National Centre for Computer Animation

INTRODUCTION TO LINUX

JONATHAN MACEY

September 27, 2005

BOURNEMOUTH MEDIA SCHOOL

Contents

List of Tables

Chapter 1

Introduction to Unix

Unix was developed in the 1970s at AT & T Bell Laboratories as a computer science exercise. Over the past thirty years it has evolved into arguably the most popular computer operating system in the world, running on the fastest supercomputers, the largest mainframes, the most popular minis, most graphics workstations and the more powerful microcomputers; and in the last five years in the guise of Linux, Free BSD and Intel Solaris onto many P.C. level machines.

1.1 Reasons for the success of Unix

- Written in C - the system is portable, modifiable and relatively easy to understand.
- Systems shell - provides a simple and effective user interface.
- Software reuse - simple primitives can be built up into new and powerful command tools.
- Hierarchical file structure - consistent, efficient and easily maintained.
- Byte oriented I/O - gives consistent file formats and allows device independence and easy redirection.
- Multi-user, multi-tasking - many users and many processes per user and per machine.
- Machine architecture hidden - allowing for portability of programs between Unix systems.
- Unix offer Portability, Flexibility, Power and Elegance.

1.1.1 Portable Operating System Interface (POSIX)

Posix is a set of standard operating system interfaces based on the Unix operating system. The need for standardisation arose because of enterprises using computers wanted to be able to develop programs that could be used on different manufacturers computers without being re-coded. Unix was selected as the basis for a standard system interface partly because it was "manufacturer neutral". However, several major versions of Unix existed so there was a need to develop a common denominator system.

Informally, each standard in the POSIX set is defined by a decimal following the POSIX. Thus POSIX.1 is the standard for an application program interface in the C language. POSIX.2 is the standard shell and utility interface (i.e. the users command interface with the operating system). These are the main POSIX interfaces however there are others such as POSIX.4 for thread management. These standards are now followed for most Unix / Linux versions and many components of Windows are now POSIX compliant.

1.2 Login

As Unix is a secure system all users must have a username and a password to enter the system. There are various user accounts on a Unix system each having different levels of access to the system.

1.2.1 The super user (root)

The *root* user, also know as the 'Super User' has full access to every part of the system. *root* is allowed to modify any part of the file system and create and destroy processes at will. As *root* is a very powerful account the use of the *root* user is heavily restricted on most systems and only certain systems administrators are given the password.

1.2.2 Other System accounts

There are other standard Unix logins on most systems with varying levels of access to the system. Some of these are used for administration purposes such as *admin* , *deamon* and *lp*. Other accounts such as *guest* and *ftp* are setup on a system to allow users to login with limited access to the system. The *ftp* account is generally used for anonymous *ftp* where the user logs into the system using an *ftp* client and the default username *ftp* and the password as the (remote) users email address.

1.2.3 User Accounts

User accounts are designed for the every day use of Unix systems users, they generally have a number of restrictions which by default are as follows

- Limited access to file system
- Reduced ability to run certain programs
- No / restricted access to hardware configurations

The main reason for this is to maintain the security and integrity of the system, it is easy for some of these restrictions to be removed but it is unwise to give a standard user account access to more powerful commands as this can compromise the integrity of the system. Therefore if a user is to have more powerful access right they should be given another administration type account to deal with system administration duties.

1.2.4 Home Directory

When a user first logs onto a Unix system, the working directory is know as the home directory. With a default Linux installation a users home directory is in the following place

```
/masters/<username>
```

Where <username> is the name the user logs on with

It is also important to note the direction of the directory separator '/' in a Unix file system / indicates the *root* of the file system. Every directory mounted from *root* (/) is followed by another /. This is unlike the windows file system where every directory is mounted from the drive letter (e.g. C:\ ) and the directory separator used is the '\'.

Chapter 2

The Unix Shell

Graphical environments for Unix have been around for many years and many complex tasks can be achieved from the GUI, so why do we need a shell in a Unix environment? For one thing it is generally faster to use the shell prompt to accomplish tasks than it is to open up a file manager move to the directory and copy the files. Also not all users have the ability to open up a GUI based program, for example if the user is running a *telnet* session from a windows machine the only way to interact with the system is to use the shell prompt.

2.1 Different Unix Shells

There are a variety of different shells available under Linux of which some are listed below

- Standard shell (sh)
- Bourne Again Shell (bash)
- C Shell (csh)
- Turbo C shell (tcsh)

Each of these shells have different features, however most modern shells feature the following basic operations

- File name completion
- History
- Doskey type command selection
- built in scripting languages

2.2 Using the Shell (bash)

When a console or terminal session is run a program called a shell is used to interpret the commands typed and execute them. The default shell installed on the systems is called *bash* .

2.2.1 File Name Completion

One of the most useful features is know as file name completion, this allows part of a file name or directory to be typed and a key pressed to complete the rest of the file name. In *bash* the tab key is used to complete the file name.

For example if there is a file called *example1.c* in the directory by typing *more e* followed by a tab the file name would be completed. However if there are several files called *example1.c example2.c* and *example3.c* typing *more e* followed by a tab would give *more example* after this pressing the tab again would list all of the files with the text *example* If *2* is then typed and tab pressed again the whole file name would be completed.

2.2.2 History

History records the most recent commands typed in the shell so the user may re-use them. The simplest method of using history is to type the shell command history which will give a recent list of all the commands typed preceded by a number. To execute a previous command from the history list use the *!* character followed by the number (note that there is no space between the *!* and the number) of the command from the history list.

To execute the last command typed in the shell the following short cut may be used *!!* followed by enter.

To execute the last command beginning with a specific character (for example if *more* was typed)

!m followed by enter.

Finally the up and down arrow key may be used to traverse the history list one command at a time.

2.3 Shell examples

The following example demonstrates the use of file name completion in the shell

```
#touch example1.c
```

The *touch* command creates an empty file with the name *example1.c*. By typing *more* and pressing the TAB key the following will be displayed

```
#more example1.c
```

Now we press the up arrow twice to get back to the *touch* command and create two more files called *example2.c* and *example3.c*

```
#touch example2.c
#touch example3.c
```

Now if we type *more e* and the TAB key twice we get the following display

```
example1.c example2.c example3.c
#more example
```

By typing 3 and then TAB again the the filename *example3.c* will be completed.

2.4 Changing the bash prompt

When executing interactively, *bash* displays the primary prompt *PS1* when it is ready to read a command, and the secondary prompt *PS2* when it needs more input to complete a command. *Bash* allows these prompt strings to be customised by inserting a number of *backslash-escaped* special characters that are decoded as follows:

\a	an *ASCII* bell character (07)
\d	the date in "Weekday Month Date" format (e.g., "Tue May 26")
\e	an *ASCII* escape character (033)
\h	the *hostname* up to the first '.'
\H	the *hostname*
\j	the number of *jobs* currently managed by the shell
\l	the basename of the shell's terminal device name
\n	newline
\r	carriage return
\s	the name of the shell, the basename of $0 (the portion following the final slash)
\t	the current time in 24-hour HH:MM:SS format
\T	the current time in 12-hour HH:MM:SS format
\@	the current time in 12-hour am/pm format
\u	the *username* of the current user
\v	the version of *bash* (e.g., 2.00)
\V	the release of *bash*, version + patchlevel (e.g., 2.00.0)
\w	the current working *directory*
\W	the basename of the current working *directory*
\!	the history number of this command
\#	the command number of this command
\$	if the effective *UID* is 0[1], a #, otherwise a $
$	\nnn the character corresponding to the octal number nnn
\\	a backslash
\[	begin a sequence of non-printing characters, which could be used to embed a terminal control sequence into the prompt
\]	end a sequence of non-printing characters

[1] The root user

2.4.1 Creating a default prompt

There are two files which may be edited to set the *PS1* and *PS2* variables. One file will be set for all users who log into the system, the other file will be specific to the user when they log in. The global file is */etc/profile* and the user file is found in the root of the users *HOME* directory and is called *.bashrc*. If a *.bashrc* file contains a *PS1* and *PS2* variable then this will override the global profile.

For example to change the default prompt to display the *username*, the basename of the current working directory and the *$* prompt if the user is a normal user or *#* if the user is *root* the following *PS1* string can be set in the console using the *export* command

```
export PS1="[\u:\W]\$"
```

This will result in the users prompts looking as follows

```
[jmacey:Unixcourse]$
```

Using the *export* command in the console will set the *PS1* string for as long as the console is running. To make this change permanent the command may be placed into the *.bashrc* or the */etc/profile* system files and the command will then be executed each time the shell is run.

2.5 Aliasing commands

Aliases allow a string to be substituted for a word when it is used as the first word of a simple command. The shell maintains a list of aliases that may be set and unset with the *alias* and *unalias* built in commands. Aliases are created and listed with the alias command, and removed with the unalias command.

For example the *rm* command will by default remove the files passed to it without prompting. To allow the default behaviour of *rm* to prompt the user for a yes / no when deleting files the user could type *rm -i*. However this can be time consuming so using the *alias* command the user can set the *rm* to always run *rm -i*. This is done as follows.

```
alias rm="rm -i"
```

No when typing *rm* the following will be shown

```
rm test.sh
rm: remove ‘test.sh’?
```

Pressing *y* will delete the file and *n* will leave it.

2.5.1 Making aliases permanent

As with the setting the default prompt *aliases* may be made permanent by adding them either to the */etc/profile* or the *.bashrc* files. When changes are made to these files they do not become active until the file has been re-interpreted. This will happen if the shell is closed and re-run or by using the *source* command as follows

```
source ~/.bashrc
```

Typing the *alias* command without any arguments will list all of the aliased commands set on the system.

2.5.2 unalias

To remove any aliases set on the system the *unalias* command is used as follows

```
unalias rm
```

2.5.3 dos to unix alias

The following set of commands can be placed into the *.bashrc* file to create a set of *dos* to *unix* commands.

```
alias dir="ls"
alias "dir/w"="ls -al"
alias "dir/p"="ls |more"
alias type="more"
alias copy="cp"
alias rn="mv"
alias rename="mv"
alias del="rm -i"
alias attrib="chmod"
alias md="mkdir"
alias help="man"
alias chkdsk="df -k"
alias print="lpr"
```

2.6 ctrl+z and bg

Sometimes when using the shell a command is run and the &is not used; if this command is a GUI based program the shell will lose it's interactivity and all commands typed will have no effect. This is because the program run now has control of the console and no more commands may be typed until this program is terminated.

To overcome this problem the *bash* contains facilities to access the operating systems job control system by typing the suspend character (typically ^Z, *CTRL-Z*) while a process is running this causes that process to be stopped and returns control to *bash*.

The user may then manipulate the state of this job, using the *bg* command to continue it in the background, the *fg* command to continue it in the foreground, or the *kill* command to kill it.

A ^Z takes effect immediately, and has the additional side effect of causing pending output and type ahead to be discarded.

2.7 ctrl +c stopping processes

Sometimes a process will either hang in a terminal or will not stop execution, if this happens the process can be interrupted with the *ctrl+c* key combination if this fails to work the kill control key *ctrl +u* may be used.

2.8 Shell movement commands

When using the command line in the shell it is possible to move left and right within the current command being typed with the arrow keys. The *BkSp* and *Del* key may be used to delete in the left and right direction respectively, and the following keys may also be used for navigation.

ctrl+a	Move to the start of the current line.
ctrl+e	Move to the end of the line.
ctrl+f	Move forward a character.
ctrl+b	Move back a character.
ctrl+f	Move forward to the end of the next word. Words are composed of alphanumeric characters (letters and digits).
alt+b	Move back to the start of the current or previous word. Words are composed of alphanumeric characters (letters and digits).
ctrl+l	Clear the screen leaving the current line at the top of the screen.

2.8.1 *clear* and *reset*

clear is used to clear the console / terminal and return the cursor to the top. This is similar to typing *ctrl+l* in *bash.*

Sometimes the terminal window will become scrambled and non printing characters will replace the usual characters[2]. The easiest way of resetting the console if this happens is by use of the *reset* console command.

2.9 Getting Help

Unix contains a comprehensive help system called *man* The word man stands for manual, a series of on-line pages which can tell the user the purpose of many commands. The *man* pages provide a summary of a command's purpose, the options available and the syntax which is used to issue the command. All man pages are formatted in the same way to give a consistent look and feel, this means that once the user is familiar with the format it is easy to quickly extract the relevant information from the man pages.

2.9.1 Man page sections

All man pages are split into the following sections

```
Name
Synopsis
Description
Options
See Also
```

The *Name* and the *Synopsis* sections give a brief description of the command being looked up.

The *description* section gives a more detailed explanation of the command and its uses, as most Unix commands have a variety of command line options these are listed next in the *options* section.

Finally related or similar commands are listed in the *See Also* section. These commands will have their own man pages which may be examined by using the *man* command again with the different command name.

The following page shows the manual page for the *touch*[3] utility with all of the different sections.

[2] This usually happens when using *more* on a binary file.
[3] Note this is for the GNU / Linux version of touch

```
TOUCH(1)                FSF                TOUCH(1)
NAME
     touch - change file timestamps
SYNOPSIS
     touch [OPTION]... FILE...
DESCRIPTION
     Update the access and modification times of each FILE  to
     the current time.

     -a     change only the access time

     -c, –no-create
          do not create any files
     -d, –date=STRING
          parse STRING and use it instead of current time
     -f     (ignored)
     -m     change only the modification time
     -r, –reference=FILE use this file's times instead of current
     -t STAMP
          use [[CC]YY]MMDDhhmm[.ss] instead of current time
     –time=WORD
          set  time  given by WORD: access atime use (same as
          -a) modify mtime (same as -m)
     –help display this help and exit
     –version
          output version information and exit
     Note that the three time-date formats recognized for the -d and -t options and   for the obsolescent argument are all different.

AUTHOR
     Written by Paul Rubin, Arnold Robbins, Jim Kingdon,  David
     MacKenzie, and Randy Smith.

REPORTING BUGS
     Report bugs to <bug-fileutils@gnu.org>.
SEE ALSO
     The  full  documentation for touch is maintained as a Texinfo manual.  If the   info and touch programs are properly installed at your site,
the command
```

Man pages are formatted to be displayed in the console if printed versions are required they must be formatted to remove any non printable characters as follows

```
#man touch | col -b | lpr
```

The above command is actually a compound command where the output of one command is fed into the input of another command. First the *man* command is used to print out the *man* page. This is then fed into the command *col -b* using the | . The *col -b* command removes any backspaces from the output of the man command. Finally the output of the *col* command is sent to the printer using the *lpr* command.

2.10 Where am I? and Who am I?

As a unix system is usually a distributed operating system with many users it is important to be able to identify who the user is, which machine the user is on an where on the machine the user is. To do this the following commands are used

id, hostname, who, pwd

```
#id
uid=1549(jmacey) gid=100(users)
#
```

The basic output from the *id* command is shown above, this gives information about the user id (*uid*) which is a numeric value with the names in brackets. Next the group id is shown (*gid*) this is the primary group the user belongs to and finally any supplementary group the user belongs to (if none this information is omitted).

```
#hostname
rh1610
#
```

The *hostname* command prints out the name of the machine that the user is currently logged into.

```
#who
jmacey   :0       Oct 17 11:59
u9573564   pts/1    Oct 17 18:30
#
```

The *who* command prints out the names of the users currently logged on to the system. As more than one user may be logged into the system at any one time this command is useful for finding out who is on the system. For more detailed information about a user logged on the system the *finger* command may be used as follows

```
#finger jmacey
Login: jmacey                          Name: Jonathan Macey
Directory: /home/jmacey                Shell: /bin/bash
On since Tue Oct 17 11:59 (BST) on :0 (messages off)
On since Tue Oct 17 18:30 (BST) on pts/1 from :0
On since Tue Oct 17 21:34 (BST) on pts/2 from :0
   6 seconds idle
No mail.
Plan:
```

The plan section of the *finger* output is created by the user using a file called .plan created in the users home directory. This is useful for telling other users on the system what you are doing however on some distributed systems (like the University) this will not work due to the way the file system has been setup for security reasons.

Finally to find out where in the file system you are the *pwd* command may be used as follows

```
#pwd
/cgstaff/jmacey
```

pwd prints out the current file system location relative to the *root* directory.

2.11 System Processes

Every time a program is executed on a Unix system it is given a unique integer id. This is known as the process id or *pid*. Only the user who has created a process can modify this process, this is know as ownership. The only exception to this rule is the *root* user who has the ability to *kill* any process created by any user.

2.11.1 Listing processes

To list processes the *ps* command is used as follows

```
#ps
 PID TTY          TIME CMD
 7073 pts/1    00:00:00 bash
 7161 pts/1    00:00:00 xclock
 7371 pts/1    00:00:00 ps
#
```

The output of the *ps* command only shows the processes running from the current terminal with the current user. However with the use of *flags* more processes may be shown

```
#ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 11:57 ?        00:00:06 init [5]
root         6     1  0 11:57 ?        00:00:00 [mdrecoveryd]
root        61     1  0 11:58 ?        00:00:00 [khubd]
daemon     442     1  0 11:58 ?        00:00:00 /usr/sbin/atd
root       468     1  0 11:58 ?        00:00:00 [cardmgr]
root       514     1  0 11:58 ?        00:00:00 [pump]
lp         515     1  0 11:58 ?        00:00:00 [lpd]
root       563     1  0 11:58 ?        00:00:00 sendmail: accepting connections
root       579     1  0 11:58 ?        00:00:01 gpm -t ps/2
root       594     1  0 11:58 ?        00:00:00 crond
xfs        627     1  0 11:58 ?        00:00:00 xfs -droppriv -daemon
root       659     1  0 11:58 ?        00:00:00 rhnsd --interval 30
root       682     1  0 11:58 ?        00:00:00 [gdm]
root       688   682  9 11:58 ?        00:59:57 /usr/bin/X11/X -auth /var/gdm/:0
jmacey    6893     1  1 15:39 ?        00:04:45 /office52/program/soffice.bin
jmacey    7073  7069  0 18:30 pts/1    00:00:00 bash
jmacey    7161  7073  0 19:56 pts/1    00:00:00 xclock
jmacey    7263  7069  0 21:34 pts/2    00:00:00 bash
```

This list now contains all of the processes on the system as the *-ef* flags tells *ps* to show all processes (*-e*) and format it in a long output (*-f*)

It is possible that there may be many processes on the system and sometimes these will take up many pages of the terminal screen. To overcome this problem we can feed the output of *ps* into the *more* utility as follows

```
#ps -ef | more
```

The output of *ps* will now be shown a page at a time and the space bar may be used to select the next page of the output.

It is sometimes desirable to select processes by another criteria such as *username* to do this we can feed the output of *ps* into the *grep* command as follows

```
#ps -ef | grep jmacey
UID        PID  PPID  C STIME TTY          TIME CMD
jmacey     738     1  0 11:59 ?        00:00:44 sawfish --sm-client-id=de ault2
jmacey     817     1  0 11:59 ?        00:00:06 xscreensaver -no-splash - imeout
jmacey    6893     1  1 15:39 ?        00:04:45 /office52/program/soffice bin
jmacey    7073  7069  0 18:30 pts/1    00:00:00 bash
jmacey    7161  7073  0 19:56 pts/1    00:00:00 xclock
jmacey    7263  7069  0 21:34 pts/2    00:00:00 bash
```

The *grep* command is used to find regular expressions in text. So in the above example all lines of text which contain the text *jmacey* are printed out. There is an alternative command called *pgrep* which does a similar thing to the command above

```
#pgrep -lu jmacey
7073  bash
7161  xclock
7263  bash
```

pgrep prints out the *pid* and the *name* of the process and the flag *-l* tells *pgrep* to print out the name of the program running and -u <*username*> specifies the user to look for.

2.11.2 Killing processes

Sometimes it is desirable to stop a process from running on the system, to do this two things are required. Firstly the *pid* of the process must be known and secondly the user trying to kill the process must either own the process or be the *root* user.

The *kill* command works by sending a *signal* to the currently running process, depending upon the *signal* sent the process will respond in different ways. A list of the common *signals* are shown in table 2.1.

Signal	Value	Action	Comment
SIGHUP	1	A	Hangup detected on controlling terminal or death of process
SIGINT	2	A	Interrupt from keyboard
SIGQUIT	3	C	Quit from keyboard
SIGILL	4	C	Illegal Instruction
SIGABRT	6	C	Abort signal from abort (3)
SIGFPE	8	C	Floating point exception
SIGKILL	9	AEF	Kill signal
SIGSEGV	11	C	Invalid memory reference
SIGPIPE	13	A	Broken pipe: write to pipe with no readers
SIGALRM	14	A	Timer signal from alarm (2)
SIGTERM	15	A	Termination signal
SIGCHLD	20,17,18	B	Child stopped or terminated
SIGCONT	19,18,25		Continue if stopped

Table 2.1: Unix Signals

The letters in the "Action" column of table2.1 are shown in table2.2.

Action	Description
A	Default action is to terminate the process.
B	Default action is to ignore the signal.
C	Default action is to terminate the process and dump core.
D	Default action is to stop the process.
E	Signal cannot be caught.
F	Signal cannot be ignored.

Table 2.2: Actions for signal

Usually the SIGKILL (-9) signal is used to *kill* a process as this will try to shut the process down properly, however sometimes processes do not respond to a *kill -9* and the command *kill -15* must be used.

The following example shows how ps and kill can be used in combination

```
#xload & xload& xload &
[6] 1371
[7] 1372
[8] 1373
```

This command will run 3 copies of the program *xload*. Notice that the & is used at the end of the call to *xload*, this tells the program to detach itself from the console and run as a separate process. If this was not used the console would not be available until the *xload* program had completed.

Now we can look for the pid of the *xload* processes by using the following command (note that the *pid* will be different for each system this example is run on)

```
#ps | grep xload
1373    pts/4   0:00    xload
1373    pts/4   0:00    xload
1373    pts/4   0:00    xload
1373    pts/4   0:00    xload
#
```

Now we can *kill* the *xload* programs by using the *kill -9* command as follows

```
#kill -9 1371
[7] killed         xload
#
```

This can then be repeated for each of the different pid's of the *xload* processes. However it is also possible to *kill* many processes at the same time by the use of the *pkill* command as follows

```
#pkill -9 xload
```

This command will kill all processes called *xload* with a signal *-9*, as long as the user calling the process either owns the process or is *root*.

2.11.3 more, cat and indirection

The following section shows how unix commands can be connected together by use of indirection. This has already been seen with the use of the pipe (|) feeding the output of *ps* into the *grep* command. Unix also allows the output of a program to be fed into a file by using the > or > > operators.

2.11.4 Using cat

The *cat* command has many uses and can operate in a number of ways, firstly it can operate in a similar fashion to *more* as follows

```
#cat /etc/passwd
```

This will print out the contents of the file */etc/passwd* to the console. However another use for the *cat* command is in combination with the > operator as follows

```
#cat /etc/passwd > pass.txt
#cat pass.txt
```

This will create a file in the current directory call *pass.txt* which contains the contents of the */etc/passwd* file.

It must be noted that every time the > command is used the resultant file is recreated from scratch. If the file is to be added to (concatenated) the > > operator is used as follows

```
#cat /etc/hosts >> pass.txt
#cat pass.txt
```

cat can also be used with multiple file arguments as follows

```
#cat /etc/hosts /etc/passwd > p2.txt
#cat p2.txt
```

cat can also be used with the standard input as a simple line editor, as follows

```
#cat > test.txt
```

This puts *cat* into line editor mode where each line of text can be edited until the return key is pressed which will then start a new line of the file.

To quit this mode *ctrl + c* must be pressed which will save the file. The file can then be viewed using the following command

```
#cat test.txt
```

2.12 Changing passwords

To change your password the *passwd* command is used as follows,

```
passwd Old password: enter your current password
New password: enter your new password
Retype new password: re-enter your new password
```

The passwords will not appear on the screen as you type, to prevent other people from seeing them. If you make a mistake, the message

```
Mismatch - password unchanged.
```

is displayed and your password remains unchanged. If the system uses a NIS or YP the call to *passwd* will call the *yppasswd* to change the password, however this will take some time to synchronise with the *NIS* database and will not take effect until the *NIS* has been re-made.

Chapter 3

The Unix File system

The unix file system follows a hierarchical structure with all disks and devices mounted from a common source. This common source is know as *root* and is usually depicted using a forward slash (/) and all directories hang off of this one common root. This is also true if the system has more than one disk as this will be mounted on to the *root* file system and will appear as a directory.

The file system can be drawn as a tree like structure as shown in figure 3.1 and most unix implementations have a similar layout.

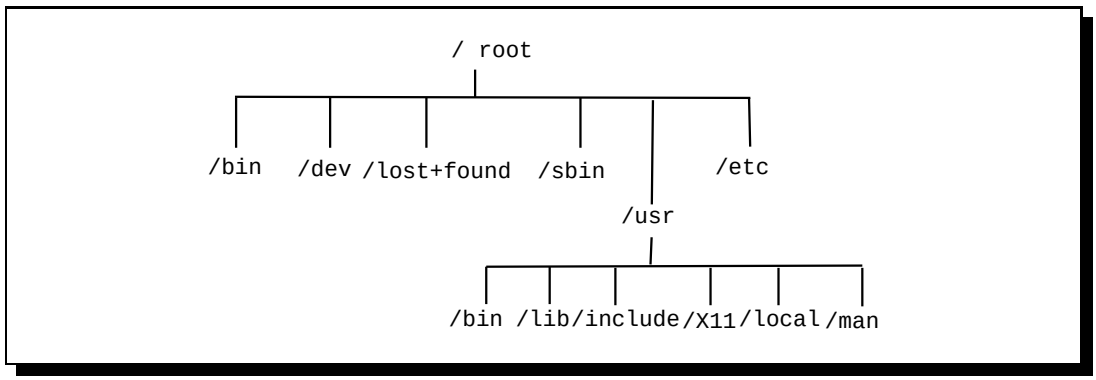

Figure 3.1: Unix File system

3.1 Studio file system structure

The following list show important directories in the Linux Studios at the NCCA

/masters - home dirs of all students

/mapublic - masters public usually used for staff to place important lecture files

/cgstaff - staff home dirs

/tmp - temp directory used by some applications

/transfer - scratch areas for large files and project, useful for sharing work and local machine access to data (important all people have access to this and it is cleared on a regular basis)

/media - various mount point areas.

IMPORTANT : your home directories are not backed up on any server so work that is deleted will be lost. It is your responsibility to make archives of work on either DVD or external hard drives.

3.2 File permissions

Each file has a number of *attributes* associated with it. These may be seen by using the *ls* (list files) command as follows

```
#ls -l
-rw------- 1 jmacey other 0 Nov 29 14:33 example1.c
-rw------- 1 jmacey other 0 Nov 29 14:33 example2.c
-rw------- 1 jmacey other 0 Nov 29 14:33 example3.c
 -rw-r--r--    1 jmacey   users         849 Oct 24 12:00 pass.txt
```

This listing may be a bit confusing at first but it is broken down into the following sections starting with the first column.

3.2.1 The file type

The first character of the first column indicates the file type and the characters in table 3.1 are used.

Type	Description
-	Indicates a regular file
d	Indicates a directory
c	Indicates a character mode special device file
b	Indicates a block mode special device file
p	Indicates a FIFO (or named pipe) used for inter process communication
l	Indicates a symbolic link

Table 3.1: File types

Some versions of unix extend these types to show whether files are symbolic links, and whether they are network mounted or not.

3.2.2 Unix file access control

Immediately following the file type character are nine characters that indicate the permissions by which the users and their processes can access the file. These characters consist of three sets of the characters, *rwx* where *r* indicates the read permissions, *w* indicates write permissions, and *x* indicates execute permissions.

The first set of permissions grant access to the owner of the file, the second set grants permissions to the group associated with the file (by default this is the same group as the owner of the file) and the third set grant permissions to all other users of the system (sometimes know as world permissions).

The following example shows a file listing

```
#ls -l
 -rwxr-x---    1 jmacey   users         849 Oct 24 12:00 test.sh
```

In the above example the file *test.sh* has read, write and execute permissions for the owner of the file (in this case the user jmacey), read and execute permissions for the group (in this case the group users) and finally no access for all other users (no permissions in this case is indicated by the use of the -).

To make it easy to set all the file permission attributes of a file at once without having to specify *r,w* and *x* individually, the numbers 0 - 7 have been assigned to a user, group and world. The original authors of unix found this handy shorthand notation as this mapping corresponded to the octal numbering system prevalent in the computer world of the time, this system is shown in table 3.2

Octal	Mode
0	—
1	–x
2	-w-
3	-wx
4	r–
5	r-x
6	rw-
7	rwx

Table 3.2: Octal file mode

To change these file permissions the *chmod* command is used with the octal values above, as shown in the following example

```
#cat >test.sh
ls -al
[ctrl + c]
#ls -l test.sh
  -rw-r--r--    1 jmacey   users         849 Oct 24 12:00 test.sh
```

The above example creates a file called *tesh.sh* which contains the line *ls -l*. Looking at the file permissions the file has *rw* for the user and *r* for the other groups. As this file contains a unix command it could be made executable and executed as a simple script (similar to a dos batch file). To do this we have to give the file execute permissions as follows

```
#chmod 750 test.sh
#ls -l test.sh
 -rwxr-x---    1 jmacey   users         849 Oct 24 12:00 test.sh
```

This command gives the file *rwx* permissions for the user *r-x* for the group and none for the world. This file can now be executed and is run by typing the name of the file as follows

```
#test.sh
```

3.2.3 Other file attributes

After the file access control bits the output of *ls -l* shows the number of links to the file, followed by the name of the user and group associated with the file. This is followed by the size of the file (in bytes), the date the file was last modified and the name of the file itself.

All of these attributes can be changed using standard unix commands however most of these are not available to standard users and for the most part they can only be changed by the *root* user.

```
#chown root test.sh
```

Changes the owner of the file *test.sh* to *root*, this command can also be used to change the user and the group in one go as follows

```
#chown root:root test.sh
```

After the username the group can be added by using the : alternatively the *chgrp* command can be used to change only the group as follows

```
#chgrp root test.sh
```

Important Note :
The above commands will not work correctly as the user must be root to change ownership. It must also be noted that if a file is copied that belongs to another user the file file's ownership will be modified to that of the user copying the file. Therefore for most systems and usages the chmod and chgrp commands are not required.

3.3 *umask*

By default the creation of files will have set permissions. This is usually set to *rwxr-xr-x* on most systems. To set the the default file creation mask to something different the *umask* function is used.

This is built into the *bash* shell and is used as follows

```
umask [-p] [-S] [mode]
```

The user file creation mask is set to mode. If mode begins with a digit, it is interpreted as an octal number; otherwise it is interpreted as a symbolic mode mask similar to that accepted by *chmod*. If mode is omitted, the current value of the mask is printed. The *-S* option causes the mask to be printed in symbolic form; the default output is an octal number. If the *-p* option is supplied, and mode is omitted, the output is in a form that may be reused as input. The return status is 0 if the mode was successfully changed or if no mode argument was supplied, and false otherwise.

3.4 File system navigation

We have already seen some file system navigation commands, two of the most useful being *pwd* (print working directory) and *ls* (list files). The following section uses these commands in conjunction with other commands to navigate the file system and create new directories.

To move within the unix file system the *cd* command is used, *cd* is exactly the same as the dos *cd* command and is used to change directories, however it also has a few additional features which makes it more flexible.

To start with the following example changes directory to the *root* of the file system

```
#cd /
#pwd
 /
#cd
#pwd
/home/jmacey/
```

The first *cd* changes to the root of the file system, this can be verified by using the *pwd* command. After this *cd* with no argument is typed. This command will return to the users home directory which can be verified by the use of the *pwd* command again.

This is very useful as wherever in the file system the user is they can change back to their home directory by just typing *cd*.

3.4.1 Making directories

To create a directory you must have permissions to the parent directory in which you wish to create the directory in. This is a security feature to restrict access to the file system. Therefore for most cases users only make directories in their home directory. The only exception to this is the *root* account who can make directories in any part of the file system and then grant permissions to other users to use the directories using the *chgrp* and *chown* commands.

The following example shows how a user can make a directory

```
#mkdir test
#cd test
#pwd
  /cgstaff/jmacey/test
```

These commands make a directory called *test* and then *cd* is used to change into the directory. It is also possible to make a directory tree using the following command

```
#cd
#mkdir -p test/another/another2
```

This will make a directory tree creating the directories if they don't exist.

To remove a directory the *rmdir* command is used as follows

```
#cd ~/test/another
#rmdir another2
```

The first command changes to the directory */cgstaff/jmacey/test/another/* however the ~/ is used. This is unix short hand for "from the home directory". Then the *rmdir* command is used to remove the directory *another2*.

It must be noted that the directory and any subsequent sub-directories must be empty to use the *rmdir* command. If the whole directory tree and files in the directory are to be removed the following command is used

```
#cd
#rm -rf test
```

This command recursively (*-r*) traverses the directory tree and deletes all files and directories without prompting the user (*-f*).

Important Note :
The rm command and especially the rm -rf command must be used with caution as there is no undo or undelete function in unix.

The *rm* command may also be used to delete files on a singular basis as follows

```
#touch 1.c 2.c 3.c 4.c 5.c
#rm 1.c
```

The first command creates 5 files with *a .c* extension. Then *rm* is used passing the name of the file to be deleted. Wild cards may also be used to delete files as in the following example

```
#rm *.c
```

This will delete all files with a *.c* extension.

Important Note :
*The * wildcard on its own means "all files" this can be dangerous when used in conjunction with the rm command especially rm -rf * which will delete recursively all files from the current directory downwards*

3.4.2 Moving and re-naming files

The unix command *mv* (move) has a dual purpose, it can either be used to rename a file or directory or it can be used to move a file or directory. The following examples show how it can be used.

```
#touch 1.c
#mv 1.c 2.c
```

The first a file is created called *1.c*. This is then renamed by using the *mv* command

```
#mkdir test
#mv test test2
```

In the above example a directory called *test* is created, next this is renamed to *test2* by using the *mv* command.

```
#touch 1.c 2.c 3.c 4.c 5.c
#mv *.c test2
#cd test2
#ls
```

In this example 5 *.c* files are created and then all files with a *.c* extension are moved into the *test2* directory using the *mv* command.

3.5 File utilities

There are a number of file system utilities which give information about the state and usage of the file system.

The following command shows the disk usage of the file systems showing each of the disk partitions mounted on the *root* file system

```
#df -k
Filesystem           1k-blocks   Used    Available  Use%    Mounted on
/dev/hda2              2016044   1659360    254272   87%    /
/dev/hda7               940788    723204    169796   81%    /home
/dev/hda1              2574972   1478100   1096872   58%    /windows
```

In the above example the output of df is shown in K blocks, and the amount used and available is shown. This is also summarised as a percentage of disk spaced used for each of the partitions.

To show the overall disk usage of a directory and sub directories the *du* command is used as follows

```
#du -k
```

This command prints out a total number of bytes used in all of all the directories in the tree and a total of all of the files at the end. However using the *du -ks* command will put *du* into silent mode and print out only the total amount of bytes used by the directory. For more detailed information about a file the *file* command can be used this prints out the type of the file as follows

```
#file test.sh
  test.sh: commands text
```

3.5.1 Finding files and text within files

To find files within the unix file system the *find* utility is used as follows

```
#find ./ -name "*.c"
1.c
2.c
3.c
```

The *find* command has the ability to execute other commands passing the name of the file found as an argument to the executed command. This is shown in the example below

```
#find /etc -name "*" -exec grep -l root {} \;
```

In this example the find command is pointed to the */etc* directory and told to find all files (*). next the *-exec* flag is used to indicate that another command is to be executed followed by the command. In this case it is the *grep* command with the flags *-l* and the text to search for as *root*.

Where the file name would usually come in the *grep* command the *{}* braces appear, this indicates where the output of the find should put the file name. Finally the *\;* terminates the command.

3.5.2 Symbolic links

Files do not actually reside inside directories. A directory is a file that contains references to other files. The directory holds two pieces of information about each file:

- Its filename

- An inode number which acts as a pointer to where the system can find the information it needs about this file.

Filenames are only used by the system to locate a file and its corresponding inode number. This correspondence is called a *link*. To the system, the file is the *inode* number. Multiple filenames can be used to refer to the same file by creating a link between an inode and each of the filenames. The following example shows how this work

```
#mkdir source
#cd source
#touch 1.c 2.c 3.c 4.c 5.c
#cd ..
#ln -s source dest
#cd dest
#ls
 1.c 2.c 3.c 4.c 5.c 6.c
```

First we make a directory called *source*, we then change into the directory and make some files. Next we change to the directory below and create a symbolic link, using the *ln* command.

The directory now has a new directory entry which is a link to the *source* directory. Changing into this *dest* directory it appears to be the same as the source and any changes made to either will be reflected in the other directory. It is also possible to make symbolic links to individual files using the ln command and this is used by systems administrators to rename files for commonality and backwards compatibility.

3.6 Creating archives using tar

In early *unix* systems the only method of backup was the use of magnetic tape. The method of backing up to a tape was to use the tape archive (*tar*) utility which would mount the tape drive and either extract or create and archive to the tape. However *tar* also has the ability to create an archive on a specified directory not just a tape so it is still used to create backup (or archives) in modern *unix* systems.

tar works by creating a an archive file containing all of the files in the directory to be archived. This means that the *tar* file will be the same size as the total of all the files in the directory.

3.6.1 *tar* command line options

tar has a number of command line options to change the way it works, these are shown below

-d	find differences between archive and file system
–delete	delete from the archive (not for use on mag tapes!)
-r	append files to the end of an archive
-t	list the contents of an archive
-u	only append files that are newer than copy in archive
-x	extract files from an archive
-f<F>	use archive file or device F (default /dev/rmt0[1])

[1] the tape device

-i	ignore blocks of zeros in archive (normally mean EOF)
-k	keep existing files; don't overwrite them from archive
-K	begin at file F in the archive
-v	verbosely list files processed
-w	ask for confirmation for every action
-z	filter the archive through gzip[2]

3.6.2 creating a *tar* file

To create a *tar* archive of a directory the following command will be used

```
tar cfv mydir.tar mydir
```

First the *tar* command is issued with the flags *cfv* which tells tar to create an archive (*c*), using the filename passed (*f*) and print out the files being added (*v*). Next the archive name is passed, in this case *mydir.tar* and finally the name of the directory to be archived (*mydir*).

3.6.3 Viewing the contents of a *tar* file

To view the contents of a *tar* archive the -t flag is used as follows

```
tar tf mydir.tar
```

Which will list out the contents of the *tar* file to the console.

3.6.4 adding to a *tar* file

To add more files to the *tar* file the *r* flag is used passing it the name of the directories to be added as follows

```
tar fvr mydir.tar newdir
```

This will then add the contents of the *newdir* directory to the archive *mydir.tar*.

3.6.5 Extracting a *tar* archive

To extract a *tar* file the *x* flag is used. When extracting a tar file the directory structure in the *tar* archive will be re-created. This means that any files in an existing directory with the same name as that of one in the archive will be replaced by default. To extract a *tar* archive the following commands are used

```
tar vfx mydir.tar
```

To ensure that files are not overwritten the *-k* flag may be used which will only extract files which are not in the directory, this is used as follows

```
tar vfxk mydir.tar
```

[2] Only works with gnu tar

3.6.6 Updating a *tar* file

To modify a *tar* file to include newer versions of a file already contained within the archive and add any newly created file the *u* flag is used as follows

```
tar -fvu mydir.tar newdir
```

3.7 Compressing files

As mentioned in section 3.6 a *tar* archive creates an archive by joining all files in the directory together into one file. These files can become very large and will take up lots of disk space. To make these files smaller compression may be used.

Under most *unix* systems there are many different compression tools which will compress files. The most common of these are *compress*, *gzip* and *bzip*2.

3.7.1 compress

The *compress* utility is found on most unix systems and will compress files using using adaptive Lempel-Ziv coding. Whenever possible, each file is replaced by one with the extension .Z, while keeping the same ownership modes, access and modification times.

To compress the *tar* archive *mydir.tar* the following commands will be used

```
compress mydir.tar
```

This will result in a file called *mydir.tar.Z* being created. To uncompress this file the *uncompress* utility may be used as follows

```
uncompress mydir.tar.Z
```

Which will result in the file *mydir.tar* which may then be extracted using the *tar* utility.

3.7.2 gzip / gunzip

A newer compression utility is the *gnu gzip* / *gunzip* set of tools. It uses Lempel-Ziv coding (LZ77). Whenever possible, each file is replaced by one with the extension *.gz*, while keeping the same ownership modes, access and modification times.

gunzip can also decompress files created by *gzip*, *zip*, *compress*, *compress -H* or pack. It will also work on some windows based *.zip*. files.

To compress a file using *gzip* the following command line is used

```
gzip mydir.tar
```

Which will result in the file *mydir.tar.gz.* To *unzip* the file the following command is used

```
gunzip mydir.tar.gz
```

3.7.3 bzip2 / bunzip2

bzip2 compresses files using the Burrows-Wheeler block sorting text compression algorithm, and Huffman coding. Compression is generally considerably better than that achieved by more conventional LZ77/LZ78-based compressors, and approaches the performance of the PPM family of statistical compressors.

The command-line options are deliberately very similar to those of GNU gzip, but they are not identical. *bzip2* expects a list of file names to accompany the command line flags. Each file is replaced by a compressed version of itself, with the name "original_name.bz2". Each compressed file has the same modification date, permissions, and, when possible, ownership as the corresponding original, so that these properties can be correctly restored at decompression time.

bzip2 attempts to guess the filename for the decompressed file from that of the compressed file as follows:

- filename.bz2 becomes filename
- filename.bz becomes filename
- filename.tbz2 becomes filename.tar
- filename.tbz becomes filename.tar
- anyothername becomes anyothername.out

If the file does not end in one of the recognised endings, *.bz2*, *.bz*, *.tbz2* or *.tbz*, *bzip2* complains that it cannot guess the name of the original file, and uses the original name with *.out* appended.

To compress a file using *bzip2* the following command line is used

```
bzip2 mydir.tar
```

Which will result in a file called *mydir.tar.bz2* being created. To extract this file the following command line is used

```
bunzip2 mydir.tar.bz2
```

3.7.4 .tgz files

It is quite common to get files with the extension .tgz this is a combined tar gzipped archive and may be uncompressed and extracted using one command as follows

```
tar vfxz GraphicsCode.tgz
```

To create a gzipped tar archive (useful for course work submission) change to the directory you wish to archive and do the following

```
tar vfcz MyArchive.tgz *
```

This will create the directory structure from the current file system position. If you wish to archive a directory just specify the directory name as follows

```
tar DirArchive.tgz MyDir
```

Chapter 4

Unix networking

Unix has a large variety of networking utilities and as with most Unix system most of the information about networks is held in a series of text files.

4.1 Exploring a network

The simplest way to explore a network is using the *ping* utility this uses the *ICMP* protocol's mandatory *ECHO_REQUEST* datagram to elicit an *ICMP ECHO_RESPONES* from a host or a gateway.

ping is used at the command line by typing *ping [host]* which will return the response [host] is alive. By use of command line options ping may be configured to send multiple requests to test the integrity of the network (*man ping* for more details)

ping is intended for use in network testing, measurement and management. Because of the load it can impose on the network, it is unwise to use ping during normal operations or from automated scripts.

4.1.1 Netstat

netstat is used to find information about the status of a network. It will give information on network connections, routing tables, interface statistics, and other networking information.

The different command line options for *netstat* differ in each Unix version so for current command line options read the man pages.

4.2 Remote login with ssh

ssh (Secure Shell) is a program for logging into a remote machine and for executing commands on a remote machine. It allows the user access to any machine on the network that they have login permission to and will let you execute commands (an applications) from the remote machine.

The first time ssh is run for a different machine it ask if you would like to add it to the list of machines available (just press return); the next time this is not asked for.

To login to another machine use the following

```
ssh rh1610
```

you will then be prompted for you password and then you will be logged into the remote machine at your home directory.

4.3 Copying files to different machine (scp)

The scp utility copies files between hosts on a network. It uses ssh for data transfer, and uses the same authentication and provides the same security as ssh.

To copy from a remote host to the current directory the following commands are used

```
scp-r rh1617:/transfer/jmacey/* ./
```

This will copy recursively the contents of the directory /transfer/jmacey from the machine rh1617 to the current directory on the local machine. You will be prompted for a password then all the files will be copied. rcp is similar to cp and uses most of the same syntax.

To copy from the current machine to a different machine the following syntax is used

```
scp-r * rh1617:/transfer/jmacey/
```

Appendix A

Unix Commands

Command 1: cd [dir]

Usage :
change directory

Flags :
no flags

Examples :
cd /etc changes to the etc directory
It must be noted that cd is built into the shell

Command 2: chmod [options] mode files

Usage :
change the access mode of one or more files. Only the owner of the file or a super user may change the mode

---

Flags :
-R recursively descend directory arguments whilst setting modes

File permissions are set on the basis of User Group and world and each section may have a Read Write and eXecute bit set. These are set using an octal number for each of the three groups. To set each bit the following values are used

4 Read
2 Write
1 Execute

A fourth bit may be set which precedes the User Group World flags. These use the following octal values

4 sets the user ID on execution
2 Set the group ID on execution
1 set sticky bit

Examples :
chmod 700 * set all files to have rwx permissions for owner and no permissions for group and world

chmod 755 * set file permissions to rwxr-xr-x for all files

Command 3: cp [options] file1 file2
cp [options] files directory

Usage :
Copy file1 to file2 or copy one or more files to the same names to a directory

---

Flags :
-i interactive mode (prompts for y/n for each file
-r recursively copy a directory, its files and subdirectories

---

Examples :
cp test.c test.c.old copy the file test.c to a new file test.c.old
cp * ./backup copy all the files in the current directory to a directory called backup

---

Command 4: du [options] [directories]

Usage :
prints the disk usage of the directory specified or present directory if not specified.

Flags :
-a print usage for all files not just subdirectories
-s printf on the grand total for cache named directory (i.e. silent mode)
-k print disk usage in K bytes not blocks

Examples :
du -ks print the total disk usage for the current directory

Command 5: find pathname(s) condition(s)

Usage :
Used to find files, find has numerous uses dependant upon the conditions set in the command line

---

Flags :
-exec command{} execute a unix command on finding a file
-name find a file with a specific name
-ok same as exec but prompts for y / n

Examples :
find ./ -name "*.c" finds all files with a .c extention
find ./ -name "*.o" -ok {} \;

Command 6: grep [options] [regexp] [files]

Usage :
search one or more files for lines that match the regular expression regexp

---

Flags :
-c print out a count of matched lines
-i ignore case

-l list file names not matched lines
-s suppress error messages

Examples :
grep main * find all files which contain the phrase main
grep -i myFunction *.c search for the text myFunction ignoring case in all files in the current directory

---

Command 7: gunzip [options] filename.gz

Usage :
unzip a GNU zipped file

Flags :
-l list contents but dont unzip file

Examples :
gunzip test.gz unzip the file test.gz

Command 8: gzip [options] filename.gz

Usage :
create a GNU zipped file

Flags :
-# 1 -9 compression ration 1 == fast 9 == best compression

Examples :
gzip -9 test.tar compress the file test.tar

Command 9: head [-n] [files]

Usage :
print the first n lines of a file

Flags :
-n number of lines to print

Examples :
head -n1 /etc/* prints the first line of every file in /etc

Command 10: lp / lpr [options] files

Usage :
sends files to print spooler

Flags :
-P [name] specifies the name of the printer
-#n number of copies to print

Examples :
lpr -P DrEvil notes.ps will print the file notes.ps

Command 11: ls [options] [names]

Usage :
List information obout files - current directory is used by default

Flags :
-l list in long format
-a show all files
-R list subdirectories recursively
-c list by file creation / modification time
-d show directories

Examples :
ls -al list all files in a directory
ls -lR list all files including subdirectories
ls -ld /bin /etc list the status of directories /bin and /etc
ls *.c list all of the .c files in the current directory

Command 12: mkdir [options] directories

Usage :
make a directory(s)

Flags :
-m mode used to set the access mode for the new directory
-p create parent directories as needed

Examples :
mkdir test creates a directory called test
mkdir -p /test/d1/old creates the whole directory structure
mkdir -m 700 test creates a directory called test with rwx—— permissions (see chmod for more details on permissions

Command 13: more [options] files

Usage :
Displays the named files in the console one screen at a time

Flags :
use the space key to scroll pages
PgUP moves up
PgDn moves down
q exits

Examples :
more /etc/passwd displays the contents of the /etc/passwd file

Command 14: mv [options] sources target

Usage :

mv is used to move or rename files.

Flags :
-i interactive mode prompt user y/n
-f force move even if target file exists

Examples :
mv file1.c file2.c renames file1.c file2.c
mv * ./backup moves all files to the backup directory
mv mydir myolddir rename a directory for mydir to myolddir

Command 15: pwd

Usage :
print working directory

Flags :
no flags

Examples :
pwd will print the current directory within the shell

Command 16: rm [options] files

Usage :
delete on or more files.

Flags :
-f force removal
-i interactive mode prompt y/n
-r recurse subdirectories

Examples :
rm *.c remove all c files from the current directory
rm -rf mydir remove contents of mydir as well as the directory itself
rm -rf * remove every thing in the current directory downward

Command 17: rmdir [options] directories

Usage :
remove directory

Flags :
-P recurse subdirectories
-s suppress standard error messages

Examples :
rmdir temp removes the temp directory

Command 18: tar [options] files

Usage :

create a tape archive (or a file in the current directory)

Flags :
v verbose mode
f specify file name and do not look for tape drive
c create a tape archive (tar file)
x extract an existing tar file

Examples :
tar cfv mydir.tar ./mydir/* create a tar file of the directory mydir called mydir.tar
tar vfx mydir.tar extracts the contents of the tar file mydir.tar

Appendix B

DOS to Unix translation

To	DOS	Unix
display a list of files	dir dir /w	ls ls -l
display contents of a file	type	cat <filename> more <filename>
display contents of a file with pause	type <filename> | more	more <filename>
copy file	copy <src> <dest>	cp <src> <dest>
find string in file	find	(f)grep “string” <filename>
compare files	comp	diff <file1> <file2>
rename files	rename or rn	mv <f1> <f2>
delete file	erase or del	rm <filename>
remove directories	rmdir or df	rmdir <dirname>
change file protection	attrib	chmod [flags] <filename>
create directories	mkdir or md	mkdir <dirname>
change working directory	chdir or cd	cd <dirname>
get help	help	man <topic>
display date,time	date time	date
display free disk space	chkdsk	df
print file	print	lpr <filename>

Index