space, see `/etc/fstab` under *The UNIX File System*.

Finding out about processes

On a practical level, administrators often have to kill crashed or otherwise errant processes. To send a signal to a process, you need to know its process ID. Let's look at ways of finding this out.

`w` – what are people doing?

Here's a sample result of the `w` command:

```
 10:49am  up 21:22,  2 users,  load average: 1.80, 1.54, 1.01
User     tty from           login@  idle   JCPU   PCPU  what
oracle   co                Thu 1pm 20:31  14:03   7:31  orapopskc 272 273 10
root     p0 1.5.0.99        8:34am           46      1  w
```

The first line is also what you get from the `uptime` command. It's 10:49; the system has been up for 21 hours. There are two users on, and there have been 1.80, 1.54 and 1.01 jobs in the run queue on average for the last 1, 5 and 15-minute periods respectively. This is an indication of how busy the system is.

Then come the users. `oracle` has been logged into the console (`co`, file `/dev/console`) since Thursday afternoon. No characters have been typed into that terminal for 20 hours. The `JCPU` field indicates the CPU time used by all processes and their children on that terminal. It might be hours:minutes; then again, it might be minutes:seconds. Probably the latter. `PCPU` is the CPU time used by the currently active process. "`what`" is the name and arguments of the current process. **Note** that `w` takes an educated guess as to which process is the "current" one. Don't rely on it entirely; use `ps` as well. Nonetheless, `w` is a very useful summary of what's happening on the system. You can also use "`w` *user*" to restrict the information to one user.

`ps` – process status

This is the command to get detailed information.
Simplified syntax:

`ps [` ***options*** **`]`**

Useful options: (note that these are quite specific to the UNIX version)

- `-#` Gives information about process number #. (This must be the last option given and cannot be used with `-a` or `-t`.)
- `-a` Displays information for processes executed from all users' terminals, not just from your terminal. (Cannot be used with `-#` or `-t`.)
- `-c` Displays the command names as stored internally in the system for accounting, not the command arguments which are kept in the process addresss space. This is less informative but more reliable as a process can destroy information in its address space.
- `-e` Displays the environment as well as the command arguments
- `-g` Displays all processes within the process group, not just process group leaders. This will show you the boring things like top-level command interpreters and processes waiting for users to log in.
- `-l` Displays information in long format, including the fields `PPID` (parent process ID)
- `-t`*x* Displays information for terminal *x* only. (*x* is `co` for the console, `?` for processes with no terminal, blank for the current terminal, `p3` for ttyp3 etc.)
- `-u` User-oriented output, including the fields `USER`, `%CPU` and `%MEM`.
- `-w` Produces 132-column output.
- `-ww` Produces arbitrarily wide output.

`-x` Displays information for all processes, including those not executed from terminals.

Output fields:

`PID` Process ID number

`TT` Control terminal

`TIME` User + system time

`STAT` State of the process, given as a sequence of five letters (e.g. `RWNAV`).

First letter: run status

`R` – running
`T` – stopped
`P` – in page wait
`D` – in disk (or other short-term) wait
`S` – sleeping for less than about 20s.
`I` – idle (sleeping for longer than about 20s)

Second letter: swapped out?

`W` – swapped out
`Z` – killed, but not yet removed ("zombie")
– process is in core (RAM)
`>` – process has a specified soft limit on memory and is exceeding it. (Not swapped.)

Third letter: altered CPU priority? (See *Be nice*)

`N` – priority reduced
`<` – priority artificially raised
– no special treatment

Fourth letter – special virtual memory state?

`A` – never mind
`S` – never mind
– normal

Fifth letter – vector process?

`V` – process using vector hardware. VAX only.
– not using vector hardware

`USER` Names the process' owner.

`%CPU` CPU usage. Not very accurate.

`NICE (NI)` The process scheduling increment (see *Be nice*).

`SIZE (SZ)` Virtual size of the process in 1024-byte units.

`RSS` Real memory ("resident set") size of the process in 1024-byte units.

`LIM` Soft limit on memory used or "`xx`" if none.

`TSIZ` Size of the text (shared program) image.

`TRS` Size of the resident (real memory) set of text.

`%MEM` Percentage of real memory used by the process.

`RE` Residency time (seconds in core).

`SL` Sleep time (seconds blocked)

`PAGEIN` Number of disk I/O operations by the process that referred to pages not in core.

`UID` User ID.

`PPID` Parent process ID.

`CP` Short-term CPU use factor, used in scheduling.

`PRI` Process priority. (Negative if the process is in a wait state that cannot be interrupted.)

ADDR	Swap address of the process or page frame of the beginning of the user page table entries.
WCHAN	The event the process is waiting for (an address in the system with the initial part of the address truncated).
F	Flags. See ps(1). Useful for debugging only.

A process that has a parent and has exited, but for which the parent has not waited, is marked <defunct>. A process that is blocked trying to exit is marked <exiting>.

Processes can get into a state where they are called an **orphan**. I can't find this in the manual (because the index is wrong), but I assume it's when a process's parent has been killed. Normally, of course, **killing a process kills its children**.

Common commands for administrators to use are "ps -aux" and "ps -auxww". Remember you can pipe it to grep rather than remember all the options for ps.

Sending signals to processes; how to kill processes

To send a signal to a process, use the kill command. Syntax:

kill [-*sig*] *processid…*
or **kill -l**

Note that you need to know the process ID number (use the ps command, above). Without any *sig* option, kill sends the TERM (terminate) signal to the process(es). If you put in a *–sig* argument, that signal is sent instead. kill -l lists the valid signals.

The default signal, TERM, is a gentle one: "please go away". Processes can ignore it. **If a process won't die, use "kill –9 *processid*".** This unceremoniously kicks it out. Signal 9 can also be referred to as KILL.

To kill a process, it must belong to you or you must be superuser. Please note that anyone can kill their own processes! The less time you spend as superuser, the smaller your chances of doing something wrong.

Be nice

If you execute a command as

nice [-*number*] *command* [*arguments*]

its priority is altered. Positive numbers lower the priority and negative numbers raise it (superuser only). The range is –20 to +20. Default is +10. If you wish to change the priority of a process that is already running, use:

/etc/renice *priority* [[-p] *processID*] [[-g] *processgroup*] [[-u] *user*]

Normally, the number you give is interpreted as a process ID; using -p, -g or -u forces renice to interpret the number as a process ID, process group ID or user ID respectively. This feature enables you to change the priority of all process belonging to a user (or process group) at once.

Two points:

1. If you make a process' priority very negative, it cannot be interrupted. To regain control you may have to make it greater than zero again.
2. Non-superusers cannot increase the scheduling priority of their own processes, even if they were the ones that originally decreased it. Heh, heh.

Monitor thyself for evil

Not part of UNIX, the **`monitor`** command gives you an interactive view of your system. If it's on your system, it should be in `/usr/bin`.

Keys. Within `monitor`, press `h` or `?` for help. Press `m` to "magnify" any field. You are presented with a list of options; `j` and `k` move the cursor up and down, `s` selects a field. Press `u` ("unmagnify") to return to the main screen. Press `q` to quit.

Notes. Page faults aren't anything nasty – a page fault occurs when a process asks for memory that's swapped out.

Other status commands

`uptime`	display system status
`cpustat`	report CPU statistics
`iostat`	report I/O statistics
`lpstat`	printer status information (see *Printing*)
`netstat`	show network status (see *Networking*)
`pstat`	print system facts This reports on the internal system tables and can give very detailed information. See `pstat`(8) for details.
`ipcs`	report interprocess communication facilities status
`vmstat`	report virtual memory statistics
`df`	**display free and used disk space** (see *Devices / Disks*)

Devices

UNIX aims to make hardware device access as transparent as possible, by presenting a uniform device-independent layer to programs and by enclosing all device-specific code in the operating system kernel itself. To this end, each supported I/O device is associated with at least on "special" file. Special files are read and written just like ordinary files, but requests to read and write activate the associated device. An entry for each special file resides in the `/dev` directory (though links can be elsewhere). These special files are protected from indiscriminate access, though few protections apply to the superuser, who must be wary. Device special files can be of block or character type, depending on whether they support the transfer of blocks of data; thus disks are block devices and terminals are character devices.

The "`file` *filespec*" command can be used to examine a device special file. This command examines files and tells you what type of file they are. It recognises executables, directories, links, empty files, C files, ASCII files and many other types… and also devices. Here it is exceptionally useful, as it gives details of device types, SCSI IDs, tape device status (offline? write-locked?) and so on.

Making devices

Device special files are usually created with the `/dev/MAKEDEV` script. The syntax is

`/dev/MAKEDEV` ***devicename*?**

where *devicename* is a supported device and ? is a logical unit number. For example, the *devicename* of a SCSI disk is `rz` and the LUN is its SCSI ID plus eight times its SCSI controller number, so to make a set of device special files for a SCSI disk on controller 0, SCSI ID 5 you would type `/dev/MAKEDEV rz5`. You may look up *devicename*s by examining `/dev/MAKEDEV` itself.

`MAKEDEV` calls `/etc/mknod` to make the device special file, having determined whether the device is block-type or character-type, calculated the major and minor device numbers (specific to each system and obtained from the system source file `conf.c`) and assigned the file a name.

A log of what `MAKEDEV` has done is kept in `/dev/MAKEDEV.log`.

Please remember that the "*devicename?*" and the device special file name(s) differ. Generally, one *devicename?* has one or more associated device special files. For example, "`/dev/MAKEDEV rz6`" makes the device special files `/dev/rz6a`, `/dev/rrz6a`, `/dev/rz6b`, `/dev/rrz6b`, `/dev/rz6c`… We will look at specific device types next.

Null

Let's start with an easy one. There is a character-special file, `/dev/null`, that is a "data sink". Data written to it is discarded; reads from it always return zero bytes. It is typically used to suppress output from a command (`command > /dev/null`).

Memory

There are two device special files which address memory: `/dev/mem` (addressing physical main memory) and `/dev/kmem` (addressing virtual main memory). Unless you are a godlike programmer willing to take risks, these are not for you.

Disks

I will only deal with SCSI disks here. The *devicename* to be passed to `MAKEDEV` is `rz`; the LUN is (`SCSI_ID + 8 * SCSI_CONTROLLER_ID`). Sixteen device special files are created. Each begins rz (block-type) or rrz (character-type). Then comes the LUN. Finally there is a letter that refers to the disk partition. Usually, `a` is the root partition, `b` is the swap partition, `c` is the whole disk; partitions `d` – `h` vary more and may not be used. As a side issue, `chpt` (along with `newfs`) is the command to redo a partition table, but it's only really of use when installing UNIX, and rarely even then.

For a disk to be mounted automatically, an entry should go in `/etc/fstab` (see */etc/fstab* under *The UNIX File System*). Note that it is the block device that you mount (`rz6c`, not `rrz6c`).

Tapes

SCSI tapes use *devicename?* `tz*`. The LUN is made up in the same way as for disks. It is irritating, but the special files' names are numbered from zero in creation order, not by LUN as for disks. Thus if you have a tape drive on LUN 4 and another on LUN 5, and you run `MAKEDEV tz4 tz5`, you will end up with `/dev/rmt0*` and `/dev/rmt1*`, not `/dev/rmt4*` and `/dev/rmt5*`. Use the **`file`** command to map device files to SCSI IDs.

Eight special files are created per tape drive, all of them character-type. The name is composed of `r` (rewind automatically when the file is closed) or `nr` (no rewind) followed by `mt` ("magnetic tape"), followed by the number, followed by `a`, `h`, `l` or `m`. These letters indicate the tape density (something, low, medium, high), relative to the capability of the drive – see `mtio`(4) for details. Here is the result of `file *mt0*` on a system with one TK50 tape drive on SCSI controller 0, SCSI ID 4:

```
nrmt0a: character special (55/60) SCSI #0 TK50 tape #4 write-locked 6666_bpi
nrmt0h: character special (55/44) SCSI #0 TK50 tape #4 write-locked 6666_bpi
nrmt0l: character special (55/36) SCSI #0 TK50 tape #4 write-locked 6666_bpi
nrmt0m: character special (55/52) SCSI #0 TK50 tape #4 write-locked 6666_bpi
rmt0a:  character special (55/56) SCSI #0 TK50 tape #4 write-locked 6666_bpi
rmt0h:  character special (55/40) SCSI #0 TK50 tape #4 write-locked 6666_bpi
rmt0l:  character special (55/32) SCSI #0 TK50 tape #4 write-locked 6666_bpi
rmt0m:  character special (55/48) SCSI #0 TK50 tape #4 write-locked 6666_bpi
```

See *Using tape drives* below for commands that manipulate tape drives.

Terminals

Console and serial line terminals are created when you install UNIX.
The *devicename?* `pty*` creates sets of 16 network pseudo-terminals (TCP/IP protocol).
The *devicename?* `lta*` creates sets of 16 network local area terminals (LAT protocol).
For example, if you want some LAT terminals, you run `MAKEDEV lta0`. This makes 16 device files. If you run out, run `MAKEDEV lta1` to make some more, et cetera.

Terminal device special files are named `/dev/tty*`. The console is `/dev/console`; serial and LAT terminals are `/dev/tty`*xx*; network pseudo-terminals are `/dev/ttyp`*xx*, `/dev/ttyq`*xx*, `/dev/ttyr`*xx* and so on.[2]

If you examine the ownership of the device special files, you will see that they are always owned by the user logged into them at the moment, or root if they are not in use. Furthermore, the `/dev/tty` special file refers to whichever terminal *you* are logged into.

Like `/etc/fstab` for disks, UNIX must also be told about which terminals to use. The terminal database is `/etc/ttys`. If you refer to *How UNIX Starts*, above, you will see that `/etc/init` is run. Well, in multiuser operation `init` creates a process for each terminal port where a user may log in. To do this, it reads `/etc/ttys`. For each terminal marked "`on`" in this file, `init` forks and invokes the command specified on that line in the file (usually `getty`, which reads the user's name

[2] "tty" stands for teletype.

and invokes `login` to log in the user and execute the shell). The command is passed the name of the terminal as the last argument. When the shell ultimately terminates, the main part of `init` wakes up and removes the appropriate entry from `/etc/utmp`, which records current users. `init` then makes an entry in `/usr/adm/wtmp`, where a history of logins and logouts is kept. Then the appropriate terminal is reopened and `getty` is reinvoked.

The `init` command catches the hangup signal (signal number 1, `SIGHUP`) and interprets it to mean that `/etc/ttys` should be re-read. The shell process on each line of `ttys` which used to be active (but isn't) is terminated; a new process is created for each line; lines unchanged in the file are undisturbed. **Therefore, when you have edited `/etc/ttys`, issue the command `kill -HUP 1` to implement the changes.** Incidentally, `kill -TERM 1` will shut the system down back to single-user mode, and `kill -TSTP 1` will tell init to stop creating new processes, so the system slowly dies away as users log off and can no longer log on. A later hangup (`HUP`) will restore full multiuser operation, and a `TERM` will initiate a single-user shell. Note that 1 is the process ID of the main part of `init`.

Format of `/etc/ttys`.
First comes the name of the terminal (the file in the `/dev` directory); then the command associated with it (usually `getty`); then the terminal type (`vt100`, `vt200`, `dialup`…); then any flags. Fields are separated by tabs or spaces. A field with more than one word should be enclosed in double quotes. Comments are preceded by a hash (#).

Legal terminal types for your system can be found in `/etc/termcap`.
Valid getty entries can be found in `/etc/gettytab`.

The flags possible are:

`on`	Enables login
`off`	Disables login (default)
`secure`	Allows root to log in on this terminal, assuming logins are permitted (off by default)
`su`	Allows a user to `su` to root (off by default)
`nomodem`	Line ignores modem signals (default)
`modem`	Line recognises modem signals
`shared`	Line can be used for incoming and outgoing connections (off by default)
`termio`	Line will open with System V default termio attributes (by default, Berkeley defaults are used)[3].
`window="`*string*`"`	Here, *string* is a window system process that `init` maintains for the terminal line.

Assorted examples to illustrate these options:

```
console "/etc/getty std.1200" vt100 on secure # Console at 1200bps, 7-bit
ttyd0 "/etc/getty d1200" dialup on            # Dial-up line at 1200bps
tty01 "/etc/getty std.9600" vt100 on          # Serial line; 7-bit VT100
tty01 "/etc/getty 8bit.9600" vt100 on  # The same terminal in 8-bit mode
ttyp0 none network
ttyp1 none network off                        # Network pseudo-terminals
# Type the following all on one line
:0 "/usr/bin/login -P /usr/bin/Xprompter -C /usr/bin/dxsession -e" none on
   secure window="/usr/bin/Xcfb"              # An X-Windows terminal
tty02 "/etc/getty 8bit.9600" vt100 on modem secure  # LAT terminal
```

Useful examples:

[3] UNIX has two historical 'flavours': BSD (Berkeley Systems Development, from the University of California at Berkeley) and System V. One of these is the same as AT&T UNIX, but I can't remember which. ULTRIX is a Berkeley UNIX clone.

```
console      "/etc/getty std.9600" vt100      on secure    # console
tty00  "/etc/getty 8bit.9600" vt100           on secure    # direct connect
tty01  "/etc/getty std.9600" vt100            on secure    # direct connect
tty11  "/etc/getty std.9600" vt100            on modem     # LAT
tty12  "/etc/getty std.9600" vt100            on modem     # LAT
tty15  "/etc/getty std.9600" vt100            off          # Laser printer
ttyd0  "/etc/getty std.9600" vt100            off shared   # Modem line
ttyp0  none   network      secure             # Network pseudo-terminal
ttyp1  none   network      secure             # Network pseudo-terminal
```

LAT configuration

Hah. Did you think that was all? No chance.

The LAT (Local Area Transport) protocol is used by terminal servers to talk to their hosts. It is relevant to both terminals (*vide supra*) and printers (*vide infra*). Having made your LAT special file (`/dev/tty`*xx*) using `MAKEDEV`, you must ensure LAT is loaded on your system. You must also tell the LAT system if any of your terminals are to be used for host-initiated connections only – in other words, for printing.

The `/etc/lcp` command is used to start LAT (from `rc.local`) and to administer it interactively. Options for `lcp`:

`-s` Starts LAT service. Enables connections from LAT terminal servers to host. If LAT parameters have not been set, they take on default values specified in the `–r` option.

`-r` Resets LAT parameters to the following default values:
multicast timer: 30 seconds
nodename: *hostname*
node description: "`ULTRIX`"
servicename: *hostname*
service description: "`ULTRIX LAT SERVICE`"

`-g` Sets groups. Never mind.

`–h` Sets a list of ttys (the next argument, separated by commas with no spaces) to be available only for host-initiated connections. (You cannot use a backslash or a carriage return to break a string; it must all be on one line). Optionally, you may associate a tty with a specific port on a specific terminal server by following the tty name with the name of the server and port, separated by colons. For example:

```
/etc/lcp -h /dev/tty15:LAT_SERVER:PORT7
```

`-H` Sets a list of ttys as being available only for terminal server-initiated connections.

`-m` Sets multicast timer (next argument, in seconds; range 10 – 255, default 30).

`-n` Sets nodename to the next argument. A LAT node must have a nodename for a terminal user to establish a connection. The nodename must be unique on the Ethernet.

`-N` Sets node description to the next argument.

`-v` Lists the services the node offers (default is one service, *hostname*). You can offer more than one service and associate each service with certain ttys, as in this example (all on one line):

```
/etc/lcp -v mainservice -v
SERV1:/dev/tty15,/dev/tty16 -v
SERV2:/dev/tty17,/dev/tty18,/dev/tty19
```

Here, `tty15` and `tty16` are used for `SERV1`; `tty17`, `tty18` and `tty19` are used for `SERV2`; all other LAT ttys are used for the default service, `mainservice`. Note that the first service listed is used as the default. Note also that if you use this interactively, the new list

	completely replaces the old.
`-V`	Sets service description. If you define multiple services, the first `-V` corresponds to the first `-v`, and so on.
`-t`	Stops LAT service.
`-d`	Display LAT characteristics.
`-z`	Zeroes error counters.
`-c`	Displays error counters
`interval`	Continuously displays error counters, with *interval* seconds between each iteration.
`-p`	Shows which LAT server/port a given tty is connected to. Example: `/etc/lcp -d /dev/tty15`

Here's the simplest entry in `rc.local` to start LAT:

```
lcp -s
```

Here's a typical entry in `rc.local`, defining printer ports with `-h`:

```
[ -f /etc/lcp ] && {
      /etc/lcp -s -v discovery -V "South Kent College DECsystem
5000/240" -h /dev/tty33,/dev/tty16,/dev/tty15,/dev/tty34 & echo
'LAT... ' > /dev/console
}
```

For explanation of the shell language used here, see *Shells and Shell Scripts.*

Printers

A typical system will have up to three types of printers: printers plugged into the host, printers plugged into a terminal server, and "remote" printers (attached to another host). **Local and terminal-server printers must have a device-special file.** See *Terminals* above for details of how to create a new terminal file. Login should be disabled on terminals attached to printers (the "`off`" flag should be present in `/etc/ttys`). For LAT printers, the terminal device needs no knowledge of the LAT server or port – all LAT terminal device files are equivalent.

Printers are described in `/etc/printcap`, the printer capability dabase. There is one entry per printer. **A change to printcap immediately affects the spooling system, unless the affected queue is active.** In this case, the spooling queue should be stopped and restarted (see *Printing*). Fields in printcap are separated by colons (`:`); theoretically each entry is one line, so each line but the last must end with a backslash (`\`) so the next line is regarded as a continuation. The first entry gives the printer's name(s), separated by a pipe (`|`). The first name is displayed in the `lpc` command (see *Printing*). The last name given typically identifies the printer fully.

For local printers, the `ct` field should be set to `dev`. For LAT printers, set `ct` to `lat`. Of course, LAT must be running and the printer's `/dev/tty*` file must be set for host-initiated LAT connections only – see *LAT configuration* above.

When a file is printed using the `lpr` command (see *Printing*) and no printer is named, and no printer name is defined in the `PRINTER` environment variable, the printer named "`lp`" is used. **There should always be a printer named "`lp`" in printcap.**

You will need to make a **spool directory** (usually in `/usr/spool`) for the new printer, and refer to it in `printcap`. Do not be overly concerned with the topic of **filters**: these days, it is the job of the application to know what kind of printer you are using and UNIX shouldn't filter anything. Simple dot-matrix printers may benefit from the `lpf` filter (`of=/usr/lib/lpdfilters/lpf`) but for graphical printers no output filter should be specified, merely the "transparent" – i.e. "do nothing" – filter, `xf` (`xf=/usr/lib/lpdfilters/xf`). It is a good idea to refer to **log files**, but nothing much should happen on them.

There is a shell script, **`/etc/lprsetup`**, to help you administer printers. It is quite self-explanatory and knows about all the possible parameters for the `printcap` database. When you create printers with `lprsetup` it makes the spool directory, links output filters and creates a `printcap` entry for you.

The best way to get a feel for a complex file is to look at some working entries. Here's an extract from a typical `printcap`, with explanatory notes beside the fields.

```
# @(#)printcap  3.1 (ULTRIX) 4/20/90

lp0|lp|0|local line printer:\
       :lp=/dev/lp:\
       :of=/usr/lib/lpdfilters/lpf:\
       :sd=/usr/spool/lpd:\
       :lf=/usr/adm/lpd-errs:
```

- Here are two printers plugged into a LAT terminal server:

```
# Brother HL8-E on DECServer 300 : CHALLENGER Port 16
finance laser|lp1|1|FINANCE LASER|FL|fl:\
       :af=/usr/adm/lp1acct:\          Accounting file name
       :br#9600:\                      Baud rate
       :ct=lat:\                       Connection type (dev,lat,remote,network)
       :fc#0177777:\                        If printer a tty, clear octal flag
values…
       :fs#023:\                       If printer a tty, set octal flag values…
       :ff=^L:\                        Form feed string
       :fo=true:\                      Print form feed when device opened
       :lf=/usr/adm/lp1err:\           Error logging file name
       :lp=/dev/tty16:\                Device name to open for output
       :mx#0:\                         Maximum file size (kbytes) or 0
       :op=PORT_16:\                        The "name" field for LAT ports
       :os=:\                          Service name (for some terminal servers)
       :pl#72:\                        Page length (lines)
       :pw#255:\                       Page width (characters)
       :sd=/usr/spool/lpd1:\           Spool directory
       :ts=CHALLENGER:\                LAT terminal server name
       :xc#0177777:\                        If it's a tty, clear local mode flags
(octal)…
       :xf=/usr/lib/lpdfilters/xf:\    Transparent mode filter
       :xs#044000:\                    If printer a tty, set local mode flags (octal)…
       :cf=/wp/shbin/wpp:              Cifplot data filter

# Finance la70 Dot Matrix on CHALLENGER PORT 15
la70|lp2|2|LA70|fd|Finance Dot:\
       :af=/usr/adm/lp2acct:\
       :br#9600:\
       :fo=true:\
       :ff=^L:\
       :ct=lat:\
       :fc#0177777:\
       :fs#023:\
       :lf=/usr/adm/lp2err:\
       :lp=/dev/tty15:\
       :mx#0:\
       :of=/usr/lib/lpdfilters/lpf:\   Output filtering program name
       :op=PORT_15:\
       :os=:\
       :pl#66:\
       :pw#255:\
       :sd=/usr/spool/lpd2:\
       :ts=CHALLENGER:\
       :xc#0177777:\
       :xf=/usr/lib/lpdfilters/xf:\
       :xs#044000:
```

- This one doesn't look very important!

```
ether:\
       :lp=/dev/null:
```

- Here's another LAT printer, but running off a different terminal server:

```
# MIS Fujitsu top printer on JUPITER port 3
MIS_FUJITSU|mis_fujitsu|mf|MF:\
       :af=/usr/adm/lp3acct:\
       :br#9600:\
       :fo=true:\
       :ct=lat:\
       :fc#0177777:\
       :fs#023:\
       :lf=/usr/adm/lp3err:\
       :lp=/dev/tty34:\
       :mx#0:\
       :op=PORT_3:\
       :pl#66:\
       :pw#255:\
       :sd=/usr/spool/lpd3:\
       :ts=JUPITER:\
       :xc#0177777:\
       :xf=/usr/lib/lpdfilters/xf:\
       :xs#044000:
```

- Here's an entry for a Novell printer being accessed remotely:

```
laserjet4:\
       :lp=:\
       :rp=mis_laserjet4:\              Remote printer name
       :ff=:\
       :sd=/usr/spool/lpd17:\
       :rm=enterprise-ii:\              Machine name for remote printer
        :mx#0:
```

For full details of all the options, see `printcap`(5).

Daemons

Concept

A daemon is a system process; it is not associated with any terminal. Daemons are usually invoked from `rc` or `rc.local` at boot time, but may be started interactively by the superuser. When they run, they dissociate themselves from the terminal that created them, return control to the process that called them (obviously, the daemon forks, with one process ending and the other losing its terminal) and live on in the system.

A brief summary of common daemons

There are daemons to do all sorts of things, from network routing to managing the swap space. Here is a list of some of the common ones.

Daemon	Function
`idleproc`	[UNIX internal] Process that's run when nothing else is happening
`pagedaemon`	[UNIX internal] Memory page manager
`swapper`	[UNIX internal] Swap space manager
`/etc/cron`	Clock daemon
`/etc/elcsd`	Error logging daemon
`/etc/init -a`	Process control initialization
`/etc/syslog`	System message log daemon
`/etc/update`	Periodically updates the super block of the file system
`/usr/lib/lpd`	Line printer daemon
`Networking:`	
`/etc/biod`	NFS asynchronous block I/O daemon
`/etc/inetd`	Internet service daemon
`/etc/lcp`	LAT control daemon
`/etc/mop_mom`	MOP down-line/up-line load listener (for booting terminal servers!)
`/etc/mountd`	NFS mount request daemon
`/etc/nfsd`	NFS server daemon
`/etc/portmap`	DARPA Internet port to RPC program number mapper
`/etc/routed`	Network routing daemon
`/etc/snmpd`	SNMP (Simple Network Management Protocol) Agent
`/etc/telnetd`	DARPA Telnet protocol server
`/usr/etc/lockd`	Network lock daemon
`/usr/etc/rwalld`	Network rwall server
`/usr/etc/statd`	Network status monitor daemon
`/usr/lib/sendmail`	Internet mail sending daemon
`tpathd`	Trusted path daemon

cron: scheduling processes, such as backups

The clock daemon `/etc/cron` executes commands at specified dates and times according to the instructions in `/usr/lib/crontab` (which is a symlink to `/etc/crontab`). Once a minute, `cron` reads `/etc/crontab` and decides what to do. **Note commands run by `cron` have `root` authority.**

Format of `/etc/crontab`

The format of a line in `crontab` is as follows:

minute hour day month weekday command

Field definitions:

minute	(0 – 59) The exact minute that the command executes
hour	(0 – 23) The hour of the day that the command executes
day	(1 – 31) The day of the month…
month	(1 – 12) The month of the year…
weekday	(1 – 7) The day of the week; Monday = 1, Tuesday = 2…
command	The complete command sequence to be executed. It must conform to Bourne shell (`sh`) syntax.

The five integer fields may be specified as follows:

- a single number in the specified range
- two numbers separated by a minus (–), meaning an inclusive range
- a comma-separated list of numbers, meaning any of the numbers
- an asterisk, meaning all legal values

A percent (%) symbol in the sixth field is translated to a new-line character. Only the first line of this field (up to a percent sign or end-of-line) is executed by the shell; the other lines are made available to the command as standard input.

Sample crontab:

```
* * * * * /usr/lib/atrun
0 04 * * 1,2,3,4,5 /backup >>/backup.log 2>&1
15 4 * * * ( cd /usr/preserve; find . -mtime +7 -a -exec rm -f {} \; )
5 4 * * * sh /usr/adm/newsyslog
15 2 1 * * for i in /usr/man/cat[1-8]; do df $i | grep -s /dev && find $i -
type f -atime +28 -a -exec rm {} \; ; done
0 02 * * 2-6 /etc/mailem.bat >> /mail.log 2>&1
```

For example, `/usr/lib/atrun` is run every minute; `/backup` is run at 4am on Mondays to Fridays inclusive.

The `at` and `batch` commands

Syntax:

`at` ***time*** **[** ***day*** **] [** ***file*** **]**
`at -r` ***job…***
`at -l` **[** ***job…*** **]**
`batch` **[** ***file*** **]**

The `at` and `batch` commands use a copy of the named file (or standard input) as input to `sh` or `csh` at a later time. A `cd` command to the current directory is inserted at the beginning, as are assignments

to all environment variables. (Note however that open files, traps and priority are lost.) The script runs with the user and group ID of the creator of the copy file (the person who ran `at` or `batch`).

The `at` command allows the user to specify the time at which the command should be executed, while commands queued with `batch` execute when the load level of the system permits.

If a user's name appears in `/usr/lib/cron/at.allow`, they can use `at` and `batch`. If there is no `at.allow`, they will be denied access if their name appears in `/usr/lib/cron/at.deny`. If neither exists, only the superuser can submit jobs (to give everyone access, have an empty `at.deny`). These files consist of one user name per line.

Time is 1 to 4 digits. It can be followed by `A` (AM), `P` (PM), `N` (noon) or `M` (midnight); these are case-insensitive. One- and two-digit numbers are interpreted as hours; three- and four-digit numbers as hours and minutes. For three-digit numbers, the first digit is the hour (`0` – `9`). If no letters follow the digits, 24-hour time is used. You can also use "`at` *hh*`:`*mm*", "`at` *h*`:`*mm*", "`at` *h*`am`", "`at` *h*`pm`", "`at noon`" and "`at midnight`".

Day is either a month name followed by a day number, or a day of the week. If the word `week` follows, the command is invoked seven days later. Standard abbreviations are recognised. For example:

```
at 8am jan 24
at 1530 fr week
```

`at` commands depend on the execution of `/usr/lib/atrun` by `cron`. The time resolution ("granularity") of `at` depends on the frequency of execution of `atrun` (a default system executes `atrun` every 15 minutes).

Standard output (1) or error output (2) is lost unless it is redirected.

The `at` and `batch` commands write the job number to standard error.

Options:

`-r` Removes scheduled jobs. Parameter is the job number. Only the superuser can remove another's jobs.

`-l` Lists all job numbers submitted by the user. To see all jobs currently scheduled, use

```
ls -l /usr/spool/at
```

In this directory, there are files named `yy.ddd.hhhh.*` which are scheduled jobs, `lasttimedone`, containing the last *hhhh* at which `at` executed, and `past`, containing activities in progress.

Printing

The printing system depends on the `lpd` daemon. Ensure that it is started by `rc` and do nothing further with it. Low-level printer control and configuration is covered in *Devices / Printers*. Here I will deal with the act of printing and high-level control.

The `lpr` command – print files

Syntax:

`lpr [` *options* `] [` *file…* `]`

Useful options:

`-h`	No banner page
`-P`*printer*	Send to *printer*
`-w`*n*	Page width of *n* characters
`-z`*n*	Page length of *n* lines
`-x`	Assume the files do not require filtering before printing

Typical command:

`lpr -hxP`*printer file*

The `lprm` command – remove jobs from printer queue

Syntax:

`lprm [ -P`*printer* `] [ – ] [` *job…* `] [` *user…* `]`

Without any arguments, `lprm` deletes the currently active job if it owned by the user. If the – flag is specified, all jobs that a user owns are removed. If the superuser uses –, the whole spool queue is emptied. If a username is specified, `lprm` attempts to remove jobs belonging to that user (only useful to the superuser). A job may be removed by number (obtain the number from `lpq`, see below). The command announces the name of each file it removes: if it says nothing, it did nothing. It kills and restarts daemons as necessary.

The `lpq` command – examine spool queue

Syntax:

`lpq [` *options* `] [` *job…* `] [` *user* `]`

With no arguments, `lpq` reports on any jobs in the default queue.
Job ordering is FIFO (first in, first out).

Options:

`+`*n*	Scans and displays the queue every *n* seconds (default 30) until the queue is empty.
`-l`	Displays the status of each job on more than one line if necessary.
`-P`*printer*	Specifies a printer. Otherwise the `PRINTER` environment variable is used, or `lp`.

The `lpstat` command – printer status information

Syntax:

`lpstat [` ***options*** **`]`**

Options:

`-a` [*printers*]	Are printers accepting requests?
`-d`	Print name of default system printer
`-o` [*printers*]	Status of print requests
`-p` [*printers*]	Status of printers
`-r`	Status of the line printer daemon, `lpd`
`-s`	Status summary
`-t`	All status information
`-u` [*users*]	Status of users' print requests

Note all options that take a list of arguments want a comma-separated list. If you include spaces between items, you must put the list in quotes.

The `lpc` command – line printer control

Syntax:

`/etc/lpc [` ***command*** **[** ***argument…*** **]]**

The line printer system is controlled by the superuser using `lpc`. Without any arguments, an interactive mode is entered; use `?` to list commands and `help` *command* for further information on a command.

Using tape drives

Remember the difference between `/dev/rmt*` and `/dev/nrmt*`: the latter doesn't rewind after a program closes the device.

Magnetic tape manipulation: `mt`

Syntax:

`mt [ -f` ***tapedevice*** **`]`** ***command*** **`[`** ***count*** **`]`**

This command performs *command*, *count* times (default 1) on device *tapedevice* (default is the `TAPE` environment variable or `/dev/nrmt0h`). Important commands are as follows:

`bsf`	Backspace *count* files
`fsf`	Forward-space *count* files
`offline, rewoffl`	Rewind the tape and place the unit off-line
`retension`	Retensions the tape (move tape one complete pass between the end and the beginning)
`rewind`	Rewinds the tape
`status`	Prints status information

Examples:

```
mt -f /dev/rmt01 rewind
mt -f /dev/nrmt1h fsf 3
```

Backing up data: `dump`

Syntax:

`/etc/dump [` ***key*** **[** ***argument…*** **]** ***filesystem*** **]**

`dump` copies all files changed after a certain date from a specified *filesystem* to a file/pipe/tape/disk. The *key* specifies the date and other options. `dump` requires operator attention in situations where an end-of-tape occurs, when `dump` ends or when an unrecoverable read error occurs. `dump` can write to all users in the "`operator`" group when it needs attention, and talks to its user at the control terminal. It gives progress reports and asks yes/no questions when it has problems. Nevertheless, it is feasible to run `dump` as part of an automated backup, and a script is given here to do so.

Options (if none are given, `9u` is assumed):

`0-9`	**Specifies the dump level. All file modified since the last date given in /etc/dumpdates, for this filesystem, for lesser levels, will be dumped. If no date is found, all files are dumped: thus level 0 causes a full dump.**
`B`	Specify size of dump medium, in kilobytes.
`d`	Specify tape density (bits per inch).
`f`	**Place the dump on the file/device specified by the next argument. If the name of the file is –, `dump` writes to standard output. Default device is `/dev/rmt0h`.**
`n`	Notifies all users in the group `operator` when `dump` needs attention.
`S`	Prints output file size in bytes, or number of volumes for devices.
`s`	Specify tape size in feet
`u`	**Writes the date of the beginning of the dump to `/etc/dumpdates` if the dump is successful. The format of `/etc/dumpdates` is one (free format) record per line: filesystem name, dump level and ctime format (see `ctime`(3) for details). It is possible to edit this file if you are**

superuser and careful.

`W` Tells the operator which file systems need to be dumped (taken from `/etc/dumpdates` and `/etc/fstab`). All other options are ignored; `dump` exits immediately.

`w` Lists only those filesystems that need to be dumped.

Examples:

`dump 9Bf 400 /dev/rra2a /dev/ra0a`
Dumps the filesystem `/dev/ra0a` to RX50 diskettes.

`dump 0undf 6250 /dev/rmt?h /usr/users`
Dumps the filesystem `/usr/users` to a 6250bpi tape on a TU78 tape drive.

`dump 0Sf test /`
Reports the number of bytes to be dumped for a level 0 dump of the root filesystem. Note: the file `test` is *not* made.

`dump -0uf /dev/nrmt1h /usr`
Dumps the entire `/usr` filesystem to a 8Gb DAT drive on `/dev/nrmt1h`.

Here is a complete automated script to backup a computer running Oracle databases. The script is run in the middle of the night and backs up to an 8Gb DAT drive (so it should never run out of space as the drives aren't that big). Note that dumping the filesystem `/` does not dump all files! Each physical filesystem must be listed. Note also that the non-rewind device is used – if the rewind device were to be used, each dumped filesystem would overwrite the previous!

```
#! /bin/sh
trap 'echo "*** backup: aborted on" `date`; exit 1' 1 2 3 15
flag=
ps -auxww | egrep 'dbclose|nrmt1h' | grep -v grep && flag=Y
if test $flag
then
  echo "*** backup: BACKUP FAILED. SCRIPT ALREADY ACTIVE OR DEVICE IN USE!"
else
  echo "*** backup: Script started. Closing Oracle: " `date`
       su – oracle -c /usr/users/oracle/bin/dbclose
  echo "*** backup: Attempt to close Oracle finished. Beginning backup."
       cd /
       mt -f /dev/nrmt1h rewind
       dump -0uf /dev/nrmt1h /
       dump -0uf /dev/nrmt1h /usr
       dump -0uf /dev/nrmt1h /var
       dump -0uf /dev/nrmt1h /database1
       dump -0uf /dev/nrmt1h /database2
       dump -0uf /dev/nrmt1h /database3
       mt -f /dev/nrmt1h rewind
  echo "*** backup: finished on " `date`
  echo "            Starting Oracle."
       su – oracle -c /usr/users/oracle/bin/dbstart
  echo "*** backup: Oracle started. Script terminating."
fi
```

This script is owned by `root` and run with a `crontab` entry whose command is `/backup >>/backup.log 2>&1`. As it is run by `cron`, it executes with `root` authority (see *Cron* above). Standard error is redirected to standard output (`2>&1`) so both are appended to `/backup.log`.

Restoring data: `restore`

Syntax:

`/etc/restore` ***key*** **[** ***name…*** **]**

`restore` reads files created by `dump`.

The `f` key can be used to specify a device or image file other than `/dev/rmt0h`. Arguments other than keys and their modifiers are file and directory names to be restored. Unless the `h` key is specified, a directory name refers recursively to all files and directories within it.

Important keys:

`i`	**Interactive restore. Highly recommended. The command `help` gives a summary of available commands.**
`f`	**As for `dump`.**
`h`	**Extracts actual directories, not the files that they reference. Prevents heirarchical restoration of complete subtrees.**
`v`	**Verbose. Causes `restore` to tell you what it's doing.**
`r`	Extract files into current directory. (A `restoresymtab` file is created to transfer information between incremental restores. Remove this when you've finished.)
`t`	List the names of the specified files if they exist on the dump media. If no name is given, the root directory is listed (so if the `h` flag isn't given, the whole contents is listed).
`x`	Extracts files specified.

Examples:

1. Here's how to restore an entire filesystem to a new disk from the default tape:

   ```
   /etc/newfs /dev/rra0g ra60
   /etc/mount /dev/ra0g /mnt
   cd /mnt
   restore r
   ```

   A further `restore` can be done to get an incremental dump back.

2. Using `dump` and `restore` in pipeline to transfer a file system:

   ```
   dump 0f – /usr | (cd /mnt; restore xf –)
   ```

3. Restoring the `/database1` filesystem interactively from a dump produced by the backup script given earlier:

   ```
   mt -f /dev/nrmt1h rewind
   mt -f /dev/nrmt1h fsf 3
   cd /database1
   restore -ivf /dev/nrmt1h
   ```

   (followed by `add` and `extract` commands within `restore`)

Archive manipulation: dd, cpio, tar

dd

The dd(1) command copies data from one place to another while performing some conversion (record size, ASCII to EBDIC, that sort of thing).

cpio

The cpio(1) command – related to ar(1) but better – is a filter designed to let you copy files to or from an archive.

Syntax:

cpio -i [-C] [*keys*] [*patterns*]
cpio -o [*keys*]
cpio -p [*keys*] *directory*

Options:

`-i` Copies files that match the specified pattern. Otherwise copies all files. Extracts files from standard input (which is assumed to be the product of a previous `cpio -o`) and places them in the user's current directory tree. For files with the same name, new replaces old unless `-u` is used.

Only files that match *patterns* are selected. Multiple patterns may be specified. Default pattern is `*`.

`-C` Old-style compatibility option.

`-o` Copies out the specified files. Reads standard input to obtain a list of files, copies them to standard output together with path name and status information.

`-p` Copies files into specified destination directory, which must exist. Reads standard input to obtain a list of path names of files that are conditionally created. This list of files is copied into the destination directory tree. For files with the same name, new replaces old unless `-u` is used.

Keys:

Key	Valid for	Description
6		UNIX Sixth Edition format.
a	-o, -p	Retains original access times of input files.
B		Block I/O with 5,120 bytes per record. Only meaningful when directing I/O to/from /dev/rmt?h or /dev/rmt?l.
b		Swaps both bytes and half words.
c	-i, -o	Creates header information in ASCII format.
d		Creates subdirectories as needed below the destination directory.
f		Copies all files *except* those that match the specified pattern.
k	-i, -o, -p	Enables symbolic link handling.
l		Creates links wherever possible.
m		Retains modification times.
r	-i	Interactively renames files. If you enter a null line, file is skipped.
s		Swaps bytes while copying files in.
S		Swaps half words while copying files in.
t		Prints table of contents of the input.
u		Copies files unconditionally.
v		Verbose.

Examples:

1. Copy the contents of a user's directory into an archive:

```
ls | cpio -o > /dev/rmt0l
```

2. Duplicate a directory heirarchy:

```
mkdir ~phares/newdir
cd ~phares/olddir
find . -print | cpio -pdl ~phares/newdir
```

3. Copy all files and directories with names containing "`chapter`" into `smith`'s home directory and underlying directories:

```
find ~smith -name '*chapter*' -print | cpio -o > /dev/rmt0h
```

`tar`

The `tar`(1) ("tape archiver") command saves and restores multiple files to and from a single archive. Tar files are popular on the Internet, particularly on FTP servers.

Syntax:

`tar [` ***key*** **`] [`** ***name…*** **`]`**

Options:

`c`	Create new archive
`r`	Write named files to the end of the archive. (`r` for write!?)
`t`	List the names of files
`u`	Add named files if they didn't exist or have changed
`x`	Extract named files (default: all files)
`0-9`	Substitute the number for the device unit number in `/dev/rmt?h`. Default `0`.
`C`	Used to perform a directory change prior to archiving
`H`	Help
`V`	Display extended verbose information
`d`	Use `/dev/rra1a` as the default device – though the `mdtar`(1) command is recommended for use with floppy disks.
`f`	Use the next argument as the name of the archive. If the name is –, use standard input/output. Here is an example that moves the directory *fromdir* to the directory *todir*: `cd fromdir; tar cf – . | (cd todir; tar xpf -)`
`h`	"Save a copy of the actual file on the output device under the symbolic link name, instead of placing the symbolic information on the output."
`i`	Ignore checksum errors
`l`	Complain if links to the files dumped cannot be resolved. (Default: no errors printed.)
`m`	Don't restore modification times
`o`	Don't put owner/mode of directories into the archive
`p`	Restore the named files to their original modes
`v`	Write name of each file treated (preceded by function letter) to diagnostic output
`w`	Print the action to be taken, followed by the filename, then wait for user confirmation (a word beginning with '`y`').

Example:

To archive files from `/usr/include` and `/etc` to the default output tape, type

```
tar c -C /usr/include . -C /etc .
```

Networking

Introduction

UNIX is an operating system that lends itself well to networking – the Internet grew up on UNIX. It is important to have at least a vague idea of the layered approach to networking. At the highest level is the **application**: imagine this saying "I want to get a file from machine X". At the next level down is the **service**: a service is something like "getting files – FTP" or "logging in to another machine – Telnet" or "resolving network addresses – ARP". A service is *bound* to an underlying **protocol**, such as IP (Internet protocol) or TCP (transmission control protocol). Protocols goven the movement of data from a service on one computer to a service on another: they wrap around the packet of data, saying things like "make way: message from machine Z to machine X". Note that the word "protocol" is often used in a looser sense to mean an agreed system for communication (its true meaning) – therefore some services call themselves protocols. At the lowest level is the **hardware**: the wires, network cards and telephone lines carrying signals.

TCP/IP: addressing

The Internet runs on the TCP/IP procotol. In this, each machine on the network – by which we mean the whole Internet, worldwide – has a unique **IP address**. This is a four-byte address[4], usually written with dots (.) between the numbers: 179.140.2.200. In order to bring some order to this, organizations are assigned addresses depending on their size, dividing the address into networks. A class A network is for *big* organizations: the first byte designates the network and the other three designate addresses within the network. The organization's machines have numbers like 153.*xx*.*xx*.*xx*: they have 256^3 addresses available. A class B network has 256^2 possible addresses within it: the first two bytes designate the network. A class C address has 256 addresses within it: the first three bytes designate the network.

Class A networks have a first byte in the range 0 – 126; class B networks have a first byte in the range 128 – 191 (and a second byte in the range 1 – 254); class C networks have a first byte in the range 192 – 223 (and a second byte in the range 0 – 255, and a third byte in the range 1 – 254). Network 127 is reserved for the local loopback address (see below). Avoid numbering hosts so that their host fields contain all ones (or, for compatibility with older systems, all zeros) – this will conflict with network broadcasts (see below).

The network address is assigned by the Network Information Center (NIC) in the United States[5]. Systems on a network need to know what portion of the four-byte IP address is the NIC address (the network portion) and which is for local machines (the subnet or host address). Therefore a **netmask** is assigned. This has binary ones in the network fields and binary zeros in the subnet address fields. Therefore a class C site has a netmask of 255.255.255.0.

Netmasks are complicated by **subnet routing**. If you have a class B network, for example, you may want several subnets (mapping to different pieces of cable). Hosts outside the network do not need to know whether it is using subnetworks: all routing is transparently handled inside the network. You might choose to use the whole of the third byte as a subnet address; therefore your netmask would be 255.255.255.0. However, you might want to use only the first three bits of the third byte for subnet routing (giving you up to eight subnets); then your netmask will be 255.255.224. Even class C networks can have subnets in this manner. However, splitting bytes in this way makes things complicated for humans, who like decimal arithmetic!

[4] Although it seems unlikely that the world has 256^4 (four billion) computers on the Internet, the subdivision of IP address bytes into network and subnet fields wastes many numbers, and addresses are running out. An eight-byte Internet addressing scheme is proposed (giving up to $1{\cdot}8 \times 10^{19}$ possible addresses).

[5] DDN Network Information Center, SRI International, Room EJ291, 333 Ravenswood Avenue, Menlo Park, CA 94025, United States of America. Telephone (800) 235-3155 or (415) 859-3695. E-mail: nic@nic.ddn.mil.

If you do not use subnet routing, your netmask will be 255.0.0.0 (class A), 255.255.0.0 (class B) or 255.255.255.0 (class C). If the netmask is anything else, subnet routing is in use; however, a netmask of 255.255.255.0 might be a class C network with no subnets or a class B network using 8 bits for subnet routing. Valid decimal values for the host (non-NIC assigned) fields of the network mask are 255 (eight subnet bits), 254 (seven), 252 (six), 248 (five), 240 (four), 224 (three), 192 (two), 128 (one) and 0 (zero).

The Internet Protocol has a system whereby messages can be sent to all hosts on a network. This is called broadcasting. One address is assigned to be the **broadcast address** – it is the same for all hosts on the network. The broadcast address is the NIC address followed by *either* all ones *or* all zeros, according to local convention. **All ones is the standard for broadcast addresses.** Therefore a class A network (NIC address 15) has a broadcast address of 15.255.255.255; a class C network (NIC address 158.8.62) has a broadcast address of 158.8.62.255.

It is wise not to use 0 or 255 as any part of your address fields. It's not worth the risk of conflict with systems that use these number for special things. You may lose a couple of addresses, but it'll save hassle. Use addresses and network numbers 1 – 254. Similarly, avoid the network 127.*x*.*x*.*x*: this is used for loopback testing.

LANs and beyond: address resolution, routing and complex services

In a real WAN, other systems need to exist too, and I will summarise them. First, there is a system for mapping Internet address to Ethernet addresses – Address Resolution Protocol (ARP). The reverse of this is logically called RARP, and is often more useful. (An Ethernet address is a six-byte number built into the Ethernet hardware, and manufacturers guarantee to supply unique Ethernet addresses in their interfaces.)

Then, systems must exist to route packets of information travelling from one IP address to another over the physical network structure. Whenever a packet travels from one piece of Ethernet cable to another, or from Ethernet to fibre-optic, or from a T1 to a telephone line, the machine attached to both must **route** the packet. Routing also involves making intelligent decisions about the fastest route to take: if two systems are linked by a fibre-optic and a telephone link, the router should pick the foptic!

There are many other protocols that exist on the Internet – time synchronisation and SNMP, for example – but I won't go into detail now.

For those of you interconnecting UNIX and PC networks, it is vital to be aware that UNIX systems use the **Ethernet II** frame type. (The frame level is one I didn't mention, lying in between the protocol and the hardware.)

Internet addresses for humans

The numerical IP addresses are not useful for humans. Therefore there is a separate textual naming system that is mapped to the underlying IP address. A system is named `machinename.domain` (and a user of that system will be `user@machinename.domain`). The domain is composed of a heirarchy of names, separated by full stops, of the form `organization.type.country`. The country field is a two-letter country code (`uk`, `il`); addresses in the USA have no country code. The type field is `co` (commercial), `ac` (academic)[6], `gov` (government), `mil` (military) and so on. So you get domains like `cam.ac.uk` (Cambridge University), `harvard.edu` (Harvard University) and `demon.co.uk` (Demon Internet Ltd). A fully specified machine names is `skcmis.demon.co.uk`.

Each domain should have a **name server** that supplies IP addresses in return for names. I will not describe this process here.

[6] In the USA, academia has the type `edu`.

Configuring UNIX

The simple way: using `netsetup`

First, you must know the IP address of the computer together with the IP broadcast address and your netmask. Then log in as `root` and run `netsetup install`. You will be asked:

1. to verify your system's name
2. to supply your network address. This is the NIC-assigned network address (class A, B or C), *without* subnets. So if you are installing the host 179.140.254.200 on a class B network with subnet routing, enter `179.140`.
3. whether you are using subnet routing
4. for your host address (`254.200` in this example[7] – note that if you are using subnet routing, you must enter the subnet number as part of the host number)
5. for the number of bits to use for subnet routing, should you be using it
6. whether to use zeros or ones for the IP broadcast address – use all ones
7. for the device name and unit nuber of your network interface (typically `ln0` for a Lance Ethernet interface)
8. for a network name for your network address, and any aliases for it
9. for the host name, abbreviations, network address and host address for each host on the network

If you specify the `install` parameter to `netsetup`, all previous network configurations are overwritten.

Essential files

The hosts database, `/etc/hosts`

This is a list of all known hosts (computers). Each line should begin with the full Internet address and continue with the name of the host, followed by any aliases. Comments are preceded by a hash (#). Begin each entry on a new line.

In addition to the 'real' hosts, there is usually an entry for `localhost`. This gives the "Internet address" of the interface used for internal loopback testing and local communications (usually the loop network interface, `lo0`). By default this address is 127.0.0.1. If our hypothetical system is called `julia`, its default hosts file would read

```
#
# Host Database
#
127.0.0.1 localhost
174.140.254.200 julia randomalias
```

The networks database, `/etc/networks`

Just like `/etc/hosts`, the `networks` database is a list of all known networks. This time the fields are name, number, aliases. The loopback network (as we have just seen, default 127) is called "`loop`", alias "`loopback`"; the Ethernet network is known as "`ethernet`" unless you bother to enter a name. Therefore the default `networks` database for our example machine would read

```
#
# Internet networks
#
loop 127 loopback
ethernet 179.140
```

[7] The manuals *System and Network Setup* (page 2–13) and `netsetup`(8) contradict each other here!

The trusted hosts database, `/etc/hosts.equiv`

This database contains a list of hosts that are 'trusted'. When an `rlogin` or `rsh` request is received from a host listed in this file, no further validity checking is performed (passwords are not requested). When a remote user is in the `hosts.equiv` file, that user is defined to be equivalent to a local user with the same user ID.

The file is a list of names, as in:

```
host1
-host2
+@group1
~@group2
```

The file is scanned sequentially and the scan stops when the requesting host is found. A line consisting of a hostname gives trust to anyone logging in from that host; if preceded by a minus (–) all users from that host are not trusted. A line consisting just of + means that all hosts are trusted – **this is very dangerous**. The `+@` and `–@` syntax is specific to the Yellow Pages (YP) service and give and revoke trust to and from groups of hosts (as served by YP).

User-by-user checking: `.rhosts`

The `hosts.equiv` file has the same format as `.rhosts`. When a user executed `rlogin` or `rsh`, the `.rhosts` file for that user (on the receiving machine, obviously) is appended to the `hosts.equiv` file for checking. **If a user is excluded by a minus entry from `hosts.equiv` but included in `.rhosts`, that user *is* trusted. If the user is `root`, only `.rhosts` is checked** (in this case, `/.rhosts`). This has nasty security implications!

To avoid security problems, the `.rhosts` file must be owned by either the user on the receiving machine, or root, and it may not be a symbolic link. (The danger this statement hints at is that a user may log in and add an entry to someone's `.rhosts` file, such as `root`'s!)

You can put two entries on one line, separated by a space. If the remote host is equivalenced by the first entry, the user named by the second is allowed to supply any name to the `-l` option (see *Client Programs / Rlogin*) as long as it exists in `/etc/passwd`. To give you the most dangerous example, suppose the machine `discovery` has the following line in `/.rhosts` (recall attempts on the `root` user only check `/.rhosts`, not `/etc/hosts.equiv`):

```
hubble rudolf
```

Then `rudolf` can log on to `discovery` from `hubble` as `root` without supplying a password. If the entry were in `/etc/hosts.equiv` instead, `rudolf` could log on to `discovery` from `hubble` as any user except `root` without a password.

Interface configuration: `ifconfig`

Every network interface must be initialized with `/etc/ifconfig`. Normally, the command to do so lives in `/etc/rc.local` and the interfaces are initialized at boot time. The syntax is as follows:

`/etc/ifconfig` ***interface*** **[** ***address*** **[** ***dest_address*** **]] [** ***parameters*** **]**

The *interface* parameter is a name and unit number, for example `ln0`. The address is either a host name present in `/etc/hosts`, or an Internet address (*xx.xx.xx.xx*).

The following parameters are of interest:

`up`	Marks an interface up.

`down`	Marks an interface down. Transmission is not attempted.
`arp`	Ethernet devices use ARP to map between Ethernet and Internet addresses. This is the default.
`-arp`	Don't do this.
`netmask`	**Sets the network mask.**
`dstaddr`	Specifies the correspondent on the other end of a point-to-point link.
`broadcast`	**Specifies the address for broadcasts to the network.**

An example of an `ifconfig` command is shown below, under *Routers / Setting up a router*.

The Internet daemon, `inetd`

This daemon handles most of the Internet service functions. When it starts, it reads a configuration file (by default `/etc/inetd.conf`) and opens a socket (see *Processes / Theory / Sockets*) for each specified service. When it receives a connection on a stream socket or a packet on a datagram socket it invokes the server specified in the configuration file to handle the request. The configuration file is reread whenever `inetd` recieves a `HANGUP` signal (see *Processes / Sending signals to processes*).

The Internet daemon configuration database, `/etc/inetd.conf`

For each service to be handled by `inetd`, a single line should exist in `/etc/inetd.conf` giving the following information:

service name	(must be in `/etc/services`, see below)
socket type	(`stream` or `dgram`)
protocol name	(must be in `/etc/protocols`, see below)
delay	(`wait` or `nowait`)
program name	(the server program's name, fully specified)
arguments	(up to five arguments for the server)

Servers marked as '`wait`' must be able to handle all requests that come to it during its lifetime; `inetd` will not invoke any new instances while there is one running. If marked as '`nowait`', a new invocation of the server will be started for every incoming request.

Here are some lines from a typical `/etc/inetd.conf`:

```
ftp           stream tcp    nowait /usr/etc/ftpd              ftpd
telnet        stream tcp    nowait /etc/telnetd         telnetd
talk          dgram  udp    wait   /etc/talkd           talkd
bootp         dgram  udp    wait   /usr/etc/bootpd      bootpd -i
```

The protocol database, `/etc/protocols`

This file lists the known protocols used on the Internet. For each protocol, a single line should contain the following information:

official protocol name
protocol number
aliases

Here is a typical protocol database:

```
#
# @(#)protocols 6.1 (ULTRIX) 11/19/91
# Internet (IP) protocols
#
ip     0      IP     # internet protocol, pseudo protocol number
icmp   1      ICMP   # internet control message protocol
ggp    3      GGP    # gateway-gateway protocol
tcp    6      TCP    # transmission control protocol
pup    12     PUP    # PARC universal packet protocol
```

```
udp    17     UDP    # user datagram protocol
```

The service database, `/etc/services`

This file lists the known services used in the Internet. Each line should contain:

official service name
port-number/protocol-name
aliases

Examples are best: here is part of our /etc/services file:

```
#      @(#)services       6.1 (ULTRIX) 11/19/91
#      services           1.16 (Berkeley) 86/04/20
#
# Network services, Internet style
#
echo          7/tcp
echo          7/udp
discard       9/tcp        sink null
discard       9/udp        sink null
…
ftp           21/tcp
telnet        23/tcp
…
whois         43/tcp       nicname
…
bootp         67/udp                          # boot program server
…
#
# UNIX specific services
#
exec          512/tcp
biff          512/udp      comsat
login         513/tcp
…
```

Routers

Routers (also known as gateways) are hosts that are connected to multiple LANs. There is a network interface for each LAN, and each interface has a unique host name and Internet address. Routers can transfer data between the LANs to which they are attached.

Setting up a router

Add the following line to `/etc/rc.local`:

`/etc/ifconfig` *device-name-number pseudohostname* `broadcast` *a.b.c.d* `netmask` *w.x.y.z*

where *device-name-number* is the new Ethernet interface and *pseudohostname* is the new host name that the router will be known as on the new network.

Then enable the `routed` daemon by placing its command in `rc.local`. Normally you will only have to uncomment out the lines:

```
# if [ -f /etc/routed ]; then
#       /etc/routed & echo -n ‘ routed’   >/dev/console
# fi
```

Edit the `/etc/networks` file to include the name, number and alias of the additional network.

Reboot to invoke the `routed` daemon.

Accessing a router

On every machine that needs to access the router…

1. Edit `/etc/networks` to include the details of the additional network you wish to access through the router.
2. Enable the `routed` daemon in `/etc/rc.local`.
3. Reboot (or invoke the `routed` daemon manually).

NFS – the Network File System

NFS enables users to share files over the network. A client computer can mount or unmount file systems and directories from an NFS server. Once mounted, a remote file system or directory can be used just as a local file system. Typically, a client mounts one or more remote file systems or directories at boot time, if entries exist in the `/etc/fstab` file.

Four programs implement the NFS service: `portmap`, `mountd`, `biod` and `nfsd`. A client's mount request is transmitted to the remote server's `mountd` daemon after obtaining its address from `portmap`. A port mapper is a Remote Procedure Call (RPC) daemon that maps RPC program numbers of network services to their User Datagram Protocol (UDP) or Transmission Control Protocol (TCP) protocol port numbers. The `mountd` daemon checks the access permission of the client and returns a pointer to the file system or directory. Subsequent access to that mount point and below goes through the pointer to the server's NFS daemon (`nfsd`) using remote procedure calls. Some file access requests (write-behind and read-ahead) are handled by the block I/O daemons (`biod`) on the client.

If you are using NFS, you may use the NFS locking service: this supports file and file region advisory locking on local and remote systems, important when several users or processes access the same file simultaneously. (Advisory locks are not enforced.)

Automatic NFS setup

1. Read the sections on server and client setup, so you understand the concepts involved.
2. Run `/etc/nfssetup` and answer its questions. It's quite self-explanatory.

Setting up an NFS server

1. Mark file systems and directories for export – `/etc/exports`

This file defines the NFS file systems and directories that can be exported by a server (compare `/etc/fstab`, which tells a client which remote NFS drives to mount). Only local file systems and directories can be exported, and a full pathname must be given. For example, to export the file system `/usr/bin/oodle` to the world, with no special permissions, add the following entry:

```
/usr/src/oodle
```

To export the same file system to a client named `endeavour` only, use this:

```
/usr/src/oodle endeavour
```

Obviously, to deny NFS access to a file system, remove or comment out its entry in `/etc/exports`. (Comments begin with a hash – # – as always.)

When you have modified `/etc/exports`, run **`showmount -e`**. This makes the changes take effect immediately; otherwise the file will be checked when next `mountd` receives a request.

The full syntax of the export file is:

pathname [-r=#] [-o] [*identifier_1 identifier_2 ...*]

where

pathname	Name of a mounted local file system, or a directory of a mounted local file system. The pathname must begin in column 1.
-r=#	Map client superuser access to user ID #. If you want client superusers to access the file system or directory with the same permissions as a local superuser, use -r=0. The default is -r=-2, which maps a client superuser to nobody, thus limiting access to world-readable files
-o	Export file system or directory read-only.
identifiers	Host names (or netgroups under Yellow Pages, or both) separated by white space, that specify the access list for this export. If no access list is specified, anyone can access it.

There can only be one entry per file system or directory exported.

Each file system that you want to export must be explicitly defined – exporting root (/) will not allow clients to mount /usr, since it is a separate file system.

Export permissions are not "inherited": if you export /usr and /usr/local, /usr/local has a completely separate set of export attributes. So /usr could be read-only while /usr/local is read-write. Mount access is checked against the closest exported ancestor: let me give an example. System black exports /usr with read-write permission (to everyone) and /usr/local/bin with read-only permission (to everyone). If system blue mounts /usr/local/bin/random, it has read-only permission to it. If it mounts /usr/local/bin, it has read-only permission to it. If it mounts /usr/local/etc, it has read-write permission to it. If it mounts /usr/local, it has read-write permission to it – and thus to /usr/local/bin! Obviously, this is a strange example, but instructive.

2. Load the NFS server daemons from /etc/rc.local

You need to run portmap, mountd and the NFS daemon. Be sure to add the NFS daemons to the rc.local file *after* and Yellow Pages entries, or if you have none, *after* the following entry:

```
/etc/ifconfig lo0 localhost
```

Add this lot:

```
# %NFSSTART%
# echo -n 'NFS daemons'                          >/dev/console
if [ -f /etc/portmap ]; then
    /etc/portmap; echo -n ' portmap' > /dev/console
fi
if [ -f /etc/mountd -a -f /etc/portmap -a -s /etc/exports ]; then
    /etc/mountd -i; echo -n ' mountd -i' > /dev/console
fi
if [ -f /etc/nfsd -a -f /etc/portmap ]; then
    /etc/nfsd 4 & echo -n ' nfsd' > /dev/console
fi
# %NFSEND%
```

Note that the "4" parameter to nfsd starts 4 NFS daemons. This is a typical number but depends on your system's load.

If you are going to run the NFS locking service, add this lot just before the # %NFSEND% entry:

```
# %NFSLOCKSTART%
echo 'Enabling NFS Locking'                      >/dev/console
```

```
      [ -f /usr/etc/nfssetlock ] && {
          /usr/etc/nfssetlock on & echo 'nfs locking enabled'
>/dev/console
      }
      [ -f /usr/etc/statd ] && {
          /usr/etc/statd & echo -n 'statd '               > /dev/console
      }
      [ -f /usr/etc/lockd ] && {
          /usr/etc/lockd & echo 'lockd'                   > /dev/console
      }
      # %NFSLOCKEND%
```

3. Reboot the system

```
      /etc/shutdown -r now
```

Notes

If you are running in multiuser mode and you want to start NFS-related commands and daemons, run these:

```
      /etc/portmap
      /etc/mountd
      /etc/nfsd 4 &
      /usr/etc/rwalld &
      /etc/biod 4 &
```

Do not run `nfssetlock`, `statd` or `lockd` while in multiuser mode. The transition from kernel-based to daemon-based locking can lose locking information.

Setting up an NFS client

1. Load NFS client daemons from `/etc/rc.local`

NFS clients must run the `portmap` and `biod` daemons. If you want your system to be notified when an NFS server is going down, you must run the `rwalld` daemon. If you are enabling NFS locking, you need an entry for `nfssetlock`. By default NFS locking is disabled.

Add the following to `/etc/rc.local`:

```
      # %NFSSTART%
      if [ -f /etc/portmap ]; then
          /etc/portmap; echo -n ' portmap'   > /dev/console
      fi
      if [ -f /etc/biod ]; then
          /etc/biod 4 & echo -n ' biod'             > /dev/console
      fi
```

Optionally, `rwalld` (which must come after `portmap`):

```
      if [ -f /usr/etc/rwalld -a -f /etc/portmap ]; then
          /usr/etc/rwalld & echo -n ' rwalld'> /dev/console
      fi
```

Optionally, the NFS locking service:

```
      # %NFSLOCKSTART%
      echo 'Enabling NFS Locking'                   >/dev/console
      [ -f /usr/etc/nfssetlock ] && {
          /usr/etc/nfssetlock on & echo 'nfs locking enabled'
>/dev/console
```

```
}
[ -f /usr/etc/statd ] && {
    /usr/etc/statd & echo -n 'statd '              > /dev/console
}
[ -f /usr/etc/lockd ] && {
    /usr/etc/lockd & echo 'lockd'                  > /dev/console
}
# %NFSLOCKEND%
```

Finally:

```
# %NFSEND%
```

2. Mount file systems using `/etc/fstab`

Use `/etc/fstab` to mount file systems you always want to have mounted. For the syntax, see *The UNIX File System: `/etc/fstab`*. The NFS-specific options are:

`bg`	**Background. If the first mount fails, retry the mount in the background the number of times specified (default 10,000).**
`grpid`	All files or directories created in the file system being mounted are created so that they inherit their parent's ID regardless of the setting of the *gid* bit in their parent's directory. In the absence of this option, all files or directories created in the file system inherit the group ID of the running process.
`hard`	**Retry the NFS operation (not the mount, the access) request until the server responds. This option applies after the mount has succeeded. Use this when mounting read-write file systems.**
`intr`	Allow hard mounted file system operations to be interrupted.
`nintr`	Disallow the above.
`noexec`	Binaries (executable files) cannot be executed from this file system.
`nosuid`	The `setuid` and `setgid` programs cannot be executed from this file system.
`pgthresh=`*n*	Set the paging threshold in kilobytes.
`port=`*n*	Set server IP port number to *n*.
`retrans=`*n*	Set the number of NFS operation retransmission to *n*. This applies after the mount has succeeded.
`retry=`*n*	Set number of mount failure retries to *n*.
`ro`	**Read-only**
`rsize=`*n*	Set read buffer size to *n* bytes.
`rw`	**Read-write**
`soft`	Return an error if the server doesn't respond to an NFS operation (not mount). Do not use for read-write file systems.
`timeo=`*n*	Set NFS timeout to *n* tenths of a second.
`wsize=`*n*	Set write buffer size to *n* bytes.

Defaults:

```
rw,hard,intr,retry=10000,timeo=11,retrans=4, \
port=NFS_PORT,pgthresh=64
```

For further options relating to how quickly a client sees updates to a file or directory that has been modified by a host, see `mount`(8nfs).

Example:

```
/usr/src@spice:/spice/usr/src:ro:0:0:nfs:bg
```

Be sure to use the `bg` option: otherwise the client may fail to reboot if a server is unavailable.

Manually mounting and dismounting remote file systems

The syntax for mounting NFS file systems is

`/etc/mount -t nfs [ -f -r -v ] [ -o` ***options*** **`]`** ***device directory***

where

`-r`	Mount read-only.
`-v`	Verbose: reports what happened.
`-o` *options*	See above

device	Either
	`host:remote_name`
	or
	`remote_name@host`

directory	Local mount point (usual rules apply).

To mount the example given above, type:

```
mount -t nfs -o bg,ro spice:/usr/src /spice/usr/src
```

The syntax for unmounting NFS file systems is

`/etc/umount [ -f -v ]` ***directory***

where

`-f`	Fast unmount: the client unmounts the file system or directory without notifying the server. This can avoid the delay of waiting for acknowledgement from a server that is down.
`-v`	Verbose: reports what happened.

Do not use the `nfs_mount` or `nfs_umount` commands; work with `mount` and `umount`. The one exception is the use of `nfs_umount -b` in the `/etc/rc.local` file of client systems: this broadcasts a message informing NFS servers that the machine no longer has any NFS filesystems mounted. It is done in case the machine had crashed while it had NFS filesystems mounted, and allows the servers to "clean up". This command is automatically added by the `nfssetup` command.

NFS security

Superuser access over the network

For increased security, don't allow client systems superuser access to exported file systems.

This may cause problems with mailing to `root@client` if the client's `/usr/spool/mail` is a remotely mounted file system. If the server has a `root` mailbox (`/usr/spool/mail/root`), the client will receive notification of such messages but won't be able to read them. If there is no such mailbox on the server, one will be created with an ownership of `-2` (`nobody`). This allows all `root` users that import the file system to read the `root` mailbox on the server. To work around this, create the aliases "`root`" and "`admin`" for the normal username of the system's administrator: this avoids using the real `root` mailbox.

Port monitoring

Only privileged users can attach to some Internet ports ("privileged ports"). NFS doesn't normally check that a client is coming in through one of these. If you activate this checking, you ensure that file access requests were generated by the client kernel and not forged by an application program. To activate NFS server port checking, type

```
/etc/nfsportmon on
```

Guess how you turn it off.

Increasing security by not acting as an NFS server

I hope that's self-explanatory.

Limiting the client systems that are allowed

And that.

Troubleshooting

Some file operations fail

Tough. Not all locking operations work through NFS, even with NFS locking enabled. In the worst case, data written to a file in append mode can be lost if other processes are writing to the same file. Moral: don't use NFS to operate on files that are likely to be modified by several users simultaneously.

Clock differences

The timestamps of a file are determined by the server. If the server's and client's clock are more than an hour out, you may have problems with applications that depend on both local time and the create/modify/access time attributes of files.

Network or server failures

Hard-mounted file systems retry indefinitely; soft-mounted file systems return an error. Hard-mounting is the default if you didn't specify soft mounting when you mounted. If a process is blocked for a long time, NFS prints this to the console and error logger:

```
NFS server hostname not responding, still trying
```

A failed operation on a soft-mounted system results in this error:

```
NFS operation failed for server hostname, Timed out
```

Run through these checks:

1. Check the server is up and running. If the server is called `yellow`, type this from the client:

   ```
   /etc/rpcinfo -p yellow
   ```

   If the server is up, you should see something like this:

   ```
   program vers proto   port
     100004    2   udp   1027  ypserv
     100004    2   tcp   1024  ypserv
     100004    1   udp   1027  ypserv
      100004    1   tcp   1024  ypserv
      100007    2   tcp   1025  ypbind
      100007    2   udp   1035  ypbind
      100007    1   tcp   1025  ypbind
      100007    1   udp   1035  ypbind
      100003    2   udp   2049  nfs
      100005    1   udp   1091  mountd
   ```

2. If you got that list, use `rpcinfo` to check if the `mountd` server is running. For the above example, type

```
/etc/rpcinfo -u yellow 100005 1
```

   If `mountd` is running, you should get

```
program 100005 version 1 ready and waiting
```

3. If these two `rpcinfo` commands fail, try this:
   Log into the server. Is it running properly? Ensure `/etc/portmap`, `/etc/mountd` and `/etc/nfsd` are running.
   If it's running, check the Internet connections.
   Check the Ethernet connections of server and host.

Remember you don't need `biod` or any NFS server daemons running to be an NFS client.

Process blocking in client programs

This could be because

… the server is down. Restart it.
… the `nfsd` daemon is malfunctioning. Kill it and run `nfsd` again.
… two or more processes are deadlocked. Kill one of them.

System hangs part way through boot

You probably forgot the background (`bg`) option on one of your NFS mounts, and the server is down.

Slow remote file access

If no `biod` daemons are running on the client, start some. Here's how to start four:

```
/etc/biod 4 &
```

Check your Ethernet connection. Typing `netstat -i` will show you if packets are being dropped. Typing `netstat -c` shows you how much retransmission is occurring: 0.5% retransmission is high: bad interface, bad connection. If you suspect a bad Ethernet board, type `netstat -s` to see if any UDP packets have been dropped (this may occur because of bad checksums, in turn causing NFS operations to time out). Many bad transmissions (`badxid` > 0.1%) indicates the timeout in the mount operation is too small. Increasing the timeout may not only fix this, but improve performance.

How a typical network starts

Do not attempt to learn about individual daemons from this: it is merely a list of things to expect.

- As we have said (see *How UNIX starts*) `rc` calls `rc.local`. This runs `ifconfig` to configure the Ethernet interface. The `ln0` device is the first Lance Ethernet interface; `lo0` is a software loopback device for testing and so forth. The primary network interface should be first. Typical commands are of the form:
  ```
  /etc/ifconfig ln0 '/bin/hostname' broadcast 179.140.255.255 netmask 255.255.0.0
  /etc/ifconfig lo0 localhost
  ```
- If the machine is a router, `/etc/routed` is loaded.
- If NFS is being used, the NFS daemons (`portmap`, `mountd`, `nfsd`, `biod`, `rwalld`) and, optionally, the NFS locking daemons (`statd`, `lockd`) are loaded
- If the machine is sending mail, `/etc/sendmail` is loaded.
- If LAT is being used, `/etc/lcp` is started.
- When `rc.local` exits, control returns to `rc`. Here, `/etc/inetd` is loaded. This controls all the Internet services (`telnetd`, `ftpd`, `fingerd` etc.).
- If SNMP is to be used, `rc` loads `/etc/snmpd`.

Remote booting – the bootp protocol

The Internet BOOTP protocol is a UDP-based protocol that allows diskless machines to find out their Internet addresses, the address of a bootserver, and the name of a file to boot.

The `bootpd` server is either started from `/etc/rc.local`, or by `inetd` (preferable). In the latter case an entry must be made in `/etc/inetd.conf`, and the `-i` flag supplied.

Options:

`-d` Logs all requests and responses

`-i` Use this (in `inetd.conf`) if `bootpd` is started by `inetd`. For an example of an `inetd.conf` entry, see *The Internet daemon configuration database* above.

The `bootpd` server reads its configuration file, `/etc/bootptab`, when it starts. When a request arrives, `bootpd` checks to see if the configuration file has been modified, and read it again if it needs to. The `/etc/bootptab` file has the format:

```
#
# /etc/bootptab: database for bootp server (/usr/etc/bootpd)
#
# home directory

/usr/local/bootfiles

# default bootfile

defaultboot

# end of first section

%%

# Now we have one line per client.
# If a bootfile cannot be found, "bootfile.host" is also tried.
#
#
# host       htype haddr         iaddr        bootfile
#                  (Ethernet)   (Internet)

hostx        1 02:60:8c:06:35:05 99.44.0.65   ultrix
hosty        1 02:07:01:00:30:02 99.44.0.03   vms

# The htype is always 1 (Ethernet). The haddr field can use
# a period (.), hyphen (-) or colon (:) as separators.
# The bootfile entry is used if the client does not know the name
# of the file it wants to boot.
```

MOP file retrieval – `mop_mom`

Here is a summary from the manual:

The `/etc/mop_mom` program listens for download (or dump upload) requests and spawns `/usr/lib/dnet/mop_dumpload` to process them.

Normally, `mop_mom` is started from `rc.local`. A client system can request a file by name; if it does not, `mop_mom` searches its node database for a file to offer. The node database is that administered by `addnode`(8) (*q.v.*). It is the DECnet node database. If `mop_mom` comes up with no absolute path, it searches `/usr/lib/mop`; if it can't find the file there it searches `/usr/lib/dnet`. ("Files in `/usr/lib/dnet` must be in lower-case with an extension.") Otherwise the filename is interpreted literally.

If no filename is given, or the `LOADUMP_SECURE` environment variable is set, the Ethernet address of the requesting machine is looked up in the nodes database. Setting the `LOADUMP_SECURE` environment variable (using `setenv LOADUMP_SECURE on` at the command line, or permanently by loading `mop_mon` with the command `LOADUMP_SECURE=on /etc/mop_mon`).

Essentially, `mop_mon` is useless and you should use `bootp` (part of the TCP/IP protocol suite).

Some important client programs for users and administrators

Most of these programs need a daemon to be loaded from `inetd.conf`; I will list them with the name of the program.

`ftp` (requires `ftpd`)

File Transfer Protocol: how to move files between computers. Launch FTP in the following way:

`ftp [` *options* `] [` *hostname* `]`

The following options are valid:

`-d`	Debugging
`-g`	Disables filename expansion
`-i`	Disables interactive prompting during multiple file transfers
`-n`	Disables autologin during the initial connection. If autologin is enabled, FTP checks the local user's `.netrc` file for an entry describing an account on the remote machine. If no entry exists, FTP uses the local user's name as the default user name on the remote machine, and prompts for a password.
`-v`	Displays all statistics and responses from the remote server

Format of the `.netrc` file, since I mentioned it:

This file lives in the user's home directory. Each line defines options for a specific machine, or defines defaults. The "default" line must be the first if it is present. Fields are separated by spaces or tabs. A default line has the following format:

`default` *default-machine-name*

A machine line has the following format:

`machine` *machinename options*

Valid options are:

`login`	Login name
`password`	Password
`account`	Additional password
`macdef`	Defines a macro (like the FTP `macdef` command). A blank line must follow the macro lines to terminate macro definition.

Here is an example of a `.netrc` file:

```
machine cactus login smith
machine nic.ddn.mil login anonymous password anonymous
machine palm.stateu.edu login smith password uonrelcome
macdef byenow
quit
```

If you put passwords in a `.netrc` file, don't give the file world read permission!

The following commands can be used at the `ftp>` prompt:

`?` [*command*]	**Synonym for `help`.**
`!`	**Invokes a local shell.**
`$` *macroname args*	Invokes a macro.
`account` [*password*]	Supplies a supplemental password (if none is given, you will be asked for it).
`append` *localfile* [*remotefile*]	Appends *localfile* to a file on the remote system (by default, of the same name).
`ascii`	Sets file transfer type to network ASCII. The default.
`bell`	Beeps after each command is completed.
`binary`	**Sets file transfer type to support binary image transfer. Make sure you use this before transferring programs!**
`bye`	**Quits.**
`case`	Toggles conversion of remote filenames to lower-case during mget commands. Off by default.
`cd` *remotedirectory*	**Changes remote directory.**
`cdup`	**Moves up one directory level on the remote machine.**
`close`	Closes FTP session.
`cr`	Toggles CR stripping during ASCII file retrieval (default on).
`debug` *debugvalue*	Sets debug level.
`delete` *remotefile*	**Deletes *remotefile*.**
`dir` [*remotedir* [*localfile*]]	**Catalogues *remotedir* (or current remote directory); optionally, places the output in *localfile*.**
`disconnect`	Same as `close`.
`form` *format*	Sets file transfer format to *format*. Default is `file`.
`get` *remotefile* [*localfile*]	**Gets *remotefile*; calls it *localfile*.**
`glob`	Toggles wildcard expansion for the multiple-file commands. Default on.
`hash`	**Toggles the printing of a hash (#) for every data block (1024 bytes) transferred.**
`lcd` [*directory*]	**Change local directory.**
`ls` [*remotedir* [*localfile*]]	**Like `dir`, but shorter output.**
`macdef` *macroname*	Defines a macro. Blank line ends. Use `\` to quote characters literally. Use `$` for argument substitution (`$1`, `$2`, etc.). Use `$i` to have the macro loop: the macro is executed once for each argument (which is substituted for `$i`).
`mdelete` *remotefiles*	**Deletes *remotefiles*. If globbing is enabled, the filenames are first expanded with `ls`.**
`mdir` *remotefiles localfile*	**Obtains a directory of *remotefiles* and places it in *localfile*.**
`mget` *remotefiles*	**Gets *remotefiles*.**
`mkdir` *directory*	**Makes *directory* on the remote machine.**
`mode` *modename*	Sets file transfer mode (default is `stream`).
`mput` *localfiles*	**Puts *localfiles* onto the remote machine.**
`nmap` [*inpattern outpattern*]	Allows mapping of filenames; useful for systems with different filename conventions. See `ftp`(1c) for details.
`ntrans` [*inchars* [*outchars*]]	Filename character translation mechanism. See `ftp`(1c) for details.
`open` *host* [*port*]	**Opens a connection to *host*.**
`prompt`	**Toggles interactive prompting during multiple file transfers. Off by default.**

`proxy` *ftp-command*	Executes an FTP command on a secondary control connection. This allows you to transfer files between two FTP servers by opening connections to both of them. The first command should be a `proxy open`. Type `proxy ?` to see the commands you can execute on the secondary connection. The following commands are different under `proxy`: • `open` doesn't define new macros during autologin • `close` doesn't erase macro definitions • `get` and `mget` transfer files from the primary to the secondary connection • `put`, `mput` and `append` transfer files from the secondary to the primary connection Third-party file transfer depends on the secondary computer's support for the FTP `PASV` command.
`put` ***localfile* [*remotefile*]**	**Puts *localfile* onto the remote machine [as *remotefile*].**
`pwd`	**Prints remote working directory.**
`quit`	**Synonym for `bye`.**
`quote` *arg1 arg2 …*	Sends data verbatim to the remote FTP server.
`recv`	Synonym for `get`.
`remotehelp` [*command*]	Requests help from remote FTP server.
`rename` ***from to***	**Remote rename.**
`reset`	Clears the reply queue.
`rmdir` ***directoryname***	**Removes remote directory**
`runique`	**Toggles the storing of files on the local system with unique filenames (appending `.1`, `.2` etc. if the file exists, reporting the new name, aborting with an error if `.99` is exceeded). Default: off.**
`send`	Synonym for `put`.
`sendport`	Toggles the use of `PORT` commands.
`status`	**Shows current status.**
`struct` *structname*	Sets file transfer structure to *structname*. Default: `file`.
`sunique`	Toggles the storing of files on the remote system with unique filenames (see `runique`). The remote computer must support the `STOU` command. Default: off.
`tenex`	Sets the required file transfer type for TENEX machines.
`trace`	Toggles packet tracing.
`type` [*typename*]	Displays or sets the file transfer type, which is network ASCII by default.
`user` ***username* [*password* [*account*]]**	**Identifies you to the remote FTP server. Usually useful after an `open` command has opened a connection but failed to authorise you.**
`verbose`	Toggles verbose mode. On by default.

Abort file transfers by pressing `^C`. Gets take longer to interrupt than puts, for obvious reasons.

Using filenames for I/O redirection.

1. A filename of "`-`" represents standard input or output.
2. If a filename begins with "`|`", the remainder of the filename is interpreted as a shell command, and that command's standard input or output is used as appropriate. If the command includes spaces, the whole filename (including `|`) must be enclosed in quotes (`""`). A useful example is "`dir |more`".

Excluding users.

Users named in `/etc/ftpusers` are prevented from transferring files by the `ftpd` daemon.

`ping` (administrative)

The analogy is from submarines and sonar (to "ping" a ship is to fire a sonar pulse at it and see what comes back). Ping sends packets to a computer and checks that they come back. It's the most basic test that a machine is up and connected to the Internet. You can control the ping process in detail, but the useful syntax is:

`ping` ***hostname***

You will either get the response "*`host`* `alive`" or – after a delay – "`no answer from` *`host`*".

`telnet` (requires `telnetd`)

This is the usual command to log into a remote computer. While you can run `telnet` on its own and use a command-line interface, the normal syntax is:

`telnet` ***hostname*****`[:`*****`port`*****`]`**

Usually, you do not need to specify a port: a TELNET server is looked for at the default port. Some machines run several TELNET services, so you can specify a port number. Typically, a computer provides a normal TELNET facility for its users on the default port, and a MUD or other time-wasting activity on a port that you have to access by number!

`finger` (requires `fingerd`)

Finger is not exclusively a network command. `finger` by itself prints the login name, full name, terminal, idle/login times, office location and phone number for every user logged on. The syntax `finger` *user* displays more detailed information about a particular user (including home directory, login shell, their `.plan` file if it exists, and the first line of their `.project` file if *it* exists). The syntax `finger user@host` displays this same information for a remote user.

`rlogin` (requires `rlogind`)

Logs in to another UNIX machine. For details of security and authorization, see *The Trusted Hosts Database* above. The full syntax for `rlogin` is:

`rlogin` ***rhost*** **`[-e`***`c`***`] [-8] [-L] [-l`** ***username*****`]`**

The options are as follows:

`-e`*`c`*	Uses *c* as the escape character, instead of the tilde (~).
`-8`	Allows an 8-bit data path at all times.
`-L`	Runs session in litout mode.
`-l` *username*	Logs in as *username*, not as your current user.

Your remote terminal type is the same as your local terminal type (set by the `TERM` environment variable); `^S` and `^Q` provide flow control as normal. Assuming the escape character is the usual tilde (~), the sequence `~.` on a new line disconnects from the remote host. A tilde followed by `^Z` suspends the session.

`rsh` (requires `rshd`)

Syntax:

`rsh` ***host*** **`[-l`** ***username*****`] [-n]`** ***command***

The remote shell connects to the remote host (as the specified user or the local user if none is given) and executes the specified command, copying the command's standard input/output/error from or to standard input/output/error. You cannot specify a password with a command (so the equivalence system described in *The Trusted Hosts Database* above must be used). The `-n` option redirects all

command input to `/dev/null`; you must use this if you run `rsh` as a background task from `csh` and do not desire input to the command (failure leads to a blocked `csh`).

Shell metacharacters that are not quoted are interpreted on the local machine; those that are quoted on the remote machine. Therefore, note the difference between the following:

```
rsh otherhost cat remotefile >> localfile
rsh otherhost cat remotefile ″>>″ otherremotefile
```

The standard host names[8] for local machines are also commands in `/usr/hosts`, so if you put this directory on your search path you can omit "`rsh`".

Do not use `rsh` for interactive commands; use `rlogin` or `telnet` instead.

Stop signals only stop the local `rsh` process.

`netstat` (administrative)

There are four formats for this command.

`netstat` [`-Aan`] [`-f` *address_family*] [*system*] [*core*]

This displays a list of active sockets for each protocol.

`netstat` [`-himnrs`][`-f` *address_family*] [*system*] [*core*]

This presents the contents of one of the other network data structures according to the option selected.

`netstat` [`-n`] [`-I` *interface*] *interval* [*system*] [*core*]

Given an interval, this form continuously displays packet traffic information on the configured network interfaces.

`netstat` [`-I` *interface* `-s`] [*system*] [*core*]

This form provides statistics for network interfaces.

For detailed syntax and options, see `netstat`(1). One useful command is **`netstat -i`**, which displays status information for autoconfigured interfaces.

`ruptime` (administrative) (requires `rwhod`)

(Note that `rwhod` is normally loaded from `inetd.conf`, but is usually disabled by default to keep network traffic down.)

`ruptime` is like `uptime`, but remote.

[8] The standard name is the first name listed for a host in `/etc/hosts`; any others are nicknames.

Rebuilding the kernel

I am not going to discuss modifications to the kernel at a source-code level. However, there are situations where kernel parameters need to be changed; at these times you must rebuild the kernel.

Editing the configuration file

I am assuming that you are using a RISC MIPS processor running ULTRIX and that the machine is called `hubble`. For such a machine, the kernel configuration file would be `/sys/conf/mips/HUBBLE`.

Here is a typical configuration file:

Global definitions

`ident`	`"HUBBLE"`	Defines the host name (in upper case)
`machine`	`mips`	Defines the hardware
`cpu`	`"DS5100"`	Defines the processor
`maxusers`	`64`	The maximum number of simultaneously active users allowed on the system. Make the number greater than or equal to the number in your license agreement.
`processors`	`1`	The number of processes in the system
`maxuprc`	`50`	The maximum number of processes one user can run simultaneously (default 50).
`physmem`	`64`	An estimate of the amount of physical memory, in megabytes. It does not limit the amount of memory used, but it is used to calculate the system page table size. Make it greater than or equal to the amount of RAM.
`timezone`	`0 dst 3`	Number of hours west of Greenwich Mean Time (negative indicates east). The `dst` parameter indicates daylight savings time; it is followed by a number requesting a particular DST correction algorithm. The values are USA 1 (default), Australia 2, Western Europe 3, Central Europe 4, Eastern Europe 5.
`smmax`	`1024`	Defines the maximum number of pages of virtual memory at which a shared memory segment may be sized. VAX pages are 512 bytes; RISC pages are 4096 bytes. Defaults are 256 and 32 respectively, giving 128 kilobytes in either case.
`smseg`	`8`	The maximum number of shared memory segments per process (defaul 6).
`scs_sysid`	`1`	"Identifies a host uniquely on the CI star cluster to the SCS subsystem. Default 1."

Other definitions, not in use on this system, are:

`maxuva` *num*	Maximum aggregate size of user virtual memory, in megabytes, default 256. Doesn't apply to RISC processors.
`bufcache` *percent*	Percentage of physical memory to be allocated as file system buffer cache ($10 \leq$ *percent* < 100).
`swapfrag` *num*	When a process requires additional swap space, it is granted *number* 512-byte blocks each time. Minimum 16; default 64; must be a power of two.
`maxtsiz` *num*	Largest text segment in megabytes. (VAX default 12, RISC default 32.)
`maxdsiz` *num*	Largest data segment in megabytes (default 32).
`maxssiz` *num*	Largest stack segment in megabytes (default 32).
`smmin` *num*	The minimum number of pages of virtual memory at which a shared memory segment may be sized. VAX pages are 512 bytes; RISC pages are 4096 bytes. Default 0.
`smsmat` *num*	The highest attachable address for shared memory

	segments, in megabytes. VAX default `MAXDSIZE`, RISC default 0 (no check is made).
`smbrk` *num*	The default spacing between the end of a private data space of a process and the beginning of its shared data space, in pages of virtual memory (VAX pages are 512 bytes, RISC pages are 4096 bytes). The VAX default is 64 (32K); the RISC default is 10 (40K). This value is important, because once a process attaches shared memory, private data cannot grow beyond the beginning of shared data.

Options definitions

`options`	`QUOTA`	Allows disk quotas to be set.
`options`	`INET`	Provides Internet communication protocols. The `inet` pseudodevice must also be listed in the Pseudodevice Definitions section.
`options`	`NFS`	Enables support for the NFS protocol. This requires (1) that you also set the RPC option; (2) that you list the `nfs` pseudodevice in the Pseudodevice Definitions section.
`options`	`RPC`	Allows RPC-based applications. It is required when the NFS option is specified. The `rpc` pseudodevice must also be set.
`options`	`DLI`	Allows the `mop_mom` program to be active (see the *Networking* section of this guide). The `dli` pseudodevice must also be set.
`options`	`UFS`	Enables the standard, local file system. If you do not use NFS, you must use UFS. Without this parameter, the system will be considered diskless. The `ufs` pseudodevice must be set.
`options`	`NETMAN`	?
`options`	`LAT`	Enables LAT support. List the `lta` and `lat` pseudodevices in the Pseudodevice Definitions section.
`options`	`PACKETFILTER`	?
`options`	`AUDIT`	Loads the audit subsystem. To specify the base size of the audit buffer in bytes, use `AUDIT =` *number*. The default is 16K.
`options`	`SYS_TPATH`	Enables the trusted path mechanism. The `sys_tpath` pseudodevicemust also be set.

Other definitions, not in use on this system, are:

`EMULFLT`	Enables emulation of the floating point instruction set if it is not present in hardware. Don't delete this option!
`FULLDUMPS`	Enables full dump support.
`DECNET`	Enables DECnet support. The `decnet` pseudodevice must be set.
`SYS_TRACE`	Enables the system call tracing capability. The `sys_trace` pseudodevice must be set.
`SMP`	Enables symmetric multiprocessor capability. Don't use with a single processor (performance will suffer).

The makeoptions definitions for RISC processors

`makeoptions`	`ENDIAN="-EL"`	Put this line in. There's no choice.

System image definitions

```
config          vmunix  root on rz0a  swap on rz0g    dumps on rz0g
```

The general format for this line is:

`config` *filename configuration-clauses*

The *filename* argument is the name to be given to the compiled kernel, by default `vmunix`. The *configuration-clauses* define the root file system, pagin/swapping space and crash dump space. Keywords are as follows:

`root` [`on`] *device*
: Specifies the device for the root file system. For diskless clients, use "`root on ln0`".

`swap` [`on`] *device* [`and` *device*] [`size` *x*] [`boot`]
: The first *device* specifies the device/partition for a paging and swapping area. The second *device* allows you to specify another, so swapping will be interleaved. The `size` clause can be used to specify a non-standard partition size for one or more swap areas (*x* is in 512-byte sectors).

  If you specify `swap on boot`, the `a` partition of the booted device becomes the root, and swap space is assumed to be the `b` partition of the same device.

`dumps` [`on`] *device*
: Specifies the partition and device where crash dumps are to be stores. The device must be on the same controller as the boot device. The default dump device is the first swap device.

Device definitions

```
adapter         ibus0      at nexus?
controller      sii0       at ibus?     vector sii_intr
disk            rz0        at sii0      drive 0
disk            rz1        at sii0      drive 1
disk            rz2        at sii0      drive 2
disk            rz3        at sii0      drive 3
disk            rz4        at sii0      drive 4
disk            rz5        at sii0      drive 5
disk            rz6        at sii0      drive 6
disk            rz7        at sii0      drive 7
tape            tz0        at sii0      drive 0
tape            tz1        at sii0      drive 1
tape            tz2        at sii0      drive 2
tape            tz3        at sii0      drive 3
tape            tz4        at sii0      drive 4
tape            tz5        at sii0      drive 5
tape            tz6        at sii0      drive 6
tape            tz7        at sii0      drive 7
device          ln0        at ibus?     vector lnintr
device          mdc0       at ibus?     vector mdcintr
```

These are all quite hardware-specific (meaning "check your manual"). The question marks ask the system to calculate the correct address. I hope that SCSI tapes and disks, at least, are obvious.

Pseudodevice definitions

A pseudodevice is an operating system component for which there is no associated hardware. Each line has the following format:

`pseudo-device` *name* [*num*]

Our sample system uses the following:

`pseudo-device`	`nfs`	Network File System protocol support.
`pseudo-device`	`rpc`	Remote Procedure Call facility.
`pseudo-device`	`dli`	DLI support of `mop_mom` activity.
`pseudo-device`	`pty`	Pseudoterminal support. Default 32. Specify *num* in increments of 16 if you need more than 32 pseudoterminals (i.e. if you want 58, say 64).
`pseudo-device`	`loop`	Network loopback interface.
`pseudo-device`	`ether`	10Mb/s Ethernet
`pseudo-device`	`ufs`	Local file system support.

`pseudo-device`	`netman`	?
`pseudo-device`	`inet`	DARPA Internet protocols.
`pseudo-device`	`lat`	Local area terminal (LAT) protocols. If you list this, you must also list `lta`.
`pseudo-device`	`lta 64`	Pseudoterminal driver. Default 16. Specify *num* in increments of 16, as for `pty`. If you list this, you must also list `lat`.
`pseudo-device`	`scsnet`	Systems Communications Services (SCS) network interface driver.
`pseudo-device`	`msdup`	?
`pseudo-device`	`packetfilter`	?
`pseudo-device`	`sys_tpath`	Trusted path support.
`pseudo-device`	`audit`	Audit support. "Provides the generation of the file `'hostname'/audit.h`, which causes the appropriate files to be rebuilt when a new system is generated."

Other pseudodevices, not in use here, are:

`presto`	Kernel support for ULTRIX Prestoserve on the DS5500.
`decnet`	DECnet support.
`sys_trace`	Support of the system call trace capability.
`bsc`	Support of 2780/3780 emulation (VAX only).

Generating the kernel and activating it

The easiest way to do this is to run `/etc/doconfig`. The recommended procedure is as follows:

1. Save the running `vmunix` as `vmunix.old`.
2. Move `/genvmunix` to `/vmunix`.
3. Reboot the system to single user mode. (If it comes up in multi-user mode, `kill -TERM 1` will return to single user mode; see *Terminals* above).
4. Check file systems.
5. Mount the `/usr` file system.
6. Run the `doconfig` program. (When execution is complete, make a note of the message `doconfig` prints showing the path and location of the new `vmunix`.)
7. Move `/vmunix` to `/genvmunix`.
8. Copy the new `vmunix` (from the message noted above) to `/vmunix`.
9. Reboot the system.

This procedure ensures that you are running the generic kernel: this is recommended for the recognition of new hardware. In practice, I have never had problems simply performing steps 1, 6, 8 and 9.

The process of running `doconfig` is slightly dangerous, as it is possible to destroy your existing confiiguration file. (`doconfig` also has the function of generating a specific kernel configuration file from the generic one, and has a tendency to copy the generic configuration file on top of the one you just spent ten minutes editing.) Proceed as follows:

1. Enter your system name, in lower case, when asked.
2. **Say no to "A system with that name already exists. Replace it?".**
3. Say no to "Do you want to edit the configuration file?" unless you do.
4. Your kernel will be made.

Software subsets

The software that makes up UNIX is organized into *subsets* to make system management easier. The `setld` command is used to manage software subsets.

Syntax:

```
/etc/setld [ -D root-path ] -l location [ subset… ]
/etc/setld [ -D root-path ] -d subset…
/etc/setld [ -D root-path ] -i [ subset… ]
/etc/setld [ -D root-path ] -v subset…
/etc/setld [ -D root-path ] -c subset message
/etc/setld [ -D root-path ] -x location [ subset… ]
```

The options are

`-D`	Specify *root-path* as the root directory for an operation. The default is `/` for all operations except `-x`, when the default is the current directory. This option allows you to operate on off-line mounted systems (e.g. plugging a disk into your system and working with that).
`-l`	Load software from the distribution mounted on *location*. If a subset is not named, a menu of available subsets is presented.
`-d`	Delete subset(s) from the system. All files in the subset which have not been modified since installation are unlinked (deleted). Some subsets are marked undeletable to avoid nasty happenings. If a subset is required by other subsets, you will be warned.
`-i`	Inventory the system or any specified subset.
`-v`	Verify the subset.
`-c`	Configure the subset, passing *message* to the subset control program.
`-x`	Extract subsets from the distribution media mounted on *location*. If a subset is not specified, a menu is presented.

The *location* can be a device (e.g. `/dev/rmt0h`), a directory (e.g. `/mnt/RISC/software`) or a remote machine (e.g. `hostname:`).

Shells and shell scripts

What is a shell?

The shell is the program that accepts commands from a terminal and executes them. A shell is normally run when a user logs in; which shell is run depends on that user's entry in `/etc/passwd` (*q.v.*). "Standard" UNIX comes with two shells: **`sh`** (the standard or Bourne shell, also known as `sh5` as it derives from UNIX System V) and **`csh`** (the C shell, which has C-like syntax). Many other shells have been written; most follow the general syntax of either `sh` (such as the Korn shell, `ksh`) or `csh` and add facilities such as command histories. In addition, most shells have a version called a *restricted shell*, usually prefixed with "`r`". (The restricted Bourne shell is called `rsh5`, not `rsh`, because `rsh` already stands for "remote shell".) Restricted shells limit the commands a user can execute.

I will deal mainly with `sh` here, with a short section on the main differences in `csh`.

Simple and background commands

The `sh` prompt is `$`. The `csh` prompt is `csh>`. The superuser prompt in either shell is #.

The basic syntax of all commands is

```
command arg1 arg2 arg3 …
```

When the shell sees this, it first checks to see if `command` is an internal shell command (such as `cd`)[9]. If it is, it executes the command. If not, it tries to find the file `command`. If `command` includes an absolute pathname (e.g. `/bin/ls`, `./myprog`) the shell tries to run that file. If not (e.g. `ls`, `myprog`), then it searches the path (see below). Assuming it finds `command` and has authority to execute it, it creates a new process and runs `command` in it, passing the arguments to `command`. (More details are to be found under *Processes* above.)

If you append an ampersand (`&`) to the command, the shell does not wait for the command to complete but reports its process number and returns you to the prompt. This is called **background execution**. The `wait` command waits for all background processes to complete.

Standard input, output, error. Redirection and pipes.

If you are a C programmer, you will be familiar with standard input (`stdin`), output (`stdout`) and error (`stderr`). The idea is that all I/O to a program goes via *file channels*. These can be channels to files on disk, or to terminals and other special devices. All programs are supplied with three channels when they start: 0 (`stdin`), 1 (`stdout`) and 2 (`stderr`). Normally, a program reads from `stdin`, writes to `stdout` and prints urgent error messages to `stderr`. Generally, all three point to the controlling terminal, so you can type input to the program and see its output.

However, all of these channels can be **redirected** to different files or devices. The syntax for a typical redirected command is

```
command arguments… <infile >outfile 2>>errorfile
```

This command will receive standard input from `infile`, be it a normal file or a device special file. Standard output will go to `outfile`, overwriting it if it exists. If no channel numbers are given, you see, standard input or output is assumed. However, to redirect standard error we must specify its

[9] Shells create a new process to execute external commands; a process includes a working directory and changing directory in a sub-process of the shell doesn't affect the shell itself. Therefore the `cd` command must be executed within the shell process.

channel number (2). You could use `1>outfile` or `0<infile` if you wanted to, of course. The `>>` notation tells the shell to *append* output to `errorfile` rather than overwriting it.

Sometimes it is useful to send standard error to the same place as standard output. You can do this by making a **copy** of the channel. The following appends standard error and standard output to `logfile`:

```
command arguments... >>logfile 2>&1
```

The `&1` means "a copy of channel 1". Very occasionally, you might need to swap channels; for example, you can swap standard output and standard error using "`3>&1 1>&2 2>&3`" – using a temporary channel 3. You can use "`<&-`" or "`>&-`" to close a file descriptor (channel).

You can also redirect I/O to *processes* as well as files and devices. This uses a mechanism called a **pipe** (discussed more fully under *Processes*). It is a way of attaching the standard output channel of one command to the standard input channel of another, transferring data between the two processes directly. Here is a typical example, piping the output of `ls -al` to `grep` in order to search for lines beginning with a `d` (thus finding all directories), and piping that to `more` so that the output comes to the terminal one page at a time:

```
ls -al | grep '^d' | more
```

Finally, if you want to send output to a terminal *and* a file, pipe it to **`tee`**(1). `tee` has the syntax

```
tee [ -i ] [ -a ] [ file... ]
```

and copies its standard input to standard output and also *file(s)*. The `-a` option causes it to append to *file(s)*; the `-i` option causes it to ignore interrupts.

Paths and environment variables

Just like DOS, UNIX has environment variables. These are text strings that programs can use to obtain information; for example, Oracle looks at environment variables to find its utility program files. Some environment variables are particularly significant because the shell uses them. These are:

`$MAIL`	Before the shell issues a prompt, it checks this variable. If the file it refers to has been modified, the shell prints "`you have mail`" before prompting for the next command. This variable is usually set in the `.profile` file (see below) in the user's login directory.
`$HOME`	The default argument for the `cd` command. Usually the user's home directory, set in the login profile.
`$PATH`	The search path. Each time a command is executed, this list of directories is searched for an executable file with the name of the command. If `$PATH` isn't set then the current directory, `/bin` and `/usr/bin` are searched by default[10]. Otherwise `$PATH` consists of directory names separated by `:`. For example, `PATH=:/usr/bin:/bin:/usr/ucb` specifies that the current directory should be searched first (the null string before the first `:`), then the listed directories.

[10] That's what the manual says. I dispute this; I think the shell searches nothing, not even the current directory, if `$PATH` is empty. In this situation every command must have a fully-specified pathname, like `./myprog` or `/usr/ucb/vi`.

	If the command contains a `/` then the path is not used; the command name is treated as a fully-specified pathname.
`$PS1`	The primary shell prompt (default "`$ `").
`$PS2`	The secondary shell prompt, used when further input is required (default "`> `").
`$IFS`	The set of characters used for *blank interpretation* (by default, blank, tab and newline). Security note: this can be used for hacking by the knowledgeable: say a user has no access to the shell, but (a) can copy files, (b) can set `$IFS`, and (c) knows that one of his scripts runs a program called `pine`. If he copies `/bin/sh` to `./pi` and then sets `$IFS` to "`n`", the script will run his shell called `pi`. Voilà, shell access. I've seen it done…

To set an environment variable, use the following method

```
PATH=/bin:/usr/ucb:. fredvar=value null= thingy=oodle
```

To read the value of a variable, prefix it with `$`; for example,

```
echo $thingy
```

will `echo oodle`. You can also enclose the variable name in braces (`{}`). For example,

```
tmp=/tmp/ps
ps -aux >${tmp}a
```

will direct the output of `ps` to `/tmp/psa`. If you missed out the braces the shell would look for a variable called `tmpa` (and in this case, fail).

The following variables have special values within the shell:

`$?`	The exit status (return code) of the last command executed, in the form of a decimal string. The convention is for a zero exit status to represent success.
`$#`	The number of positional parameters (arguments).
`$$`	The process number of this shell, in decimal. Often used to generate temporary files, since process numbers are unique. As in `ps -aux > /tmp/ps$$; … ; rm /tmp/ps$$`.
`$!`	The process number of the last process run in the background.
`$-`	The current shell flags (such as `-x` and `-v`), set on shell invocation or by `set`.

To pass all positional parameters, unevaluated, use `$@`. Positional parameters are `$1`, `$2`, …

Shell scripts

A shell script is the UNIX equivalent of a DOS batch file. It is a text file that contains a list of shell commands, and when a shell tries to execute a shell script – assuming the file's flags allow the user to

execute it – a new shell process is spawned to parse the script. Note that while an executable binary only needs its execute flag set to be run, a shell script must also be *readable* by the shell.

There is a way to specify which shell should be used to execute a script.[11] If the first character of the script is a # – that is, the script starts with a comment – the script is run by `/bin/csh`. If the script starts with "`#! shellname`", the script is executed using the shell `shellname`. If it begins "`#! shellname arguments`", "`shellname arguments`" is executed but not the rest of the script, which isn't very useful.

The notation

```
. scriptname
```

allows a file to be executed *as if it were being typed in* – that is, no new shell process is created. If the script changes directory, for example, the calling shell will have its working directory changed. The script file need have no execute flags set, merely a read flag.

The sh command language in brief

Quoting

Characters with special meaning (such as `< > * ? | &`) can be "escaped" with `\`. Thus `echo \?` prints a single question mark. Single quotes quote everything inside them (except for `'`). Double quotes quote everything inside them except for `$ ` \ "`.

Command separation

The semicolon (`;`) can be used to separate commands. The value returned by a command list is that returned by the last simple command in the list.

`for`

Syntax:

`for` *name* [**`in`** *w1 w2 …*]
`do` *command-list*
`done`

If the `in` clause is omitted, `in $*` is assumed (all the arguments to the command, in order).

Creating files

Aside from the `touch` command (see *Other Handy Commands*), you can use

`>` *file*

to make sure *file* exists and is empty.

`case`

Example:

```
case $# in
      1)    cat >>$1 ;;
      2)    cat >>$2 <$1 ;;
      *)    echo 'usage: append [from] to' ;;
esac
```

The *pattern* before the `)` has the usual pattern syntax (`*` for any character(s) including none, `?` for any single characer, `[…]` for any of the enclosed characters).

[11] I couldn't find the formal definition of this system in the manual, so I have described it based on experimentation.

"Here documents"

Example:

```
for i
do grep $i <<!
        …
        fred mh0123
        bert mh0789
        …
!
done
```

The shell takes the lines between `<<!` and `!` as the standard input for `grep`. The `!` is arbitrary; you can use any string to start and end the "document". Parameters are substituted before the input is made available to `grep`.

`test`

This isn't actually in the shell. It performs a test and returns a zero exit status for "true" and non-zero for "false". There are several options – see `test`(1) for them – but a few useful ones are

`test` ***s***	True if *s* is not a null string
`test -f` ***file***	True if *file* exists
`test -r` ***file***	True if *file* is readable
`test -w` ***file***	True if *file* is writable
`test -d` ***file***	True if *file* is a directory

An alternative syntax is **`[ -f`** ***file*** **`]`** (and similar for the other tests).

`while`

Syntax:

`while` *command-list*$_1$
`do` *command-list*$_2$
`done`

The value tested by `while` is the exit status of the last simple command following `while`. If this is non-zero, the loop terminates.

`shift`

Renames the positional parameters (arguments) `$2`, `$3`, … as `$1`, `$2`, … and loses `$1`.

`until`

The syntax is the same as a `while` loop, but with `until` instead of `while`. The termination condition is reversed.

`if`

Syntax:

`if` *command-list*
`then` *command-list*
`else` *command-list*
`fi`

The value tested is the exit status of the last simple command following `if`. Of course, `if` statements may be nested.

Command grouping

There are two ways to group commands:

`{` *command-list* **`;`** **`}`**

and

(*command-list* **)**

The first form simply executes the commands; the second form executes them as a separate process.

Debugging

Using **`set -v`** causes the lines of the procedure to be printed as they are executed: verbose mode. It may also be invoked by typing **`sh -v`** ***script***. This may be used in conjunction with the **`-n`** flag, which prevents execution of subsequent commands (note that saying `set -n` at a terminal will lock the terminal until an end-of-file is typed).

The command **`set -x`** will produce an execution trace: following parameter substitution each command is printed as it is executed.

Both flags may be turned off using **`set -`**, and the current setting of the shell flags is available as **`$-`**.

Substition

Variable substitution has been discussed. It is possible to specify defaults (**`${var-default}`**), assign the default to the variable if it wasn't set (**`${var=default}`**) and abort with a message if a variable isn't set (**`${var?message}`**). Command output substitution has the syntax:

`‘`*command***`‘`**

So, for example, on the system `hubble`, the command

```
cd /sys/MIPS/‘/bin/hostname‘/config
```

will change to the directory `/sys/MIPS/hubble/config`.

And, or

There are two conditional-execution tests:

command$_1$ **`&&`** *command*$_2$

command$_1$ **`||`** *command*$_2$

The first executes *command*$_2$ if *command*$_1$ returns a zero value. The second is the same, but for a non-zero value. Newlines may appear in the command list instead of semicolons, to delimit commands.

Invoking shells, login scripts and restricted shells

The **`sh`** shell takes the following parameters:

`-`	Indicates that this is a login shell. If **`$HOME/.profile`** exists, it is executed. This is the normal `sh` login script; put your commands here.
`-c` *string*	Commands are read from *string*.
`-s`	Commands are read from standard input; output is written to descriptor 2. (This is the default.)
`-i`	Shell is interactive. (This state is also assumed if the shell is attached to a terminal, as told by `gtty`.) The terminate signal `SIGTERM` is ignored (so `kill 0` doesn't kill an interactive shell); the interrupt signal `SIGINT` is caught and ignored (so that `wait` is interruptible). In all cases `SIGQUIT` is ignored.

csh executes the **.cshrc** file in your home directory when it starts. Additionally, if this is a login shell, it executes **.login** when it starts and **.logout** when is finishes.

The **restricted shell** rsh5 differs from sh5 in that the following are disallowed:

- changing directory
- setting $PATH
- specifying path or command names containing /
- redirecting output (> and >>)

These restrictions are enforced after .profile is interpreted. When a command is found to be a shell procedure, rsh5 invokes sh5 to run it. Therefore shell scripts can be provided that have the full power as the normal shell, providing a limited selection of commands to the user. In this case, you would not want to give the user write and execute permissions to his directory. It is common to set up a directory of restricted commands, usually /usr/rbin, to simplify the management of many restricted users.

Two lines about csh

It's got different names for its login/logout scripts. It's more powerful than sh. It's got a command history and command-line editing. It's got a command syntax like C. Look it up as csh(1).

Accounting

System accounting can be performed for (1) user logins; (2) command usage; (3) printer usage. Most of the commands that will be described rely on the presence of the optional acccounting software subset.

Login accounting

The system maintains two login accounting files: `/etc/utmp` records active logins and `/usr/adm/wtmp` maintains a login history. To generate a report of the login history, run

`/etc/ac [ -p ] [ -d ] [` ***people*** **`]`**

where

`-p`	prints totals for individuals (otherwise the grand total you get isn't much use!)
`-d`	gives a breakdown on a daily basis
people	restricts the report to *people*

To clear the login history, truncate it:

```
cp /dev/null /usr/adm/wtmp
```

The system only maintains a login history if `/usr/adm/wtmp` exists (so remove it to stop login accounting and create it using `touch` to restart login accounting).

The `last` command shows the last logins of users and teletypes. Its syntax is

`last [ -`*n*` ] [ `*name*`... ] [ `*tty*`... ]`

This gives login times and duration of the last *n* logins (default: all) for *names* and *ttys*. (Note that `last root console` shows logins of `root` *and* onto the console, not just of `root` on the console).

Command usage accounting

Normally, `/etc/rc` enables process accounting at system startup. The system records information on each executed process in `/usr/adm/acct`. This can be disabled to save disk space. Process accounting is suspended when free space < 2%, and resumes when free space > 4%.

To generate a report, use `sa`:

`sa [` ***options*** **`]`**

where

`-s`	Merge accounting file into summary file `/usr/adm/savacct` when done.
(others)	See `sa`(8).

To enable and disable process accounting, use `/etc/accton`. On its own, `/etc/accton` will disable process accounting immediately. The normal line in the `/etc/rc` file to enable accounting is given below: comment it out to disable accounting more permanently.

```
/etc/accton /usr/adm/acct; echo -n ' accounting' > /dev/console
```

The `lastcomm` command shows the last commands executed in reverse order. Its syntax is

`lastcomm [` *command* `] [` *user* `] [` *terminal* `]`

For example, `lastcomm print root ttyp3` gives details of all `print` commands executed by `root` on `ttyp3`. For details of the output including the status flags it gives, see `lastcomm`(1).

Printer accounting

If a default accounting file is specified in the `/etc/printcap` file, all printer accouting information is recorded to it. If this is the case, you generate a printer usage report using `/etc/pac`, or for a specific printer using `/etc/pac -P`*printer*. To configure a useful accounting system, see `printcap`(5) and `pac`(8) for full details.

Error logs

Error logs are generated by the error logging daemon, **/etc/elcsd**, which is invoked by /etc/rc. Hardware and system-related error packets and messages are logged to syserror.*hostname*, where *hostname* is the name of the system. The elcsd daemon must be running whenever the system is in multiuser mode.

The daemon is configured by **/etc/elcsd.conf**. Here is a typical configuration file:

```
#static char    *sccsid = "@(#)elcsd.conf       1.3   (ULTRIX)  3/17/86";
#
#       elcsd - errlog configuration file
#
{                       # delimiter DON'T remove or comment out!
1                       # 1-local,2-logrem,4-remlog,5-remlog+priloglocal
                        # errlog file size limit num. of blocks
/usr/adm/syserr                 # errlog dir. path
                                # backup errlog dir. path
/                               # single user errlog dir. path
/usr/adm/syserr                 # log remote hosts errlog dir. path
                        # remote hostname to log to
}                       # delimiter DON'T remove or comment out!
#       hosts to log :S - separate file or :R - remotes file (together)
#remote1:S            - (example) log errors from remote1 into separate
file
```

Entries are delimited by a newline and a blank line constitutes a null entry. The status line (containing "1" above) can take the following values:

1 (default)	Logs error packets (messages) locally
2	Logs error packets from a remote system or systems to the local machine
3	Logs local and remote packets locally
4	Logs error packets from the local system to a remote system
5	Logs error packets from the local system remotely and logs high priority messages locally

The errorlog file size defines the maximum size of the errorlog file (what a surprise), in 512-byte blocks. If you do not specify a maximum, you will be notified when the file system is 98% full. The default errorlog directory is /usr/adm/syserr. If you change this, you must change the default for uerf too (see below). If the daemon cannot write to the primary errorlog, it uses the backup errorlog. If you run the daemon in single user mode (elcsd -s) it logs to the root directory by default (then moves the logged messages to the multiuser log when the system goes multiuser) – you can redirect this too. You can also specify a path to log remote hosts' errorlogs. If you specify remote hosts to log, append :S to make a separate log for a particular host, or :R to log to syserr.remotes. The default is :S. Remote logging is accomplished via an Internet datagram socket.

The elcsd daemon logs general status and error messages to /usr/adm/elcsdlog in multiuser mode; this file is purged whenever the daemon is started.

The program **/etc/eli** can be used to control the error logging daemon. It takes the following options:

-d	Disables error logging.
-e	Enables error logging in multiuser mode.
-f	Forces the subsequent option (no prompt is given). This is the only option that can be used with another.
-h	Help.
-i	Initializes the kernel errorlog buffer.
-l	Logs a one-line status message to the kernel errorlog buffer. Examples: `eli -f -l "This is a test message"`

```
eli -f -l < myfile > /dev/null
```

(The second example logs a message up to and including the first newline from `myfile`.)

`-n` Disables the logging of packets to disk by `elcsd`. High priority messages continue to be logged to the console; others can be viewed by `uerf -n` (see below) but otherwise are logged.

`-q` The console will not display the missed error messages that periodically occur with a full errorlog buffer.

`-r` Reconfigure. Use this after changing `/etc/elcsd.conf`.

`-s` Enables error logging in single user mode.

`-w` Enables the missed error message to appear on the console every 15 minutes (the opposite of `-q`).

And to generate a report from the errorlog? Use **`/etc/uerf`**, the ULTRIX error report formatter. For full syntax see `uerf`(8). For a very good summary of the options, type `uerf -h`. On its own, `uerf` generates a report from the default input file; it is wise to redirect it (`uerf > errorreport`) because it's usually long. Once you have the textual output you can purge the error log.

Mail

I will merely summarize the commands relevant to the standard UNIX mail system.

Command	Purpose
`ali` (1mh)	*list mail aliases*
`aliases` (5)	*aliases file for sendmail (`/etc/aliases`)*
`biff` (1)	*be notified if mail arrives and who it is from*
`capsar` (1)	*prepares documents not in ASCII format for transport in the mail system*
`from` (1)	*identifies sender of mail*
`inc` (1mh)	*incorporate new mail*
`mail` (1)	*send or read mail*
`newaliases` (1)	*rebuild the database for the mail aliases file*
`prmail` (1)	*print out mail in the post office*
`uuencode`, `uudecode` (1c)	*encode/decode a binary file for transmission via mail*
`xsend`, `xget`, `enroll` (1)	*secret mail (available only if the Encryption layered product is installed)*

- To send a message to `bigbird`, use "`mail bigbird`". Finish your message with `^D`.
- To retrieve your mail, type `mail`. Once in `mail`, type `?` for a summary of commands. Useful commands are `n` (go to and type next message) and `d` (delete message). Type `q` to quit.

Notes:

- Sending mail to remote superusers – see *NFS*.

Other handy commands: things left over

`which`	If I type a command, which file will be executed?
`clear`	Clear screen
`tty`	What terminal device am I connected to?
`who`	Who's on the system
`whoami`	Your username
`who am i`	Your username, the machine you're logged on, from where and since when
`w`	Who's logged on and what they're doing
`f`	Who's logged on, from where, since when, their real name and their office phone number. Same as finger with no parameters.
`finger` ***username***	Tells all about a user: real name, phone, home directory, where they're logged in from or their last login, on some systems whether they have mail waiting and when they last checked their mail, and their plan: if they have a file called "`.plan`" in their home directory, this is printed (and also the first line of a file called "`.project`", if it exists). Finger can also be used remotely, as in `finger user@host`.
`script`	Generate script of your terminal session
`file`	Determine file type
`mesg [n] [y]`	Allows or disallows talk/write messages. On its own, displays current setting.
`write` ***user*** **[***ttyname***]**	This command works for any user (although only the superuser can override a `mesg n` command) and can be a two-way chat if two users write to each other at the same time. Press `^D` to terminate your message. The full form of the command is useful when someone is logged on more than once: a typical command would be `write root ttyp0`.
`talk` ***user*****[***@host***]** **[***ttyname***]**	This is an interactive split-screen chat program. Use `talk` to initiate and to respond. Press `^L` to redraw the screen and `^C` to quit. Talk requires the `talkd` server daemon to be running (usually loaded from `inetd.conf`, see *Networking*).
`touch` ***file***	Updates the access date of *file*, creating it with zero length if it doesn't exist.
`chsh [` ***user*** **]**	Change the login shell for *user*.
`cmp` ***file1 file2***	Compares *file1* and *file2* and reports the differences.
`date [` ***parameters*** **]**	On its own, `date`(1) queries the system date and time. There are many options to retrieve the date in a certain format, and also to operate with UCT/GMT instead of local time. If you are superuser, you may use `date` to set the date and time. **Do not** change the date while the system is in

multi-user mode. The parameters for setting date and time are

```
[yy[mm[dd]]]hhmm[.ss][-[-]tttt][z]
```

Of these, `yy` is the last two digits of the year and is optional (1970 is the earliest, so “69” will mean 2069); `mm` is the month number; `dd` the day in the month; `hh` is the hour (24-hour clock); `mm` the minute; `ss` the second. The `-[-]tttt` specifies minutes west of Greenwich (a negative sign means east) and `z` is a one-letter DST correction mode (`n`=none, `u`=USA, `a`=Australian, `w`=Western Europe, `m`=middle Europe, `e`=Eastern Europe).

Getting help: where to go from here

The online manual, man

There are two versions of the `man` command (`/usr/bin/man` and `/usr/ucb/man`, the latter being the normal one) and the `man` command can be very complicated. Type "`man man`" for the complete syntax (there are 15 screenfuls of information!). However, the following is all you really need to know:

`man -k` *keyword* or `apropos` *keyword*	Prints a one-line summary of all commands whose name or description includes *keyword*. Use this when you're thinking "I want a command that does *this*…"
`man` *command*	Prints formatted pages of information about *command*. Press space for the next page, `q` to quit.
`man` *section command*	Looks up *command* in the specified *section* of the online manual. **Note:** the online manual contains the *Reference Pages* only. Its main sections are listed against *RefPages* in the list of manuals below.

One convention is worth noting, and has been used in this guide. If you see a reference to `stty`(3), for example, it refers to `stty` in section 3 of the *Reference Pages*. To look it up, type "`man 3 stty`". As it happens, there is also a command `stty`(1); the section 1 command is an executable program while the section 3 command is a C library function. If you just type "`man stty`", you will get the first entry (section 1).

About the text manuals

Only UNIX would need a comment on how to use its manuals! We have about seven feet of UNIX manuals and finding information is not easy.

The first thing to find is the Master Index. Individual manuals do not have an index. The Index will refer you to a manual and a page. Look up the manual from its abbreviation in the list below. Secondly, note that there are often several manuals in one folder.

The ULTRIX manuals and their abbreviations

Abbreviation	Manual Title
AdvInstall	Advanced Installation Guide
Backup	Guide to Backup and Restore
BasicInstall	Basic Installation Guide
BGB	The Big Gray Book: The Next Step with ULTRIX
BIND/Hes	Guide to the BIND/Hesiod Service
Configfile	Guide to Configuration File Maintenance
CrashRec	Guide to System Crash Recovery
CursesX	Guide to X/Open curses Screen Handling
DECrpc	DECrpc Programming Guide

DiskMaint	Guide to Disk Maintenance
DMS	Guide to Diskless Management Service
ErrorLogger	Guide to the Error Logger
Ethernet	Guide to Ethernet Communications Servers
Exercisers	Guide to System Exercisers
InterNatl	Guide to Developing International Software
Kerberos	Guide to Kerberos
LocBrok	Guide to the Location Broker
LAN	Introduction to Networking and Distributed System Services
nawk	Guide to the nawk Utility
Network_Prog	Guide to Network Programming
NFS	Guide to the Network File System
Primer	The Little Gray Book: An ULTRIX Primter
RefPages	Reference Pages Section 1: Commands Reference Pages Section 2: System Calls Reference Pages Section 3: Library Routines Reference Pages Section 4: Special Files Reference Pages Section 5: File Formats Reference Pages Section 7: Macro Packages and Conventions Reference Pages Section 8: Maintenance
RIS	Guide to Remote Installation Services
sccs	Guide to the Source Code Control System
setld	Guide to Preparing Software for Distribution on ULTRIX Systems
Shutdown	Guide to Shutdown and Startup
SysEnviron	Guide to System Environment Setup
SysNetSetup	Guide to System and Network Setup
TechSumm	Technical Summary
TermEmul	Guide to IBM Terminal Emulation for VAX Processors
uucp	Guide to the uucp Utility
X/OpenTransport	Guide to the X/Open Transport Interface
YP	Guide to the Yellow Pages Service

The End