Table of contents

Chapter 1	Unix history and Structure
Chapter 2	Vi Editor
Chapter 3	Directory Navigation and Control
Chapter 4	File Maintenance Commands
Chapter 5	Display Commands
Chapter 6	System Resources
Chapter 7	Shells
Chapter 8	Special Unix Features
Chapter 9	Other Useful Commands
Chapter 10	Shell Programming
Chapter – 11	System administration
Appendix	Unix Command Summary
Lab Exercises	

Chapter 1

Unix history and Structure

Introduction

Unix is a multi-user, multi-tasking, multi-processing, efficient, fast and very powerful operating system. It is very much like UNIX and behaves the same way. You can think of Unix as UNIX that is specially crafted to suit the desktop PC platform of today.

Unix started its life back in the early nineties as a college project of Linus Torvalds, then a student of Computer Science at the Helsinki University in Finland. From that somewhat uncertain start, Unix has grown today into a powerful operating system that is challenging top-of-the-line server products of big and established software companies. While learning Unix, you need to be aware of certain terms at the very beginning.

The core of an operating system is called the kernel, Linus Torvalds wrote the first Unix kernel almost single-handed, without borrowing a single line of code from any source. From the very beginning, he intended this operating system to be available freely to everybody. Hence, Unix is a free product. Here, the term free does not necessarily mean that it is free of cost. What it means actually is that it should be freely distributable along with source code, without any sort of copyright. The taker is free to modify the source code, and hence the kernel and re-distribute along the same principles. Till date, Linus Torvalds maintain the development of the Unix kernel, along with a host of contributors. The current kernel version is 2.2.

The kernel alone gives the operating system the bare minimum capabilities. The operating system is build on top of the kernel, and consists of additional commands, utilities, command interpreters or shells, language compilers and debuggers, text processors etc. The Free Software Foundation's (FSF) GNU project (GNU's Not Unix) involved thousands of software developers who co-operated across the Internet and wrote thousands of commands, utilities and tools. Unix supports all these GNU tools and commands, thus becoming a very rich and powerful operating system. Quite a few companies are combining the latest Unix kernel, all the free GNU commands, tools and utilities and their own setup program, configuration scripts etc. To form what are known as Unix distributions. The major Unix distributions are Red Hat, Caldera, SuSe, and Slackware. Red Hat Unix is the most commonly used in India.

The kernel:

The kernel is the heart of the any operating system. This is relatively small piece of code that directly sits on the hardware. It is a collection of programs that are mostly written in C. Every UNIX/UNIX system will have a kernel, it automatically gets loaded into memory as soon as the system is booted. (In other way booting UNIX is nothing but loading UNIX kernel into memory).

There are two types of basic kernel architecture; monolithic and microkernel.

Unix kernel is monolithic although its advanced use of kernel modules makes it some what of hybrid.

In monolithic kernel:

- The kernel functions are one big program running in kernel mode. Processes running in user space interact with kernel through well defined and limited set of system calls, in which arguments are passed on CPU registers.
- In contrast , all built-in kernel layers threads of execution have full access to the entire kernel APIs. Kernel subsystems interact with each other by calling functions with arguments passed on the stack , as any C- program.
- On the other hand kernel modules have access only to a more restricted set of exported functions.

In a micro kernel architecture :

- The kernel provides only a small set of functions , such as some synchronization facilities, an elementary scheduler, and inter process communication mechanisms.
- Important functions, like memory management, device drivers, system call handlers , etc run on the top of the micro kernel . these are integrated together.

Main kernel tasks:

Processes

Create/destroy
Schedule
IPC
Input/ Output

Memory

Virtual memory allocation/ free space management

File systems

Virtual file systems/native file systems. Maintaining inodes as well as data in a file

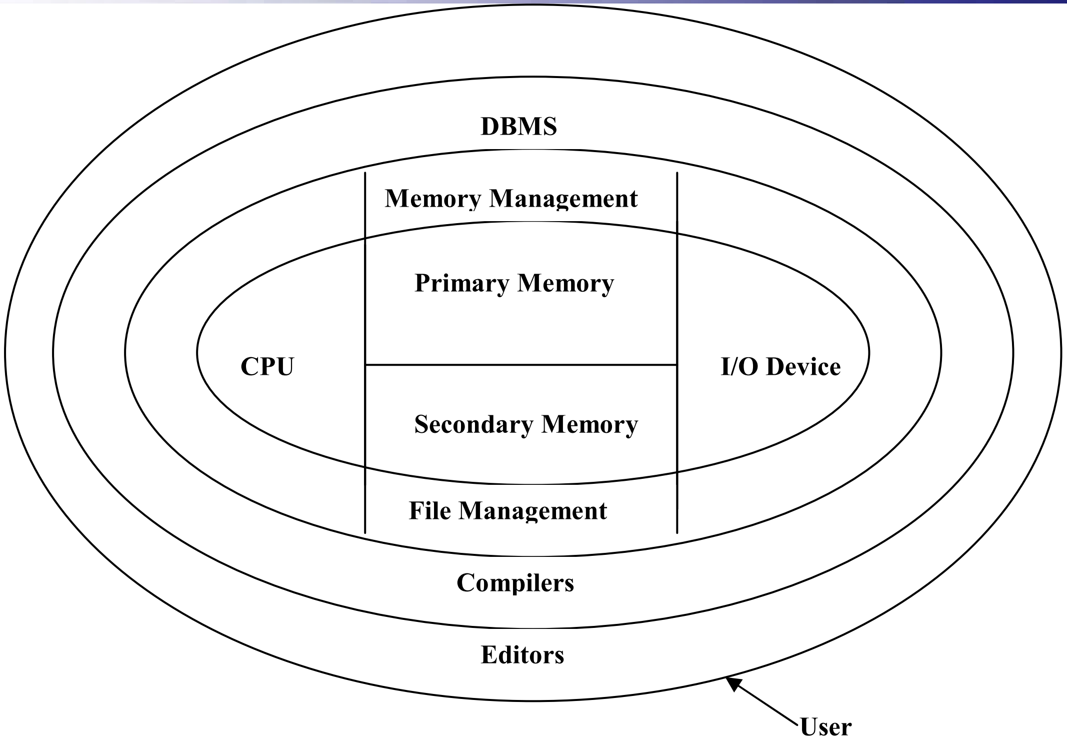
Devices

Device drivers to control all peripherals

Networking

Receive packets – identify and dispatch Sending packets – Routing and Address Resolution

Unix kernel architecture:



Welcome to Unix: logging in

To use Unix, you must have a user account with the operating system. This account will be identified by a name, either your name or whatever you choose and must have been already set up. This account name is called login name or user name. At the very beginning, Unix will prompt you to enter your login name with the following login prompt:

Red Hat Linux Release 9.0 publisher's Edition (Hedwig)

Kernel 2.4.20-1 on an i586

Local host login:

You should enter your login name at this prompt and press the Enter Key. Note here that Unix is a case-sensitive operating system. So, login name mohan will not be treated the same as Mohan, or MOHAN. Next, you will be prompted to enter your password. You must carefully choose your password. Ideally, the password should not be based on common dictionary words, should be a random combination of uppercase and lowercase letters and digits and should be at least 5 characters long. This will ensure that the password will be heard to break and your files and data will remain absolutely safe and private, as Unix does not permit unauthorized access to files, or even to directories.

There is a special account root that is reserved for the system administrator or super user. This account has unfettered to all system resources, including all files, directories, and data. This account must be used with extreme care. Even if you have installed the Unix system on your own PC, and you are the super user yourself, you should make a habit of creating a user account, and working with that account, till you are very proficient with Unix. The reason is that the super user has full access and control over every thing and a slight mistake can cause irreparable damage to the system.

The password you enter will not be echoed to the screen, for obvious reasons, if you have entered the user name and password correctly, you will get the shell prompt:

```
Last Login:  Thu Aug 5  17:48:17 on tty2

[Localhost@localdomain mohan]  $
```

The $ prompt indicates that the login was correct, and Unix is ready to accept commands from you. Here, we have used the account name mohan. If, on the other hand, the login was incorrect, either due to an incorrect user name or password, or both, Unix will refuse access to the system, and prompt for login again:

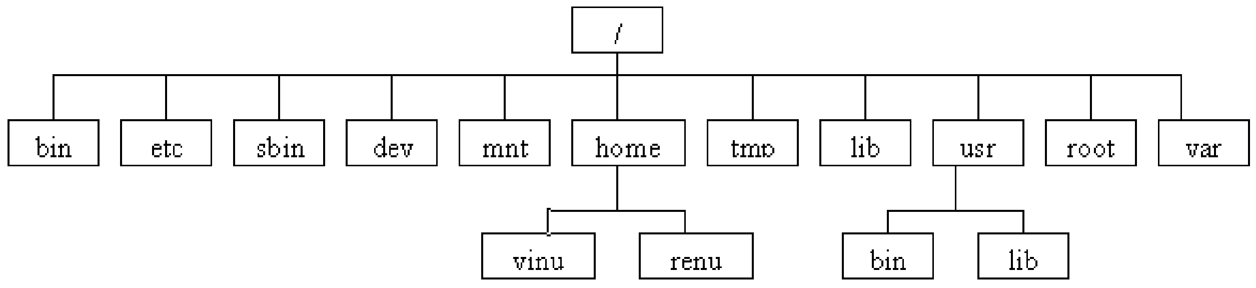
```
Login  incorrect

                    Login:
```

After a successful login, as the $ prompt comes, you can start issuing commands. Let us start with a few simple commands. Unix has a very rich collection of commands, and each command has a lot of command line switches to alter the behaviour of the command in subtle ways. Throughout our discussion, we shall be talking about the most commonly used commands and their most frequently used switches.

File Organization in Unix

During installation, Unix organizes all the files and directories neatly on the hard disk, following a norm generally referred to as Unix FSSTND (File system Standard). This defines the names of standard directories, and which type of file will go into which directory. This is in sharp contrast to the basically instructed way in which MS-DOS or Windows operate. There is a slight variation in file organization between Unix distributions, but most of it is common. Red Hat Unix strictly follows Unix FSSTND.

A Unix filesystem always starts with the root directory at the top (do not confuse the root directory with the root super user account; they are different entities). The root directory is represented by the forward slash character (/). All other directories and files come in levels below root in the directory hierarchy. The most important directories are shown in the figure below.



Note the following points about Unix filesystems:

The forward slash character (/) acts as the pathname separator. Thus the full pathname of the user2 directory in the above diagram will be /home/user2. The first forward slash indicates here that we are starting at the root.

The current directory (the directory the user is in at the moment) is represented by the dot character (.). The immediate parent directory is represented by the double dot characters (..). We will see these in the ls command in a moment.

Everything in Unix is treated as a file. These include ordinary files, directories, special files representing devices connected to the system like the keyboard, the monitor etc.

Filenames and directory names, as well as command names are case-sensitive. This means that the file names myfile and MyFile will be treated as different files.

Unix does not impose an 8.3 file naming convention. File names can be up to 256 characters long. Characters that are usually used are the lowercase and uppercase letters, digits, dash (-), underscore (_), and period (.). Note also that Unix does not treat the characters following a period as filename extension. Hence, you can have multiple periods within a file name. You must not use certain characters like *, /,;,?,",',` etc. as these have special meaning to Unix.

A brief description of the purpose of the most important directories follows:

/home This directory contains the home directories of all the user accounts in

the system. The home directory of a user will have the same name as his login name. Thus, the home directory of the user mohan will be /home/mohan. This home directory will be automatically set up when the user mohan is created by the system administrator, and as soon as mohan logs in, he will be placed in his home directory. Of course, he can change to other directories later. The home directory in Unix is represented by the tilde character (~). The super user's home directory is /root.

/bin Most of the executable files, like command files are kept here.

/sbin Important executable files needed by the system are kept here.

/lib Essential library files are kept here. These libraries are required by the system to operate properly.

/etc All system configuration script files are kept in this directory, and in sub-directories under this directory.

/dev This directory contains special device files, i.e., special files that represent devices. All devices needing access by the system must have entries in this directory.

/mnt This directory generally provides the mount point for external filesystems, like the floppy disk or the CD-ROM.

/tmp Temporary files are kept here.

/var Files with variable data are kept here. These files are maintained by Unix, and their contents keep changing to reflect the current state of the system.

/usr/bin This directory generally contains the executable files of different utilities that are not part of the core system.

/usr/lib Library files needed by the external utilities mentioned above are kept here.

Apart from the directories described above, Unix contains many other directories, which you should explore on your own.

After connecting with a Unix system, a user is prompted for a login username, then a password. The login username is the user's unique name on the system. The password is a changeable code known only to the user. At the login prompt, the user should enter the username; at the password prompt, the current password should be typed.

Note: Unix is case sensitive. Therefore, the login and password should be typed exactly as issued; the login, at least, will normally be in lower case.

Don't	use a word (or words) in any language
	use a proper name
	use information that can be found in your wallet
	use information commonly known about you (car license, pet name, etc)
	use control characters. Some systems can't handle them
	write your password anywhere
	ever give your password to *anybody*
Do	use a mixture of character types (alphabetic, numeric, special)
	use a mixture of upper case and lower case
	use at least 6 characters
	choose a password you can remember
	change your password often
	make sure nobody is looking over your shoulder when you are entering your password

Passwords

When your account is issued, you will be given an initial password. It is important for system and personal security that the password for your account be changed to something of your choosing. The command for changing a password is "*passwd*". You will be asked both for your old password and to type your new selected password twice. If you mistype your old password or do not type your new password the same way twice, the system will indicate that the password has not been changed.

Some system administrators have installed programs that check for appropriateness of password (is it cryptic enough for reasonable system security). A password change may be rejected by this program.

When choosing a password, it is important that it be something that could not be guessed -- either by somebody unknown to you trying to break in, or by an acquaintance who knows you. Suggestions for choosing and using a password follow:

Exiting

^D - indicates end of data stream; can log a user off. The latter is disabled

on many systems

^C - interrupt

Logout - leave the system

Exit - leave the shell

Control Keys

Control keys are used to perform special functions on the command line or within an editor. You type these by holding down the Control key and some other key simultaneously. This is usually represented as ^Key. Control-S would be written as ^S. With control keys upper and lower case are the same, so ^S is the same as ^s. This particular example is a stop signal and tells the terminal to stop accepting input. It will remain that way until you type a start signal, ^Q.

Control-U is normally the "line-kill" signal for your terminal. When typed it erases the entire input line.

In the *vi* editor you can type a control key into your text file by first typing ^V followed by the control character desired, so to type ^H into a document type ^V^H.

stty - terminal control

stty reports or sets terminal control options. The "tty" is an abbreviation that harks back to the days of teletypewriters, which were associated with transmission of telegraph messages, and which were models for early computer terminals.

For new users, the most important use of the *stty* command is setting the erase function to the appropriate key on their terminal. For systems programmers or shell script writers, the *stty* command provides an invaluable tool for configuring many aspects of I/O control for a given device, including the following:

- erase and line-kill characters

- data transmission speed

- parity checking on data transmission

- hardware flow control

- newline (NL) versus carriage return plus linefeed (CR-LF)

- interpreting tab characters

- edited versus raw input

- mapping of upper case to lower case

This command is very system specific, so consult the man pages for the details of the *stty* command on your system.

Syntax

stty [options]

Options

(none)	report the terminal settings
all	(or -a) report on all options
echoe	echo ERASE as BS-space-BS
dec	set modes suitable for Digital Equipment Corporation operating systems (which distinguishes between ERASE and BACKSPACE) (Not available on all systems)
kill	set the LINE-KILL character
erase	set the ERASE character
intr	set the INTERRUPT character

Examples

You can display and change your terminal control settings with the *stty* command. To display all (-a) of the current line settings:

```
% stty –a
```

```
speed 38400 baud, 24 rows, 80 columns
parenb -parodd cs7 -cstopb -hupcl cread -clocal -crtscts
-ignbrk brkint ignpar -parmrk -inpck istrip -inlcr -igncr icrnl -iuclc
ixon -ixany -ixoff imaxbel
isig iexten icanon -xcase echo echoe echok -echonl -noflsh -tostop
echoctl -echoprt echoke
opost -olcuc onlcr -ocrnl -onocr -onlret -ofill -ofdel
erase kill werase rprnt flush lnext susp intr quit stop eof
^H ^U ^W ^R ^O ^V ^Z/^Y ^C ^\ ^S/^Q ^D
```

You can change settings using *stty*, e.g., to change the erase character from ^? (the delete key) to ^H:

```
% stty erase ^H
```

This will set the terminal options for the current session only. To have this done for you automatically each time you login, it can be inserted into the .login or .bash_profile file that we'll look at later.

Getting Help

The Unix manual, usually called man pages, is available on-line to explain the usage of the Unix system and commands. To use a man page, type the command "*man*" at the system prompt followed by the command for which you need information.

Syntax

man [options] command_name

Common Options

-k	keyword list command synopsis line for all keyword matches
-M	path to man pages
-a	show all matching man pages (SVR4)

Examples

You can use *man* to provide a one line synopsis of any commands that contain the keyword that you want to search on with the "-k" option, e.g. to search on the keyword password, type:

```
% man -k password

    passwd (5)      - password file
    passwd (1)      - change password information
```

The number in parentheses indicates the section of the man pages where these references were found. You can then access the man page (by default it will give you the lower numbered entry, but you can use a command line option to specify a different one) with:

```
% man passwd

PASSWD(1)     USER COMMANDS     PASSWD(1)
NAME
        passwd - change password information
SYNOPSIS
        passwd [ -e login_shell ] [ username ]
DESCRIPTION
        passwd changes (or sets) a user's password.
        passwd prompts twice for the new password, without displaying
        it. This is to allow for the possibility of typing mistakes.
        Only the user and the super-user can change the user's password.
OPTIONS
        -e Change the user's login shell.
```

Here we’ve paraphrased and truncated the output for space and copyright concerns.

Chapter 2

Vi Editor

a. Using vi

To edit a file	vi [*filename*]
To recover an editing session	vi -r [*filename*]

Notes on vi commands and modal editing

All vi commands are entered in command mode. To enter command mode, press the ESC key. Some vi commands cause vi to enter another mode. For example, the **i** (insert command) causes vi to enter insert mode after which all keystrokes are inserted as text. To return to command mode from insert mode, press the ESC key. The **:set showmode** command will cause vi to display the current editing mode in the lower right corner of the editing screen.

Controlling The Screen Display of Your Session

Repaint the current screen	{ctrl-l}
Display line #, # of lines, etc..	{ctrl-g}

Moving the Cursor

Beginning of current line	0 or ^
Beginning of first screen line	H
Beginning of last screen line	L
Beginning of middle screen line	M
Down one line	j, {return}, +
End of current line	$
Left one character	h, {ctrl-h}
Left to beggining of word	b, B
Right one character	l, {space}
Right to end of word	e, E
Right to beginning of word	w, W
Up one line	k, -
Beginning of next sentence	)
Beginning of previous sentence	(

Paging Through Text

Back one screen	{ctrl-b}
Down half a screen	{ctrl-d}
Down one screen	{ctrl-f}
Forware to end of file	G
Move cursor to specified line	*line no.* G
Up half a screen	{ctrl-j}

Special Pattern Characters

Beginning of line	^
End of line	$
Any character except newline	.
Any number of the preceding character	*

Any set of characters (except newline)	`.*`

Searching Through Text

Backward for pattern	?*pattern*
Forward for pattern	/*pattern*
Repeat previous search	n
Reverse direction of previous search	N
Show *all* lines containing pattern	:beg,endg/pattern/p

:1,$g/compiler/p Will print all lines with the pattern **compiler**.

Substitute patt2 for all patt1 found.	:beg,ends/patt1/patt2/g

:%s/notfound/found/g Will change all occurences of notfound to found.

Creating Text

Append text after cursor	a
Append text after end of line	A
Insert text before cursor	i
Insert text at beginning of line	I
Open new line after current line	o
Open new line before current line	O
Take next character literally (*i.e. control characters...*) and display it	{ctrl-v}

Modifying Text

Change current word	cw, cW
Change current line (cursor to end)	C
Delete character (cursor forward)	x
Delete character (before cursor)	X
Delete word	dw, dW
Delete line	dd
Delete text to end of line	D
Duplicate text	*(use yank and put)*
Join current line with next line	J
Move text	*(use delete and put)*
Put buffer text after/below cursor	p
Put buffer text before/above cursor	P
Repeat last modification command	.
Replace current character	r
Replace text to end of line	R
Substitute text for character	s
Undo your previous command	u
Transpose characters	xp
Yank (copy) word into buffer	yw
Yank (copy) current line into buffer	Y

Making Corrections During Text Insertions

Overwrite last character	{delete}
Overwrite last word	{ctrl-w}

Ending Your Editing Sessions

Quit (no changes made)	:q

```
   Quit and save changes              ZZ, :wq
   Quit and discard changes           :q!
```

Using ex Commands From Within vi

```
   Copy specified lines                       :co, t
   Display line numbers                       :set nu
   Disable display of line numbers   :set nonu
   Move lines after specified line   :m
   Read file in after specified line :r filename
   Review current editor options     :set
   Review editor options             :set all
   Set new editor option             :set option
   Write changes to original file    :w
   Write to specified file           :w filename
   Force write to a file             :w! filename
```

Some Useful ex commands for use in vi

Some useful set options for your ~/.exrc file:

```
   :set all           Display all Set options
   :set autoindent    Automagically indent following lines to the indentation
                        of previous line.
   :set ignorecase    Ignore case during pattern matching.
   :set list          Show special characters in the file.
   :set number                Display line numbers.
   :set shiftwidth=n  Width for shifting operators << and >>
   :set showmode      Display mode when in Insert, Append, or Replace mode.
   :set wrapmargin=n  Set right margin 80-n for autowrapping lines
                        (inserting newlines).  0 turns it off.
```

Chapter 3

Directory Navigation and Control

The Unix file system is set up like a tree branching out from the root. The root directory of the system is symbolized by the forward slash (/). System and user directories are organized under the root. The user does not have a root directory in Unix; users generally log into their own home directory. Users can then create other directories under their home. The following table summarizes some directory navigation commands.

Command/Syntax	**What it will do**
cd [directory]	Change directory
ls [options] [directory or file]	List *directory* contents or *file* permissions
mkdir [options] directory	Make a *directory*
pwd	Print working (current) directory
rmdir [options] directory	Remove directory

If you're familiar with DOS the following table comparing similar commands might help to provide the proper reference frame.

Command	**Unix**	**DOS**
list directory contents	ls	dir
make directory	mkdir	md & mkdir
change directory	cd	cd & chdir
delete (remove) directory	rmdir	rd & rmdir
return to user's home directory	cd	cd\
location in path (present working directory)	pwd	cd

pwd - print working directory

At any time you can determine where you are in the file system hierarchy with the *pwd*, print working directory, command, e.g.:

```
% pwd
    /home/frank/src
```

cd - change directory

You can change to a new directory with the *cd*, change directory, command. *cd* will accept both absolute and relative path names.

Syntax

cd [directory]

Examples

cd (also *chdir* in some shells)

change directory

cd	changes to user's home directory
cd /	changes directory to the system's root
cd ..	goes up one directory level
cd ../..	goes up two directory levels

cd /full/path/name/from/root

changes directory to absolute path named (note the leading slash)

cd path/from/current/location

changes directory to path relative to current location (no leading slash)

cd ~username/directory

changes directory to the named username's indicated directory

(Note: the ~ is not valid in the Bash shell;)

mkdir - make a directory

You extend your home hierarchy by making sub-directories underneath it. This is done with the *mkdir*, make directory, command. Again, you specify either the full or relative path of the directory:

Syntax

mkdir [options] directory

Common Options

-p create the intermediate (parent) directories, as needed

-m mode access permissions (SVR4)

Examples

```
% mkdir /home/frank/data
```

or, if your present working directory is /home/frank the following would be equivalent:

```
% mkdir data
```

rmdir - remove directory

A directory needs to be empty before you can remove it. If it's not, you need to remove the files first. Also, you can't remove a directory if it is your present working directory; you must first change out of it.

Syntax

rmdir directory

Examples

To remove the empty directory /home/frank/data while in /home/frank use:

```
% rmdir data
```

or

```
% rmdir /home/frank/data
```

ls - list directory contents

The command to list your directories and files is *ls*. With options it can provide information about the size, type of file, permissions, dates of file creation, change and access.

Syntax

ls [options] [argument]

Common Options

When no argument is used, the listing will be of the current directory. There are many very useful options for the ls command. A listing of many of them follows. When using the command, string the desired options together preceded by "-".

-a lists all files, including those beginning with a dot (.).

-d lists only names of directories, not the files in the directory

-F indicates type of entry with a trailing symbol:

directories /sockets =symbolic links@executables *

-g displays Unix group assigned to the file, requires the -l option (BSD only) -or- on an SVR4 machine, e.g. Solaris, this option has the opposite effect

-L if the file is a symbolic link, lists the information for the file or directory the link references, not the information for the link itself

-l long listing: lists the mode, link information, owner, size, last modification (time). If the file is a symbolic link, an arrow (-->) precedes the pathname of the linked-to file.

The mode field is given by the -l option and consists of 10 characters. The first character is one of the following:

character	**if entry is a**
d	directory - plain file
b	block-type special file
c	character-type special file
l	symbolic link
s	socket

The next 9 characters are in 3 sets of 3 characters each. They indicate the file access permissions: the first 3 characters refer to the permissions for the user, the next three for the users in the Unix group assigned to the file, and the last 3 to the permissions for other users on the system. Designations are as follows:

r	read permission
w	write permission
x	execute permission
-	no permission

There are a few less commonly used permission designations for special circumstances. These are explained in the man page for *ls*.

Examples

To list the files in a directory:

```
% ls
demofiles frank Linda
```

To list all files in a directory, including the hidden (dot) files try:

```
% ls -a
. .cshrc .history .plan .rhosts frank
```

```
    .. .emacs .login .bash_profile demofiles linda
```

To get a long listing:

```
    % ls -al
```

```
total 24
drwxr-sr-x    5   workshop   acs   512   Jun 7 11:12     .
drwxr-xr-x    6   root       sys   512   May 29 09:59    ..
-rwxr-xr-x    1   workshop   acs   532   May 20 15:31    .cshrc
-rw-------    1   workshop   acs   525   May 20 21:29    .emacs
-rw-------    1   workshop   acs   622   May 24 12:13    .history
-rwxr-xr-x    1   workshop   acs   238   May 14 09:44    .login
-rw-r--r--    1   workshop   acs   273   May 22 23:53    .plan
-rwxr-xr-x    1   workshop   acs   413   May 14 09:36    .bash_profile
-rw-------    1   workshop   acs   49    May 20 20:23    .rhosts
drwx------    3   workshop   acs   512   May 24 11:18    demofiles
drwx------    2   workshop   acs   512   May 21 10:48    frank
drwx------    3   workshop   acs   512   May 24 10:59    linda
```

Chapter 4

File Maintenance Commands

To create, copy, remove and change permissions on files you can use the following commands. If you're familiar with DOS the following table comparing similar commands might help to provide the proper reference frame.

Command/Syntax	**What it will do**
chgrp [options] *group file*	change the group of the file
chmod [options] *file*	change file or directory access permissions
chown [options] *owner file*	change the ownership of a file; can only be done by the superuser
cp [options] *file1 file2*	copy *file1* into *file2*; *file2* shouldn't already exist. This command creates or overwrites *file2*.
mv [options] *file1 file2*	move *file1* into *file2*
rm [options] *file*	remove (delete) a file or directory (-r recursively deletes the directory and its contents) (-i prompts before removing files)

Command	**Unix**	**DOS**
Copy file	*cp*	*copy*
move file	*mv*	*move (not supported on all versions of DOS)*
rename file	*mv*	*rename & ren*
delete (remove) file	*rm*	*erase & del*

cp - copy a file

Copy the contents of one file to another with the *cp* command.

Syntax

cp [options] old_filename new_filename

Common Options

-i	interactive (prompt and wait for confirmation before proceeding)
-r	recursively copy a directory

Examples

% cp old_filename new_filename

You now have two copies of the file, each with identical contents. They are completely independent of each other and you can edit and modify either as needed. They each have their own inode, data blocks, and directory table entries.

mv - move a file

Rename a file with the move command, *mv*.

Syntax

mv [options] old_filename new_filename

Common Options

-i	interactive (prompt and wait for confirmation before proceeding)
-f	don't prompt, even when copying over an existing target file (overrides -i)

Examples

% mv old_filename new_filename

You now have a file called new_filename and the file old_filename is gone. Actually all you've done is to update the directory table entry to give the file a new name. The contents of the file remain where they were.

rm - remove a file

Remove a file with the *rm*, remove, command.

Syntax

rm [options] filename

Common Options

-i interactive (prompt and wait for confirmation before proceeding)

-r recursively remove a directory, first removing the files and subdirectories

beneath it

-f don't prompt for confirmation (overrides -i)

Examples

% rm old_filename

A listing of the directory will now show that the file no longer exists. Actually, all you've done is to remove the directory table entry and mark the inode as unused. The file contents are still on the disk, but the system now has no way of identifying those data blocks with a file name. There is no command to "unremove" a file that has been removed in this way. For this reason many novice users alias their remove command to be "*rm -i*", where the -i option prompts them to answer yes or no before the file is removed. Such aliases are normally placed in the .cshrc file for the C shell.

File Permissions

Each file, directory, and executable has permissions set for who can read, write, and/or execute it. To find the permissions assigned to a file, the *ls* command with the -l option should be used. Also, using the -g option with "*ls -l*" will help when it is necessary to know the group for which the permissions are set (BSD only).

When using the "*ls -lg*" command on a file (*ls -l* on SysV), the output will appear as follows:

```
-rwxr-x--- user Unixgroup size Month nn hh:mm filename
```

The area above designated by letters and dashes (-rwxr-x---) is the area showing the file type and permissions as defined in the previous Section. Therefore, a permission string, for example, of -rwxr-x--- allows the user (owner) of the file to read, write, and execute it; those in the Unixgroup of the file can read and execute it; others cannot access it at all.

chmod - change file permissions

The command to change permissions on an item (file, directory, etc) is *chmod* (change mode). The syntax involves using the command with three digits (representing the user (owner, u) permissions, the group (g) permissions, and other (o) user's permissions) followed by the argument (which may be a file name or list of files and directories). Or by using symbolic representation for the permissions and who they apply to.

Each of the permission types is represented by either a numeric equivalent:

read=4, write=2, execute=1

or a single letter:

read=r, write=w, execute=x

A permission of 4 or r would specify read permissions. If the permissions desired are read and write, the 4 (representing read) and the 2 (representing write) are added together to make a permission of 6.Therefore, a permission setting of 6 would allow read and write permissions.

Alternatively, you could use symbolic notation which uses the one letter representation for who and for the permissions and an operator, where the operator can be:

+	add permissions
-	remove permissions
=	set permissions

So to set read and write for the owner we could use "u=rw" in symbolic notation.

Syntax

chmod nnn [argument list]		numeric mode

chmod [who]op[perm] [argument list]		symbolic mode

where nnn are the three numbers representing user, group, and other permissions, who is any of u, g, o, or a (all) and perm is any of r, w, x. In symbolic notation you can separate permission specifications by commas, as shown in the example below.

Common Options

-f	force (no error message is generated if the change is unsuccessful)
-R	recursively descend through the directory structure and change the modes

Examples

If the permission desired for file1 is user: read, write, execute, group: read, execute, other: read, execute, the command to use would be

chmod 755 file1	or	*chmod u=rwx,go=rx file1*

Reminder: When giving permissions to group and other to use a file, it is necessary to allow at least execute permission to the directories for the path in which the file is located. The easiest way to do this is to be in the directory for which permissions need to be granted:

chmod 711 .	or	*chmod u=rw,+x .*	or	*chmod u=rwx,go=x .*

where the dot (.) indicates this directory.

chown - change ownership

Ownership of a file can be changed with the *chown* command. On most versions of Unix this can only be done by the super-user, i.e. a normal user can’t give away ownership of their files. *chown* is used as below, where # represents the shell prompt for the super-user:

Syntax

chown [options] user[:group] file (SVR4)

chown [options] user[.group] file (BSD)

Common Options

-R	recursively descend through the directory structure
-f	force, and don’t report any errors

Examples

```
# chown new_owner file
```

chgrp - change group

Anyone can change the group of files they own, to another group they belong to, with the *chgrp* command.

Syntax

chgrp [options] group file

Common Options

-R	recursively descend through the directory structure
-f	force, and don’t report any errors

Examples

```
% chgrp new_group file
```

Chapter 5

Display Commands

There are a number of commands you can use to display or view a file. Some of these are editors, which we will look at later. Here we will illustrate some of the commands normally used to display a file.

Command/Syntax	**What it will do**
cat [options] *file*	concatenate (list) a file
echo [text string]	echo the text string to stdout
head [-number] *file*	display the first 10 (or number of) lines of a file
more (or *less* or *pg*) [options] *file*	page through a text file
tail [options] *file*	display the last few lines (or parts) of a file

echo - echo a statement

The *echo* command is used to repeat, or echo, the argument you give it back to the standard output device. It normally ends with a line-feed, but you can specify an option to prevent this.

Syntax

echo [string]

Common Options

-n	-don’t print <new-line> (BSD, shell built-in)
\c	-don’t print <new-line> (SVR4)
\n	-where n is the 8-bit ASCII character code (SVR4)
\t	-tab (SVR4)
\f	-form-feed (SVR4)
\n	-new-line (SVR4)
\v	-vertical tab (SVR4)

Examples

```
% echo Hello Class          or      echo "Hello Class"
```

To prevent the line feed:

```
% echo -n Hello Class  or      echo "Hello Class \c"
```

where the style to use in the last example depends on the *echo* command in use. The \x options must be within pairs of single or double quotes, with or without other string characters.

cat - concatenate a file

Display the contents of a file with the concatenate command, *cat*.

Syntax

cat [options] [file]

Common Options

-n	precede each line with a line number
-v	display non-printing characters, except tabs, new-lines, and form-feeds
-e	display $ at the end of each line (prior to new-line) (when used with -v option)

Examples

```
% cat filename
```

You can list a series of files on the command line, and *cat* will concatenate them, starting each in turn, immediately after completing the previous one, e.g.:

```
% cat file1 file2 file3
```

more, less, and pg - page through a file

more, *less*, and *pg* let you page through the contents of a file one screenful at a time. These may not all be available on your Unix system. They allow you to back up through the previous pages and search for words, etc.

Syntax

more [options] [+/pattern] [filename]

less [options] [+/pattern] [filename]

pg [options] [+/pattern] [filename]

Options

more	less	pg	Action
-c	-c	-c	clear display before displaying
	-i		ignore case
-w	default	default	don’t exit at end of input, but prompt and wait
-lines	-lines	# of lines/screenful	
+/pattern	+/pattern	+/pattern	search for the pattern

Internal Controls

more	displays (one screen at a time) the file requested
<space bar>	to view next screen
<return> or *<CR>*	to view one more line
q	to quit viewing the file
h	help
b	go back up one screenful
/word	search for word in the remainder of the file
	See the man page for additional options
less	similar to *more*; see the man page for options
pg	the SVR4 equivalent of *more* (page)

head - display the start of a file

head displays the head, or start, of the file.

Syntax

head [options] file

Common Options

-n number number of lines to display, counting from the top of the file

-number	same as above

Examples

By default *head* displays the first 10 lines. You can display more with the "-n number", or"-number" options, e.g., to display the first 40 lines:

```
% head -40 filename          or          head -n 40 filename
```

tail - display the end of a file

tail displays the tail, or end, of the file.

Syntax

tail [options] file

Common Options

-number	number of lines to display, counting from the bottom of the file

Examples

The default is to display the last 10 lines, but you can specify different line or byte numbers, or a different starting point within the file. To display the last 30 lines of a file use the -number style:

```
% tail -30 filename
```

Chapter 6

System Resources

System Resource Commands

Command/Syntax	**What it will do**
date [options]	report the current date and time
df [options] [resource]	report the summary of disk blocks and inodes free and in use
du [options] [*directory* or *file*]	report amount of disk space in use+
hostname/uname	display or set (super-user only) the name of the current machine
kill [options] [-SIGNAL] [pid#] [%job]	send a signal to the process with the process id number (pid#) or job control number (%n). The default signal is to kill the process.
man [options] *command*	show the manual (man) page for a command
passwd [options]	set or change your password
ps [options]	show status of active processes
script file	saves everything that appears on the screen to file until *exit* is executed
stty [options]	set or display terminal control options
whereis [options] *command*	report the binary, source, and man page locations for the command named
which command	reports the path to the command or the shell alias in use
who or *w*	report who is logged in and what processes are running

df - summarize disk block and file usage

df is used to report the number of disk blocks and inodes used and free for each file system. The output format and valid options are very specific to the OS and program version in use.

Syntax

df [options] [resource]

Common Options

-l	local file systems only (SVR4)
-k	report in kilobytes (SVR4)

Examples

```
$ df
Filesystem  Kbytes   used     avail    capacity  Mounted on
/dev/hda0   20895     19224   0        102%      /
/dev/hda1   319055   131293   155857   46%       /usr
/dev/hdb0   637726   348809   225145   61%       /usr/local
/dev/hdb1   240111   165489   50611    77%
/home/sachin
peri:/usr/local/backup       1952573  976558   780758    56%
```

du - report disk space in use

du reports the amount of disk space in use for the files or directories you specify.

Syntax

du [options] [directory or file]

Common Options

-a	display disk usage for each file, not just subdirectories
-s	display a summary total only
-k	report in kilobytes (SVR4)

Examples

```
     $ du
1    ./.elm
1    ./Mail
1    ./News
20  ./uc
86.
     $ du -a uc
7    uc/Unixgrep.txt
5    uc/editors.txt
1    uc/.emacs
1    uc/.exrc
4    uc/telnet.ftp
1    uc/uniq.tee.txt
20  uc
```

ps - show status of active processes

ps is used to report on processes currently running on the system. The output format and valid options are very specific to the OS and program version in use.

Syntax

ps [options]

Common Options

UNIX / BSD	SVR4	
-a	-e	all processes, all users
-e		environment/everything
-g		process group leaders as well
-l	-l	long format
-u	-u user	user oriented report
-x	-e	even processes not executed from terminals
-f		full listing
-w		report first 132 characters per line

Note -- Because the *ps* command is highly system-specific, it is recommended that you consult the man pages of your system for details of options and interpretation of *ps* output.

Examples

```
    $       ps
PIDTT       STAT    TIME    COMMAND
15549       p0      IW      0:00    -tcsh (tcsh)
15588       p0      IW      0:00     man nice
15594       p0      IW      0:00     sh -c less /tmp/man15588
15595       p0      IW      0:00     less /tmp/man15588
15486       p1      S       0:00     -tcsh (tcsh)
15599       p1      T       0:00     emacs Unixgrep.txt
15600       p1      R       0:00     ps
```

kill - terminate a process

kill sends a signal to a process, usually to terminate it.

Syntax

kill [-signal] process-id

Common Options

-l	displays the available kill signals:

The -KILL signal, also specified as -9 (because it is 9th on the above list), is the most commonly used *kill* signal. Once seen, it can't be ignored by the program whereas the other signals can.

```
$   kill -9 15599
    [1] + Killed            emacs Unixgrep.txt
```

who - list current users

who reports who is logged in at the present time.

Syntax

who [am i]

Examples

```
    $       who
Sandhya     ttyp1   Apr 21 20:15    (apple.acs.ohio-s)
Vinu                ttyp2   Apr 21 23:21    (worf.acs.ohio-st)
Vikas               ttyp3   Apr 21 23:22    (127.99.25.8)
Sheshadri   ttyp4   Apr 2122:27     (slip1-61.acs.ohi)
Sharath     ttyp5   Apr 21 23:07    (picard.acs.ohio-)
```

```
Kavitha     ttyp6   Apr 21 23:00   (ts31-4.homenet.o)
Mallesh     ttyp7   Apr 21 23:24   (data.acs.ohio-st)
Mahantesh          ttyp8   Apr 21 23:32   (slip3-10.acs.ohi)
Harsha             ttypc   Apr 21 23:38   (lcondron-mac.acs)
Raju               ttype   Apr 21 22:30   (slip3-36.acs.ohi)
Raghu              ttyq2   Apr 21 21:12   (ts24-10.homenet.)
```

```
    $ who am I
            Vinu  ttyp2 Apr 21 23:38 (lcondron-mac.acs)
```

whereis - report program locations

whereis reports the filenames of source, binary, and manual page files associated with command(s).

Syntax

whereis [options] command(s)

Common Options

-b	report binary files only
-m	report manual sections only
-s	report source files only

Examples

```
    $ whereis     mail
            Mail: /usr/ucb/Mail /usr/lib/Mail.help /usr/lib/Mail.rc /
usr/man/man1/Mail.1
    $ whereis -b mail
            Mail: /usr/ucb/Mail /usr/lib/Mail.help /usr/lib/Mail.rc
    $ whereis -m mail
            Mail: /usr/man/man1/Mail.1
```

which - report the command found

which will report the name of the file that is be executed when the command is invoked. This will be the full path name or the alias that's found first in your path.

Syntax

which command(s)

example

```
$ which Mail
        /usr/ucb/Mail
```

hostname/uname - name of machine

hostname (*uname -n* on SysV) reports the host name of the machine the user is logged into, e.g.:

```
$ hostname
        Vsquare comp1
```

uname has additional options to print information about system hardware type and software version.

date - current date and time

date displays the current data and time. A superuser can set the date and time.

Syntax

date [options] [+format]

Common Options

-u	use Universal Time (or Greenwich Mean Time)
+format	specify the output format
%a	weekday abbreviation, Sun to Sat
%h	month abbreviation, Jan to Dec
%j	day of year, 001 to 366
%n	<new-line>
%t	<TAB>
%y	last 2 digits of year, 00 to 99
%D	MM/DD/YY date
%H	hour, 00 to 23
%M	minute, 00 to 59
%S	second, 00 to 59
%T	HH:MM:SS time

Examples

```
    $date

            Mon Jun 10 09:01:05 EDT 1996

$date –u

            Mon Jun 10 13:01:33 GMT 1996

    $date +%a%t%D

            Mon 06/10/96

$date '+%y:%j'

            96:162
```

Chapter 7

Shells

The Shell sits between you and the operating system, acting as a command interpreter. It reads your terminal input and translates the commands into actions taken by the system. The shell is analogous to *command.com* in DOS. When you log into the system you are given a default shell. When the shell starts up it reads its startup files and may set environment variables, command search paths, and command aliases, and executes any commands specified in these files.

The original shell was the Bash shell, *bash*. Every Unix platform will either have the Bash shell, or a Bash compatible shell available. It has very good features for controlling input and output, but is not well suited for the interactive user. To meet the latter need the C shell, *csh*, was written and is now found on most, but not all, Unix systems. It uses C type syntax, the language Unix is written in, but has a more awkward input/output implementation. It has job control, so that you can reattach a job running in the background to the foreground. It also provides a history feature, which allows you to modify and repeat previously executed commands.

The default prompt for the Bash shell is $ (or #, for the root user). The default prompt for the C shell is%.

Numerous other shells are available from the network. Almost all of them are based on either *sh* or *csh* with extensions to provide job control to *sh*, allow in-line editing of commands, page through previously executed commands, provide command name completion and custom prompt, etc. Some of the more well known of these may be on your favorite Unix system: the Korn shell, *ksh*, by David Korn and the Bash Again SHell, *bash*, from the Free Software Foundations GNU project, both based on *sh*, the T-C shell, *tcsh*, and the extended C shell, *cshe*, both based on *csh*. Below we will describe some of the features of *sh* and *csh* so that you can get started.

Built-in Commands

The shells have a number of built-in, or native commands. These commands are executed directly in the shell and don’t have to call another program to be run. These built-in commands are different for the different shells.

bash

For the Bash shell some of the more commonly used built-in commands are:

:	null command
.	source (read and execute) commands from a file
case	Case conditional loop
cd	change the working directory (default is $HOME)
echo	write a string to standard output
eval	evaluate the given arguments and feed the result back to the shell
exec	execute the given command, replacing the current shell
exit	Exit the current shell
export	share the specified environment variable with subsequent shells
for	for conditional loop
if	if conditional loop
pwd	Print the current working directory
read	Read a line of input from stdin
set	set variables for the shell
test	evaluate an expression as true or false
trap	trap for a typed signal and execute commands
umask	set a default file permission mask for new files
unset	unset shell variables
wait	Wait for a specified process to terminate
while	while conditional loop

Environment Variables

Environmental variables are used to provide information to the programs you use. You can have both global environment and local shell variables. Global environment variables are set by your login shell and new programs and shells inherit the environment of their parent shell. Local shell variables are used only by that shell and are not passed on to other processes. A child process cannot pass a variable back to its parent process.

The current environment variables are displayed with the "*env*" or "*printenv*" commands. Some common ones are:

• DISPLAY	The graphical display to use, e.g. nyssa:0.0
• EDITOR	The path to your default editor, e.g. /usr/bin/vi
•GROUP	Your login group, e.g. Staff
• HOME	Path to your home directory, e.g. /home/frank
• HOST	The hostname of your system, e.g. Nyssa
• IFS	Internal field separators, usually any white space (defaults to tab, space and <newline>)
• LOGNAME	The name you login with, e.g. frank
• PATH	Paths to be searched for commands, e.g. /usr/bin:/usr/ucb:/usr/local/bin
• PS1	The primary prompt string, Bash shell only (defaults to $)
• PS2	The secondary prompt string, Bash shell only (defaults to >)
• SHELL	The login shell you're using, e.g. /usr/bin/csh
• TERM	Your terminal type, e.g. xterm
• USER	Your username, e.g. frank

Many environment variables will be set automatically when you login. You can modify them or define others with entries in your startup files or at anytime within the shell. Some variables you might want to change are PATH and DISPLAY. The PATH variable specifies the directories to be automatically searched for the command you specify. Examples of this are in the shell startup scripts below.

You set a global environment variable with a command similar to the following for the C shell:

% setenv NAME value and for Bash shell:

```
$ NAME=value; export NAME
```

You can list your global environmental variables with the *env* or *printenv* commands. You unset them with the *unsetenv* (C shell) or *unset* (Bash shell) commands.

To set a local shell variable use the *set* command with the syntax below for C shell. Without options *set* displays all the local variables.

```
% set name=value
```

For the Bash shell set the variable with the syntax:

```
$ name=value
```

The current value of the variable is accessed via the "$name", or "${name}", notation.

The Bash Shell, bash

bash*(Bourne again shell)* uses the startup file .bash_profile in your home directory. There may also be a system-wide startup file, e.g. /etc/profile. If so, the system-wide one will be sourced (executed) before your local one.

A simple .bash_profile could be the following:

```
PATH=/usr/bin:/usr/ucb:/usr/local/bin:.       # set the PATH
export PATH                                   # so that PATH is available to subshells
# Set a prompt
PS1="{`hostname` `who am i`} "                # set the prompt, default is "$"
# functions
ls() { /bin/ls -sbF "$@";}
ll() { ls -al "$@";}
# Set the terminal type
stty erase ^H                                 # set Control-H to be the erase key
eval `tset -Q -s -m ':?xterm'`                # prompt for the terminal type,
                                   assume xterm
#
umask 077
```

Whenever a # symbol is encountered the remainder of that line is treated as a comment. In the PATH variable each directory is separated by a colon (:) and the dot (.) specifies that the current directory is in your path. If the latter is not set it’s a simple matter to execute a program in the current directory by typing:

./program_name

It’s actually a good idea not to have dot (.) in your path, as you may inadvertently execute a program you didn’t intend to when you *cd* to different directories.

A variable set in .bash_profile is set only in the login shell unless you "*export*" it or source .bash_profile from another shell. In the above example PATH is exported to any subshells. You can source a file with the built-in "." command of *sh*, i.e.:

$ . ./.bash_profile

You can make your own functions. In the above example the function *ll* results in an "l*s -al*" being done on the specified files or directories.

With ***stty*** the erase character is set to Control-H (^H), which is usually the Backspace key.

The ***tset*** command prompts for the terminal type, and assumes "xterm" if we just hit <CR>. This command is run with the shell built-in, *eval*, which takes the result from the tset command and uses it as an argument for the shell. In this case the "-s" option to tset sets the TERM and TERMCAP variables and exports them.

The last line in the example runs the *umask* command with the option such that any files or directories you create will not have read/write/execute permission for group and other.

For further information about *sh* type "***man sh***" at the shell prompt.

Job Control

With the C shell, *csh*, and many newer shells including some newer Bash shells, you can put jobs into the background at anytime by appending "&" to the command, as with *sh*. After submitting a command you can also do this by typing ^Z (Control-Z) to suspend the job and then "*bg*" to put it into the background. To bring it back to the foreground type "*fg*".

You can have many jobs running in the background. When they are in the background they are no longer connected to the keyboard for input, but they may still display output to the terminal, interspersing with whatever else is typed or displayed by your current job. You may want to redirect I/O to or from files for the job you intend to background. Your keyboard is connected only to the current, foreground, job.

The built-in *jobs* command allows you to list your background jobs. You can use the *kill* command to kill a background job. With the %n notation you can reference the nth background job with either of these commands, replacing n with the job number from the output of *jobs*. So kill the second background job with "*kill %2*" and bring the third job to the foreground with "*fg %3*".

History

The Bash shell, C shell, the **Korn** shell and some other more advanced shells, retain information about the former commands you've executed in the shell. How history is done will depend on the shell used. Here we'll describe the C shell history features.

You can use the **history** and **savehist** variables to set the number of previously executed commands to keep track of in this shell and how many to retain between logins, respectively. You could put a line such as the following in **.cshrc** to save the last 100 commands in this shell and the last 50 through the next login.

```
set history=100 savehist=50
```

The shell keeps track of the history list and saves it in **~/.history** between logins.

You can use the built-in *history* command to recall previous commands, e.g. to print the last 10:

```
% history 10
52 cd workshop
53 ls
54 cd Unix_intro
55 ls
56 pwd
57 date
58 w
59 alias
60 history
61 history 10
```

You can repeat the last command by typing !!:

```
% !!
53 ls
54 cd Unix_intro
55 ls
56 pwd
57 date
```

```
58 w
59 alias
60 history
61 history 10
62 history 10
```

You can repeat any numbered command by prefacing the number with a !, e.g.:

```
% !57
date
Tue Apr 9 09:55:31 EDT 1996
```

Or repeat a command starting with any string by prefacing the starting unique part of the string with a !, e.g.:

```
% !da
date
Tue Apr 9 09:55:31 EDT 2004
```

When the shell evaluates the command line it first checks for history substitution before it interprets anything else. Should you want to use one of these special characters in a shell command you will need to escape, or quote it first, with a \ before the character, i.e. \!. The history substitution characters are summarized in the following table.

TABLE : Bash and C Shell History Substitution

!!	repeat last command
!n	repeat command number n
!-n	repeat command n from last
!str	repeat command that started with string str
!?str?	repeat command with str anywhere on the line
!?str?%	select the first argument that had str in it
!:	repeat the last command, generally used with a modifier
!:n	select the nth argument from the last command (n=0 is the command name)
!:n-m	select the nth through mth arguments from the last command
!^	select the first argument from the last command (same as !:1)
!$	select the last argument from the last command

!*	select all arguments to the previous command
!:n*	select the nth through last arguments from the previous command
!:n-	select the nth through next to last arguments from the previous command
^str1^str2^	replace str1 with str2 in its first occurrence in the previous command
!n:s/str1/str2/	substitute str1 with str2 in its first occurrence in the nth command, ending with a g substitute globally

Additional editing modifiers are described in the man page.

Changing your Shell

To change your shell you can usually use the "*chsh*" or "*passwd -e*" commands. The option flag, here -e, may vary from system to system (-s on BSD based systems), so check the man page on your system for proper usage. Sometimes this feature is disabled. If you can't change your shell check with your System Administrator.

The new shell must be the full path name for a valid shell on the system. Which shells are available to you will vary from system to system. The full path name of a shell may also vary. Normally, though, the Bash and C shells are standard, and available as:

```
/bin/bash
/bin/csh
```

Some systems will also have the Korn shell standard, normally as:

```
/bin/ksh
```

Some shells that are quite popular, but not normally distributed by the OS vendors are bash and tcsh. These might be placed in /bin or a locally defined directory, e.g. /usr/local/bin or /opt/local/bin. Should you choose a shell not standard to the OS make sure that this shell, and all login shells available on the system, are listed in the file /etc/shells. If this file exists and your shell is not listed in this file the file transfer protocol daemon, *ftpd*, will not let you connect to this machine. If this file does not exist only accounts with "standard" shells are allowed to connect via ftp.

You can always try out a shell before you set it as your default shell. To do this just type in the shell name as you would any other command.

Chapter 8

Special Unix Features

One of the most important contributions Unix has made to Operating Systems is the provision of many utilities for doing common tasks or obtaining desired information. Another is the standard way in which data is stored and transmitted in Unix systems. This allows data to be transmitted to a file, the terminal screen, or a program, or from a file, the keyboard, or a program; always in a uniform manner. The standardized handling of data supports two important features of Unix utilities: I/O redirection and piping.

With output redirection, the output of a command is redirected to a file rather than to the terminal screen. With input redirection, the input to a command is given via a file rather than the keyboard. Other tricks are possible with input and output redirection as well, as you will see. With piping, the output of a command can be used as input (piped) to a subsequent command. In this chapter we discuss many of the features and utilities available to Unix users.

File Descriptors

There are 3 standard file descriptors:

• **stdin**	0 Standard input to the program
• **stdout**	1 Standard output from the program
• **stderr**	2 Standard error output from the program

Normally input is from the keyboard or a file. Output, both **stdout** and **stderr**, normally go to the terminal, but you can redirect one or both of these to one or more files.

You can also specify additional file descriptors, designating them by a number 3 through 9, and redirect I/O through them.

File Redirection

Output redirection takes the output of a command and places it into a named file. Input redirection reads the file as input to the command. The following table summarizes the redirection options.

TABLE: File Redirection

Symbol	**Redirection**
>	output redirect
>!	same as above, but overrides noclobber option of *csh*
>>	append output
>>!	same as above, but overrides noclobber option on csh and creates the file if it doesn't already exist.
|	pipe output to another command
<	input redirection

Symbol	Redirection
<<	String read from standard input until "String" is encountered as the only thing on the line.Also known as a "here document"
<<\	String same as above, but don't allow shell substitutions

An example of output redirection is:

$ cat file1 file2 > file3

The above command concatenates file1 then file2 and redirects (sends) the output to file3. If file3 doesn't already exist it is created. If it does exist it will either be truncated to zero length before the new contents are inserted, or the command will be rejected, if the noclobber option of the *csh* is set. (See the *csh* in Chapter 4). The original files, file1 and file2, remain intact as separate entities.

Output is appended to a file in the form:

$ *cat file1 >> file2*

This command appends the contents of file1 to the end of what already exists in file2. (Does not overwrite file2).

Input is redirected from a file in the form:

$ program < file

This command takes the input for *program* from file.

To pipe output to another command use the form:

command | command

This command makes the output of the first command the input of the second command.

Csh

>& file	redirect stdout and stderr to file
>>&	append stdout and stderr to file
|& command	pipe stdout and stderr to command

To redirect stdout and stderr to two separate files you need to first redirect stdout in a sub-shell, as in:

% (command > out_file) >& err_file

Bash

2> file	direct stderr to file

> file 2>&1	direct both stdout and stderr to file
>> file 2>&1	append both stdout and stderr to file
2>&1 | command	pipe stdout and stderr to command

To redirect stdout and stderr to two separate files you can do:

```
$ command 1> out_file 2> err_file
```

or, since the redirection defaults to stdout:

```
$ command > out_file 2> err_file
```

With the Bash shell you can specify other file descriptors (3 onwards) and redirect output through them. This is done with the form:

n>&m	redirect file descriptor n to file descriptor m

We used the above to send stderr (2) to the same place as stdout (1), 2>&1, when we wanted to have error messages and normal messages to go to file instead of the terminal. If we wanted only the error messages to go to the file we could do this by using a place holder file descriptor, 3. We'll first redirect 3 to 2, then redirect 2 to 1, and finally, we'll redirect 1 to 3:

```
$ (command 3>&2 2>&1 1>&3) > file
```

This sends stderr to 3 then to 1, and stdout to 3, which is redirected to 2. So, in effect, we've reversed file descriptors 1 and 2 from their normal meaning. We might use this in the following example:

```
$ (cat file 3>&2 2>&1 1>&3) > errfile
```

So if file is read the information is discarded from the command output, but if file can't be read the error message is put in errfile for your later use.

You can close file descriptors when you're done with them:

m<&-	closes an input file descriptor
<&-	closes stdin
m>&-	closes an output file descriptor
>&-	closes stdout

Other Special Command Symbols

In addition to file redirection symbols there are a number of other special symbols you can use on a command line. These include:

;	command separator
&	run the command in the background

;	command separator
&&	run the command following this only if the previous command completes successfully, e.g.:*grep* string file && *cat* file
||	run the command following only if the previous command did not complete successfully, e.g.:
	grep string file || echo "String not found."
( )	the commands within the parentheses are executed in a subshell. The output of the subshell can be manipulated as above.
’ ’	literal quotation marks. Don’t allow any special meaning to any characters within these quotations.
\	escape the following character (take it literally)
" "	regular quotation marks. Allow variable and command substitution with theses quotations (does not disable $ and \ within the string).
‘command‘	take the output of this command and substitute it as an argument(s) on the command line
#	everything following until <newline> is a comment

The \ character can also be used to escape the <newline> character so that you can continue a long command on more than one physical line of text.

Wild Cards

The shell and some text processing programs will allow meta-characters, or wild cards, and replace them with pattern matches. For filenames these meta-characters and their uses are:

?	match any single character at the indicated position
*	match any string of zero or more characters
[abc...]	match any of the enclosed characters
[a-e]	match any characters in the range a,b,c,d,e

?	match any single character at the indicated position
[!def]	match any characters not one of the enclosed characters, sh only
{abc,bcd,cde }	match any set of characters separated by comma (,) (no spaces), csh only
~	home directory of the current user, csh only
~user	home directory of the specified user, csh only

Chapter 9

Other Useful Commands

Working With Files

This section will describe a number of commands that you might find useful in examining and manipulating the contents of your files.

TABLE : File utilities

Command/Syntax	**What it will do**
cmp [options] *file1 file2*	compare two files and list where differences occur (text or binary files)
cut [options] [*file*(s)]	cut specified field(s)/character(s) from lines in file(s)
diff [options] *file1 file2*	compare the two files and display the differences (text files only)
file [options] *file*	classify the file type
find directory [options] [actions]	find files matching a type or pattern
ln [options] *source_file target*	link the *source_file* to the *target*
paste [options] *file*	paste field(s) onto the lines in *file*
sort [options] *file*	sort the lines of the *file* according to the options chosen
strings [options] *file*	report any sequence of 4 or more printable characters ending in <NL> or <NULL>. Usually used to search binary files for ASCII strings.
tee [options] *file*	copy stdout to one or more files
touch [options] [date] *file*	create an empty file, or update the access time of an existing file
tr [options] *string1 string2*	translate the characters in string1 from stdin into those in string2 in stdout
uniq [options] *file*	remove repeated lines in a file
wc [options] [*file*(s)]	display word (or character or line) count for *file*(s)

cmp - compare file contents

The *cmp* command compares two files, and (without options) reports the location of the first difference between them. It can deal with both binary and ASCII file comparisons. It does a byte-by-byte comparison.

Syntax

cmp [options] file1 file2 [skip1] [skip2]

The skip numbers are the number of bytes to skip in each file before starting the comparison.

Common Options

-l	report on each difference
-s	report exit status only, not byte differences

Examples

Given the files :

Temp1	Temp2
ageorge	ageorge
bsmith	cbetts
cbetts	jchen
jchen	jdoe
jmarsch	jmarsch
lkeres	lkeres
mschmidt	proy
sphillip	sphillip
wyepp	wyepp

The comparison of the two files yields:

```
% cmp Temp1 Temp2
        Temp1 Temp2 differ: char 9, line 2
```

The default it to report only the first difference found.

This command is useful in determining which version of a file should be kept when there is more than one version.

diff - differences in files

The *diff* command compares two files, directories, etc, and reports all differences between the two. It deals only with ASCII files. It's output format is designed to report the changes necessary to convert the first file into the second.

Syntax

diff [options] file1 file2

Common Options

-b	ignore trailing blanks
-i	ignore the case of letters
-w	ignore <space> and <tab> characters
-e	produce an output formatted for use with the editor, *ed*
-r	apply diff recursively through common sub-directories.

Examples

For the Temp1 and Temp2 files above, the difference between them is given by:

```
% diff Temp1 Temp2

        2d1

        < bsmith

        4a4

        > jdoe

        7c7

        < mschmidt

        ---
```

- proy
-

Note that the output lists the differences as well as in which file the difference exists. Lines in the first file are preceded by "< ", and those in the second file are preceded by "> ".

cut - select parts of a line

The *cut* command allows a portion of a file to be extracted for another use.

Syntax

cut [options] file

Common Options

-c	character_list character positions to select (first character is 1)
-d	delimiter field delimiter (defaults to <TAB>)
-f	field_list fields to select (first field is 1)

Both the character and field lists may contain comma-separated or blank-character-separated numbers (in increasing order), and may contain a hyphen (-) to indicate a range. Any numbers missing at either before (e.g. -5) or after (e.g. 5-) the hyphen indicates the full range starting with the first, or ending with the last character or field, respectively. Blank-character-separated lists must be enclosed in quotes. The field delimiter should be enclosed in quotes if it has special meaning to the shell, e.g. when specifying a <space> or <TAB> character.

Examples

In these examples we will use the file users:

```
RRaju           Ranga raju            04/15/96
RK              Radhakrishna          03/12/96
MKabadagi       Mahanthesh kabadagi         01/05/96
YChopra         Yash chopra           04/17/96
SKhan           Sharukh Khan          04/02/96
```

If you only wanted the username and the user's real name, the *cut* command could be used to get only that information:

```
% cut -f 1,2 users
```

```
RRaju           Ranga Raju
RK              Radha krishna
MKabadagi       Mahanthesh kabadagi
YChopra         Yash chopra
SKhan           Sharukh Khan
```

The *cut* command can also be used with other options. The -c option allows characters to be the selected cut. To select the first 4 characters:

```
% cut -c 1-4 users
```

This yields:

RRaj	Ranga raju	04/15/96
RK	Radhakrishna	03/12/96
MKab	Mahanthesh kabadagi	01/05/96
YCho	Yash chopra	04/17/96
SKha	Sharukh Khan	04/02/96

thus cutting out only the first 4 characters of each line.

paste - merge files

The *paste* command allows two files to be combined side-by-side. The default delimiter between thecolumns in a paste is a tab, but options allow other delimiters to be used.

Syntax

paste [options] file1 file2

Common Options

-d	list list of delimiting characters
-s	concatenate lines

The list of delimiters may include a single character such as a comma; a quoted string, such as a space; or any of the following escape sequences:

\n	<newline> character
\t	<tab> character
\\	Backslash character
\0	Empty string (non-null character)

It may be necessary to quote delimiters with special meaning to the shell.

A hyphen (-) in place of a file name is used to indicate that field should come from standard input.

Examples

Given the file users:

```
jdoe          John Doe      04/15/96
lsmith        Laura Smith   03/12/96
pchen         Paul Chen     01/05/96
jhsu          Jake Hsu      04/17/96
sphilip       Sue Phillip   04/02/96
```

and the file phone:

```
John Doe          555-6634
Laura Smith       555-3382
Paul Chen         555-0987
Jake Hsu          555-1235
Sue Phillip       555-7623
```

the *paste* command can be used in conjunction with the *cut* command to create a new file, listing, that includes the username, real name, last login, and phone number of all the users. First, extract the phone numbers into a temporary file, temp.file:

```
% cut -f2 phone > temp.file
555-6634
555-3382
555-0987
555-1235
555-7623
```

The result can then be pasted to the end of each line in users and directed to the new file, listing:

```
    % paste users temp.file > listing
```

```
jdoe      John Doe      04/15/96     237-6634
lsmith    Laura Smith   03/12/96     878-3382
pchen     Paul Chen     01/05/96     888-0987
jhsu      Jake Hsu      04/17/96     545-1235
```

```
sphilip    Sue Phillip    04/02/96    656-7623
```

This could also have been done on one line without the temporary file as:

```
% cut -f2 phone | paste users - > listing
```

with the same results. In this case the hyphen (-) is acting as a placeholder for an input field (namely, the output of the *cut* command).

touch - create a file

The touch command can be used to create a new (empty) file or to update the last access date/time on an existing file. The command is used primarily when a script requires the pre-existence of a file (for example, to which to append information) or when the script is checking for last date or time a function was performed.

Syntax

touch [options] [date_time] file

touch [options] [-t time] file

Common Options

-a	change the access time of the file (SVR4 only)
-c	don't create the file if it doesn't already exist
-f	force the touch, regardless of read/write permissions
-m	change the modification time of the file (SVR4 only)
-t time	use the time specified, not the current time (SVR4 only)

When setting the "-t time" option it should be in the form:

[[CC]YY]MMDDhhmm[.SS]

where:

CC	first two digits of the year

CC	first two digits of the year
YY	second two digits of the year
MM	month, 01-12
DD	day of month, 01-31
hh	hour of day, 00-23
mm	minute, 00-59
SS	second, 00-61

The date_time options has the form:

```
MMDDhhmm[YY]
```

where these have the same meanings as above.

The date cannot be set to be before 1969 or after January 18, 2038.

Examples

To create a file:

```
% touch filename
```

wc - count words in a file

wc stands for "word count"; the command can be used to count the number of lines, characters, or words in a file.

Syntax

wc [options] file

Common Options

-c	count bytes
-m	count characters (SVR4)
-l	count lines
-w	count words

If no options are specified it defaults to "-lwc".

Examples

Given the file users:

jdoe	John Doe	04/15/96
lsmith	Laura Smith	03/12/96
pchen	Paul Chen	01/05/96
jhsu	Jake Hsu	04/17/96
sphilip	Sue Phillip	04/02/96

the result of using a *wc* command is as follows:

```
% wc users
5       20      121 users
```

The first number indicates the number of lines in the file, the second number indicates the number of words in the file, and the third number indicates the number of characters.

Using the *wc* command with one of the options (-l, lines; -w, words; or -c, characters) would result in only one of the above. For example, "*wc -l users*" yields the following result:

```
5 users
```

ln - link to another file

The *ln* command creates a "link" or an additional way to access (or gives an additional name to) another file.

Syntax

ln [options] source [target]

If not specified target defaults to a file of the same name in the present working directory.

Common Options

-f	force a link regardless of target permissions; don't report errors (SVR4 only)
-s	make a symbolic link

Examples

A symbolic link is used to create a new path to another file or directory. If a group of users, for example, is accustomed to using a command called *chkmag*, but the command has been rewritten and is now called *chkit*, creating a symbolic link so the users will automatically execute *chkit* when they enter the command *chkmag* will ease transition to the new command.

A symbolic link would be done in the following way:

```
% ln -s chkit chkmag
```

The long listing for these two files is now as follows:

```
16  -rwxr-x---    1    lindadb    acs    15927  Apr 23 04:10   chkit
1   lrwxrwxrwx    1    lindadb    acs    5      Apr 23 04:11   chkmag -> chkit
```

Note that while the permissions for *chkmag* are open to all, since it is linked to *chkit*, the permissions, group and owner characteristics for *chkit* will be enforced when *chkmag* is run.

With a symbolic link, the link can exist without the file or directory it is linked to existing first.

A hard link can only be done to another file on the same file system, but not to a directory (except by the superuser). A hard link creates a new directory entry pointing to the same inode as the original file. The file linked to must exist before the hard link can be created. The file will not be deleted until all the hard links to it are removed. To link the two files above with a hard link to each other do:

```
% ln chkit chkmag
```

Then a long listing shows that the inode number (742) is the same for each:

```
% ls -il chkit chkmag
742    -rwxr-x---    2    lindadb    acs    15927  Apr 23 04:10   chkit
742    -rwxr-x---    2    lindadb    acs    15927  Apr 23 04:10   chkmag
```

sort - sort file contents

The *sort* command is used to order the lines of a file. Various options can be used to choose the order as well as the field on which a file is sorted. Without any options, the sort compares entire lines in the file and outputs them in ASCII order (numbers first, upper case letters, then lower case letters).

Syntax

sort [options] [+pos1 [-pos2]] file

Common Options

-b	ignore leading blanks (<space> & <tab>) when determining starting and ending characters for the sort key
-d	dictionary order, only letters, digits, <space> and <tab> are significant
-f	fold upper case to lower case
-k	keydef sort on the defined keys (not available on all systems)
-i	ignore non-printable characters
-n	numeric sort
-o	outfile output file
-r	reverse the sort
-t	char use char as the field separator character
-u	unique; omit multiple copies of the same line (after the sort)
+pos1 [-pos2]	(old style) provides functionality similar to the "-k keydef" option.

For the +/-position entries pos1 is the starting word number, beginning with 0 and pos2 is the ending word number. When -pos2 is omitted the sort field continues through the end of the line. Both pos1and pos2 can be written in the form w.c, where w is the word number and c is the character within the word. For c 0 specifies the delimiter preceding the first character, and 1 is the first character of the word. These entries can be followed by type modifiers, e.g. n for numeric, b to skip blanks, etc.

The keydef field of the "-k" option has the syntax:

start_field [type] [,end_field [type]]

where:

start_field, end_field	define the keys to restrict the sort to a portion of the line
type	modifies the sort, valid modifiers are given the single characters (bdfiMnr) from the similar sort options, e.g. a type b is equivalent to "-b", but applies only to the specified field

Examples

In the file users:

```
jdoe        John Doe        04/15/96
```

```
lsmith      Laura Smith    03/12/96
pchen       Paul Chen      01/05/96
jhsu        Jake Hsu       04/17/96
sphilip     Sue Phillip    04/02/96
```

sort users yields the following:

```
jdoe        John Doe       04/15/96
jhsu        Jake Hsu       04/17/96
lsmith      Laura Smith    03/12/96
pchen       Paul Chen      01/05/96
sphilip      Sue Phillip   04/02/96
```

If, however, a listing sorted by last name is desired, use the option to specify which field to sort on (fields are numbered starting at 0):

```
    % sort +2 users:
```

```
pchen       Paul Chen      01/05/96
jdoe        John Doe       04/15/96
jhsu        Jake Hsu       04/17/96
sphilip     Sue Phillip    04/02/96
lsmith      Laura Smith    03/12/96
```

To sort in reverse order:

```
    % sort -r users:
```

```
sphilip     Sue Phillip    04/02/96
pchen       Paul Chen      01/05/96
lsmith      Laura Smith    03/12/96
```

```
jhsu        Jake Hsu      04/17/96

jdoe         John Doe     04/15/96
```

A particularly useful *sort* option is the -u option, which eliminates any duplicate entries in a file while ordering the file. For example, the file todays.logins:

```
sphillip

jchen

jdoe

lkeres

jmarsch

ageorge
```

shows a listing of each username that logged into the system today. If we want to know how many unique users logged into the system today, using sort with the -u option will list each user only once.

(The command can then be piped into "*wc -l*" to get a number):

```
% sort -u todays.logins

ageorge

jchen

jdoe

jmarsch

lkeres

proy

sphillip
```

tee - copy command output

tee sends standard in to specified files and also to standard out. It's often used in command pipelines.

Syntax

tee [options] [file[s]]

Common Options

-a	append the output to the files
-i	ignore interrupts

Examples

In this first example the output of *who* is displayed on the screen and stored in the file users.file:

```
$ who | tee users.file
condron    ttyp0   Apr 22 14:10    (lcondron-pc.acs.)
frank      ttyp1   Apr 22 16:19   (nyssa)
condron    ttyp9   Apr 22 15:52   (lcondron-mac.acs)
$ cat users.file
condron     ttyp0  Apr 22 14:10   (lcondron-pc.acs.)
frank      ttyp1    Apr 22 16:19  (nyssa)
condron    ttyp9   Apr 22 15:52    (lcondron-mac.acs)
```

In this next example the output of *who* is sent to the files users.a and users.b. It is also piped to the *wc* command, which reports the line count.

```
      $ who | tee users.a users.b | wc –l
              3
      $ cat users.a


condron    ttyp0   Apr 22 14:10   (lcondron-pc.acs.)
frank      ttyp1   Apr 22 16:19    (nyssa)
condron    ttyp9   Apr 22 15:52   (lcondron-mac.acs)


      $ cat users.b
condron     ttyp0  Apr 22 14:10   (lcondron-pc.acs.)
frank       ttyp1  Apr 22 16:19    (nyssa)
condron    ttyp9   Apr 22 15:52   (lcondron-mac.acs)
```

In the following example a long directory listing is sent to the file files.long. It is also piped to the *grep* command which reports which files were last modified in August.

```
$  ls -l | tee files.long |grep Aug

1   drwxr-sr-x   2    condron    512    08/08/95     News/
2   -rw-r--r--   1    condron    1076   08/08/95     magnus.cshrc
2   -rw-r--r--   1    condron    1252   08/08/95     magnus.login

$ cat files.long
total 34
2   -rw-r--r--   1    condron    1253   Oct 10 1995    #.login#
1   drwx------   2    condron     512   Oct 17 1995    Mail/
1   drwxr-sr-x   2    condron     512   Aug 8 1995      News/
5    -rw-r--r--  1     condron    4299  Apr 21 00:18   editors.txt
2   -rw-r--r--   1    condron    1076   Aug 8 1995     magnus.cshrc
2   -rw-r—r--     1    condron    1252  Aug 8 1995     magnus.login
7   -rw-r--r--   1    condron     6436  Apr 21 23:50   resources.txt
4   -rw-r--r--   1    condron     3094  Apr 18 18:24   telnet.ftp
1   drwxr-sr-x    2   condron     512   Apr 21 23:56   uc/
1   -rw-r--r--   1    condron     1002  Apr 22 00:14   uniq.tee.txt
1   -rw-r--r--   1    condron    1001   Apr 20 15:05   uniq.tee.txt~
7   -rw-r--r--   1    condron     6194  Apr 15 20:18   Unixgrep.txt
```

find - find files

The *find* command will recursively search the indicated directory tree to find files matching a type or pattern you specify. *find* can then list the files or execute arbitrary commands based on the results.

Syntax

find directory [search options] [actions]

Common Options

For the time search options the notation in days, n is:

+n more than n days

n exactly n days

-n   less than n days

Some file characteristics that *find* can search for are:

time that the file was last accessed or changed

atime n	access time, true if accessed n days ago
-ctime n	change time, true if the files status was changed n days ago
-mtime n	modified time, true if the files data was modified n days ago
-newer	filename true if newer than filename
-type type	type of file, where type can be:
b	block special file
c	character special file
d	directory
l	symbolic link
p	named pipe (fifo)
f	regular file
-fstype type	type of file system, where type can be any valid file system type, e.g.: ufs (Unix File System) and nfs (Network File System)
-user username	true if the file belongs to the user username
-group groupname	true if the file belongs to the group groupname
-perm [-]mode	permissions on the file, where mode is the octal modes for the *chmod* command. When mode is precede by the minus sign only the bits that are set are compared.

-exec command	execute command. The end of command is indicated by and escaped semicolon (\;). the command argument, {}, replaces the current path name.
-name filename	true if the file is named filename. Wildcard pattern matches are allowed if the meta-character is escaped from the shell with a backslash (\).
-ls	always true. It prints a long listing of the current pathname.
-print	print the pathnames found (default for SVR4, not for BSD)

Complex expressions are allowed. Expressions should be grouped within parenthesis (escaping the parenthesis with a backslash to prevent the shell from interpreting them). The exclamation symbol (!) can be used to negate an expression. The operators: -a (and) and -o (or) are used to group expressions.

Examples

find will recursively search through sub-directories, but for the purpose of these examples we will just use the following files:

```
14  -rw-r--r--   1 frank   staff  6682   Feb   05:10:04 am   library
6   -r--r-----   1 frank   staff  3034   Mar   16 1995       netfile
34  -rw-r--r--   1 frank   staff  17351  Feb   05:10:04 am   standard
2   -rwxr-xr-x   1 frank   staff  386    Apr   26:09:51      tr25*
```

To find all files newer than the file, library:

```
% find . -newer library –print
        ./tr25
        ./standard
```

To find all files with general read or execute permission set, and then to change the permissions on those files to disallow this:

```
% find . \( -perm -004 -o -perm -001 \) -exec chmod o-rx {} \; -exec ls -al {} \;
```

```
-rw-r-----   1 frank   staff  6682   Feb   05:10:04 am   ./library
-rwxr-x---   1 frank   staff  386    Apr   26:09:51      ./tr25
-rw-r-----   1 frank   staff  17351  Feb   05:10:04 am   ./standard
```

In this example the parentheses and semicolons are escaped with a backslash to prevent the shell from interpreting them. The curly brackets are automatically replaced by the results from the previous search and the semicolon ends the command.

We could search for any file name containing the string "ar" with:

```
% find . -name \*ar\* -ls
```

```
326584    7 -rw-r-----   1 frank    staff   6682  Feb   5 10:04 ./library
326585   17 -rw-r-----   1 frank    staff   17351 Feb   5 10:04 ./standard
```

where the -ls option prints out a long listing, including the inode numbers.

tar - archive files

The *tar* command combines files into one device or filename for archiving purposes. The *tar* command does not compress the files; it merely makes a large quantity of files more manageable.

Syntax

tar [options] [directory file]

Common Options

c	create an archive (begin writting at the start of the file)
t	table of contents list
x	extract from an archive
v	verbose
f	archive file name
b	archive block size

tar will accept its options either with or without a preceding hyphen (-). The archive file can be a disk file, a tape device, or standard input/output. The latter are represented by a hyphen.

Examples

Given the files and size indications below:

```
45 logs.beauty
89 logs.bottom
74 logs.photon
84 logs.top
```

tar can combine these into one file, logfile.tar:

```
% tar -cf logfile.tar logs.* ; ls -s logfile.tar
304 logfile.tar
```

Many anonymous FTP archive sites on the Internet store their packages in compressed tar format, so the files will end in .tar.Z or .tar.gz. To extract the files from these files you would first uncompress them, or use the appropriate zcat command and pipe the output into tar, e.g.:

```
% zcat archive.tar.Z | tar -xvf –
```

where the hyphen at the end of the *tar* command indicates that the file is taken from stdin.

Chapter 10

Shell Programming

Shell Scripts

You can write shell programs by creating scripts containing a series of shell commands. The first line of the script should start with #! which indicates to the kernel that the script is directly executable. You immediately follow this with the name of the shell, or program (spaces are allowed), to execute, using the full path name. Generally you can count on having up to 32 characters, possibly more on

some systems, and can include one option. So to set up a Bash shell script the first line would be:

```
#! /bin/sh
```

or for the C shell:

```
#! /bin/csh -f
```

where the "-f" option indicates that it should not read your .cshrc. Any blanks following the magic symbols, #!, are optional.

You also need to specify that the script is executable by setting the proper bits on the file with *chmod*,

e.g.:

```
% chmod +x shell_script
```

Within the scripts # indicates a comment from that point until the end of the line, with #! being a special case if found as the first characters of the file.

Setting Parameter Values

Parameter values, e.g. param, are assigned as:

Bash shell	C shell
param=value	set param = value

where value is any valid string, and can be enclosed within quotations, either single (’value) or double ("value"), to allow spaces within the string value. When enclosed with backquotes (‘value‘) the string is first evaluated by the shell and the result is substituted. This is often used to run a command, substituting the command output for value, e.g.:

```
$ day=‘date +%a‘
$ echo $day
Wed
```

After the parameter values has been assigned the current value of the parameter is accessed using the $param, or ${param}, notation.

Quoting

We quote strings to control the way the shell interprets any parameters or variables within the string. We can use single (’) and double (") quotes around strings. Double quotes define the string, but allow variable substitution. Single quotes define the string and prevent variable substitution. A backslash (\) before a character is said to escape it, meaning that the system should take the character literally, without assigning any special meaning to it. These quoting techniques can be used to separate a variable from a fixed string. As an example lets use the variable, var, that has been assigned the value bat, and the constant string, man. If I wanted to combine these to get the result "batman" I might try:

$varman

but this doesn’t work, because the shell will be trying to evaluate a variable called varman, which doesn’t exist. To get the desired result we need to separate it by quoting, or by isolating the variable with curly braces ({}), as in:

"$var"man	- quote the variable
$var""man	- separate the parameters
$var"man"	- quote the constant
$var''man	- separate the parameters
$var'man'	- quote the constant
$var\man	- separate the parameters
${var}man	- isolate the variable

These all work because ", ’, \, {, and } are not valid characters in a variable name.

We could not use either of

```
’$var’man
\$varman
```

because it would prevent the variable substitution from taking place.

When using the curly braces they should surround the variable only, and not include the $, otherwise, they will be included as part of the resulting string, e.g.:

```
% echo {$var}man
  {bat}man
  Variables
```

Variables

There are a number of variables automatically set by the shell when it starts. These allow you to reference arguments on the command line.

These shell variables are:

TABLE : Shell Variables

Variable	Usage	sh	csh
$#	number of arguments on the command line	x	
$-	options supplied to the shell	x	
$?	exit value of the last command executed	x	
$$	process number of the current process	x	x
$!	process number of the last command done in background	x	
$n	argument on the command line, where n is from 1 through 9, reading left to right	x	x
$0.00	the name of the current shell or program	x	x
$*	all arguments on the command line ("$1 $2 ... $9")	x	x
$@	all arguments on the command line, each separately quoted ("$1" "$2" ... "$9")	x	
$argv[n]	selects the nth word from the input list		x
${argv[n]}	same as above		x
$#argv	report the number of words in the input list		x

We can illustrate these with some simple scripts. First for the Bash shell the script will be:

```
#!/bin/sh
echo "$#:" $#
echo '$#:' $#
echo '$-:' $-
echo '$?:' $?
echo '$$:' $$
echo '$!:' $!
echo '$3:' $3
echo '$0:' $0
echo '$*:' $*
echo '$@:' $@
```

When executed with some arguments it displays the values for the shell variables, e.g.:

```
$ ./variables.sh one two three four five
 5: 5
 $#: 5
 $-:
 $?: 0
 $$: 12417
 $!:
 $3: three
 $0: ./variables.sh
 $*: one two three four five
 $@: one two three four five
```

As you can see, we needed to use single quotes to prevent the shell from assigning special meaning to $. The double quotes, as in the first echo statement, allowed substitution to take place.

Similarly, for the C shell variables we illustrate variable substitution with the script:

```
#!/bin/csh -f
echo '$$:' $$
echo '$3:' $3
echo '$0:' $0
echo '$*:' $*
echo '$argv[2]:' $argv[2]
echo '${argv[4]}:' ${argv[4]}
echo '$#argv:' $#argv
```

which when executed with some arguments displays the following:

```
% ./variables.csh one two three four five
 $$: 12419
 $3: three
 $0: ./variables.csh
 $*: one two three four five
 $argv[2]: two
 ${argv[4]}: four
 $#argv: 5
```

Parameter Substitution

You can reference parameters abstractly and substitute values for them based on conditional settings using the operators defined below. Again we will use the curly braces ({}) to isolate the variable and its operators.

$parameter	substitute the value of parameter for this string.
${parameter}	same as above. The brackets are helpful if there's no separation between this parameter and a neighboring string.
$parameter=	sets parameter to null.
${parameter-default}	if parameter is not set, then use default as the value here. The parameter is not reset.
${parameter=default}	if parameter is not set, then set it to default and use the new value
${parameter+newval)	if parameter is set, then use newval, otherwise use nothing here. The parameter is not reset.
${parameter?message}	if parameter is not set, then display message. If parameter is set, then use its current value.

There are no spaces in the above operators. If a colon (:) is inserted before the -, =, +, or ? then a test if first performed to see if the parameter has a non-null setting.

The C shell has a few additional ways of substituting parameters:

$list[n]	selects the nth word from list

${list[n]}	same as above
$#list	report the number of words in list
$?parameter	return 1 if parameter is set, 0 otherwise
${?parameter}	same as above
$<	read a line from stdin

The C shell also defines the array, $argv[n] to contain the n arguments on the command line and $#argv to be the number of arguments, as noted in Table above.

To illustrate some of these features we'll use the test script below.

```
#!/bin/bash
  param0=$0
  test -n "$1" && param1=$1
  test -n "$2" && param2=$2
  test -n "$3" && param3=$3
  echo 0: $param0
  echo "1: ${param1-1}: \c" ;echo $param1
  echo "2: ${param2=2}: \c" ;echo $param2
  echo "3: ${param3+3}: \c" ;echo $param3
```

In the script we first test to see if the variable exists, if so we set a parameter to its value. Below this we report the values, allowing substitution.

In the first run through the script we won't provide any arguments:

```
$ ./parameter.sh
  0: ./parameter.sh # always finds $0
  1: 1: # substitute 1, but don't assign this value
  2: 2: 2 # substitute 2 and assign this value
  3: : # don't substitute
```

In the second run through the script we'll provide the arguments:

```
$ ./parameter one two three
  0: ./parameter.sh # always finds $0
  1: one: one # don't substitute, it already has a value
  2: two: two # don't substitute, it already has a value
```

```
    3: 3: three # substitute 3, but don’t assign this value
```

Dealing with numbers in shell scripts

The special shell command $((….)) evaluates any numerical expression within the innermost parenthesis, and outputs the numerical result. This result can be then assigned to shell variables. This commands illustrate the use of this special command.

```
$ sum=$(( 5  +  4))

$ echo  $sum

9
```

here, after adding 5 and 4 the sum 9 is assigned to variable sum.

The other arithmetic operators will work as illustrated below.

```
$prod=$(( 6 * 8 ))

$echo $prod

48
```

We can also use value of existing variable like:

```
$ prod=$(($prod * 5 ))

$ echo   $prod

240
```

Here, the existing value of prod is used for further calculation. We can also use parentheses to indicate precedence in calculations:

```
$ sum=$(( 90 + 10 ) * 10))

    $ echo $sum

    1000
```

Here Document

A here document is a form of quoting that allows shell variables to be substituted. It’s a special form of redirection that starts with <<WORD and ends with WORD as the only contents of a line. In the Bash shell you can prevent shell substitution by escaping WORD by putting a \ in front of it on the redirection line, i.e. <<\WORD, but not on the ending line. To have the same effect the C shell expects the \ in front of WORD at both locations.

The following scripts illustrate this,

for the Bash shell:	and for the C shell:
#!/bin/bash	#!/bin/csh -f
does=does	set does = does
not=""	set not = ""
cat << EOF	cat << EOF
This here document	This here document
$does $not	$does $not
do variable substitution	do variable substitution
EOF	EOF
cat << \EOF	cat << \EOF
This here document	This here document
$does $not	$does $not
do variable substitution	do variable substitution
EOF	\EOF

Both produce the output:

```
This here document
does
do variable substitution
This here document
$does $not
do variable substitution
```

In the top part of the example the shell variables $does and $not are substituted. In the bottom part they are treated as simple text strings without substitution.

Interactive Input

Shell scripts will accept interactive input to set parameters within the script.

Bash

bash uses the built-in command, read, to read in a line, e.g.:

```
read param
```

We can illustrate this with the simple script:

```
#!/bin/bash

echo "Input a phrase \c"        # This is /bin/echo which requires "\c"
            to prevent <newline>

read param

echo param=$param
```

When we run this script it prompts for input and then echoes the results:

```
$ ./read.sh

Input a phrase hello frank      # I type in hello frank <return>

param=hello frank
```

Functions

The Bash shell has functions. These are somewhat similar to aliases in the C shell, but allow you more flexibility. A function has the form:

```
fcn () { command; }
```

where the space after {, and the semicolon (;) are both required; the latter can be dispensed with if a <newline> precedes the }. Additional spaces and <newline>'s are allowed. We saw a few examples of this in the sample .bash_profile in an earlier chapter, where we had functions for ls and ll:

```
ls() { /bin/ls -sbF "$@";}

ll() { ls -al "$@";}
```

The first one redefines *ls* so that the options -sbF are always supplied to the standard */bin/ls* command, and acts on the supplied input, "$@". The second one takes the current value for *ls* (the previous function) and tacks on the -al options.

Functions are very useful in shell scripts. The following is a simplified version of one I use to automatically backup up system partitions to tape.

```
#!/bin/bash

# Cron script to do a complete backup of the system

HOST=`/bin/uname -n`

admin=frank

Mt=/bin/mt

Dump=/usr/sbin/ufsdump
```

```
Mail=/bin/mailx
device=/dev/rmt/0n
Rewind="$Mt -f $device rewind"
Offline="$Mt -f $device rewoffl"
# Failure - exit
failure () {
        $Mail -s "Backup Failure - $HOST" $admin << EOF_failure
$HOST
Cron backup script failed. Apparently there was no tape in the device.

EOF_failure
        exit 1
        }

# Dump failure – exit

dumpfail () {
        $Mail -s "Backup Failure - $HOST" $admin << EOF_dumpfail
$HOST
Cron backup script failed. Initial tape access was okay, but dump failed.

EOF_dumpfail
        exit 1
        }
# Success
success () {
        $Mail -s "Backup completed successfully - $HOST" $admin << EOF_success
$HOST
Cron backup script was apparently successful. The /etc/dumpdates file is:
`/bin/cat /etc/dumpdates`
EOF_success
```

```
        }
    # Confirm that the tape is in the device
    $Rewind || failure
    $Dump 0uf $device / || dumpfail
    $Dump 0uf $device /usr || dumpfail
    $Dump 0uf $device /home || dumpfail
    $Dump 0uf $device /var || dumpfail
    ($Dump 0uf $device /var/spool/mail || dumpfail) && success
    $Offline
```

This script illustrates a number of topics that we've looked at in this document. It starts by setting various parameter values. HOST is set from the output of a command, admin is the administrator of the system, Mt, Dump, and Mail are program names, device is the special device file used to access the tape drive, Rewind and Offline contain the commands to rewind and off-load the tape drive, respectively, using the previously referenced Mt and the necessary options. There are three functions defined: failure, dump fail, and success. The functions in this script all use a here document to form the contents of the function. We also introduce the logical OR (||) and AND (&&) operators here; each is position between a pair of commands. For the OR operator, the second command will be run only if the first command does not complete successfully. For the AND operator, the second command will be run only if the first command does complete successfully.

The main purpose of the script is done with the Dump commands, i.e. backup the specified file systems. First an attempt is made to rewind the tape. Should this fail, || failure, the failure function is run and we exit the program. If it succeeds we proceed with the backup of each partition in turn, each time checking for successful completion (|| dumpfail). Should it not complete successfully we run the dumpfail subroutine and then exit. If the last backup succeeds we proceed with the success function ((...) && success). Lastly, we rewind the tape and take it offline so that no other user can accidently write over our backup tape.

test

Conditional statements are evaluated for true or false values. This is done with the *test*, or its equivalent, the *[]* operators. It the condition evaluates to true, a zero (TRUE) exit status is set, otherwise a non-zero (FALSE) exit status is set. If there are no arguments a non-zero exit status is set. The operators used by the Bash shell conditional statements are given below.

For filenames the options to *test* are given with the **syntax**:

-option filename

The options available for the *test* operator for files include:

-r	true if it exists and is readable
-w	true if it exists and is writable
-x	true if it exists and is executable
-f	true if it exists and is a regular file (or for csh, exists and is not a directory)
-d	true if it exists and is a directory
-h or -L	true if it exists and is a symbolic link
-c	true if it exists and is a character special file (i.e. the special device is accessed one character at a time)
-b	true if it exists and is a block special file (i.e. the device is accessed in blocks of data)
-p	true if it exists and is a named pipe (fifo)
-u	true if it exists and is setuid (i.e. has the set-user-id bit set, s or S in the third bit)
-g	true if it exists and is setgid (i.e. has the set-group-id bit set, s or S in the sixth bit)
-k	true if it exists and the sticky bit is set (a t in bit 9)
-s	true if it exists and is greater than zero in size

There is a test for file descriptors:

-t [file_descriptor]	true if the open file descriptor (default is 1, stdin) is associated with a terminal

There are tests for strings:

-z string	true if the string length is zero
-n string	true if the string length is non-zero

string1 = string2	true if string1 is identical to string2
string1 != string2	true if string1 is non identical to string2
string	true if string is not NULL

There are integer comparisons:

n1 -eq n2	true if integers n1 and n2 are equal
n1 -ne n2	true if integers n1 and n2 are not equal
n1 -gt n2	true if integer n1 is greater than integer n2
n1 -ge n2	true if integer n1 is greater than or equal to integer n2
n1 -lt n2	true if integer n1 is less than integer n2
n1 -le n2	true if integer n1 is less than or equal to integer n2

The following logical operators are also available:

!	negation (unary)
-a	and (binary)
-o	or (binary)

() expressions within the () are grouped together. You may need to quote the () to prevent the shell from interpreting them.

Control Commands

Conditional if

The conditional if statement is available in both shells, but has a different syntax in each.

```
if condition1
then
        command list if condition1 is true
[elif condition2
        then command list if condition2 is true]
[else
        command list if condition1 is false]
fi
```

The conditions to be tested for are usually done with the *test*, or *[]* command. The if and then must be separated, either with a <newline> or a semicolon (;).

```
#!/bin/bash
if [ $# -ge 2 ]
then
        echo $2
elif [ $# -eq 1 ]; then
        echo $1
else
        echo No input
fi
```

There are required spaces in the format of the conditional test, one after [and one before]. This script should respond differently depending upon whether there are zero, one or more arguments on the command line. First with no arguments:

```
$ ./if.sh
    No input
```

Now with one argument:

```
$ ./if.sh one
    one
```

And now with two arguments:

```
$ ./if.sh one two
     two
```

Csh

```
if (condition) command
        -or
if (condition1) then
        command list if condition1 is true
[else if (condition2) then
        command list if condition2 is true]
[else
        command list if condition1 is false]
```

```
    endif
```

The if and then must be on the same line.

```
    #!/bin/csh -f
    if ( $#argv >= 2 ) then
            echo $2
    else if ( $#argv == 1 ) then
            echo $1
    else
            echo No input
    endif
```

Again, this script should respond differently depending upon whether I have zero, one or more arguments on the command line. First with no arguments:

```
    % ./if.csh
          No input
```

Now with one argument:

```
    % ./if.csh one
          one
```

And now with two arguments:

```
    % ./if.csh one two
          two
```

Conditional switch and case

To choose between a set of string values for a parameter use *case* in the Bash shell and *switch* in the C shell.

Bash

case parameter *in*

```
        pattern1[|pattern1a]) command list1;;
        pattern2) command list2
                command list2a;;
        pattern3) command list3;;
        *) ;;
esac
```

You can use any valid filename meta-characters within the patterns to be matched. The ;; ends each choice and can be on the same line, or following a <newline>, as the last command for the choice. Additional alternative patterns to be selected for a particular case are separated by the vertical bar, |, as in the first pattern line in the example above. The wildcard symbols,: ? to indicate any one character and * to match any number of characters, can be used either alone or adjacent to fixed strings.

This simple example illustrates how to use the conditional case statement.

```
#!/bin/bash
case $1 in
        aa|ab) echo A
        ;;
b?)     echo "B \c"
        echo $1;;
c*)     echo C;;
*)      echo D;;
esac
```

So when running the script with the arguments on the left, it will respond as on the right:

aa	A
ab	A
ac	D
bb	B bb
bbb	D

```
c      C
cc     C
fff    D
```

Csh

```
switch (parameter)
case pattern1:
        command list1
        [breaksw]
case pattern2:
        command list2
        [breaksw]
default:
        command list for default behavior
        [breaksw]
endsw
```

breaksw is optional and can be used to break out of the switch after a match to the string value of the parameter is made. Switch doesn’t accept "|" in the pattern list, but it will allow you to string several case statements together to provide a similar result. The following C shell script has the same behavior as the Bash shell case example above.

```
#!/bin/csh -f
switch ($1)
        case aa:
        case ab:
                echo A
                breaksw
        case b?:
                echo -n "B "
                echo $1
                breaksw
        case c*:
                echo C
                breaksw
        default:
                echo D
        endsw
```

for and foreach

One way to loop through a list of string values is with the *for* and *foreach* commands.

Bash

```
    for variable [in list_of_values]
    do
            command list
    done
```

The list_of_values is optional, with $@ assumed if nothing is specified. Each value in this list is sequentially substituted for variable until the list is emptied. Wildcards can be used and are applied to file names in the current directory. Below we illustrate the for loop in copying all files ending in .old to similar names ending in .new. In these examples the *basename* utility extracts the base part of the name so that we can exchange the endings.

```
    #!/bin/bash
    for file in *.old
    do
            newf=`basename $file .old`
            cp $file $newf.new
    done
```

Csh

```
    foreach variable (list_of_values)
            command list
    end
```

The equivalent C shell script to copy all files ending in .old to .new is:

```
#!/bin/csh -f
foreach file (*.old)
        set newf = `basename $file .old`
        cp $file $newf.new
end
```

while

The *while* commands let you loop as long as the condition is true.

Bash

```
while condition
do
        command list
        [break]
        [continue]
done
```

A simple script to illustrate a while loop is:

```
#!/bin/bash
while [ $# -gt 0 ]
do
        echo $1
        shift
done
```

This script takes the list of arguments, echoes the first one, then shifts the list to the left, losing the original first entry. It loops through until it has shifted all the arguments off the argument list.

```
$ ./while.sh one two three
one
two
three
```

Csh

```
*while* (condition)
        command list
        [*break*]
        [*continue*]
end
```

If you want the condition to always be true specify 1 within the conditional test.

A C shell script equivalent to the one above is:

```
#!/bin/csh -f
while ($#argv != 0 )
        echo $argv[1]
        shift
end
```

until

This looping feature is only allowed in the Bash shell.

```
until condition
do
        command list while condition is false
done
```

The condition is tested at the start of each loop and the loop is terminated when the condition is true.

A script equivalent to the while examples above is:

```
#!/bin/bash
until [ $# -le 0 ]
do
        echo $1
        shift
```

```
done
```

Notice, though, that here we’re testing for *less than or equal*, rather than *greater than or equal*, because the until loop is looking for a false condition.

Both the until and while loops are only executed if the condition is satisfied. The condition is evaluated before the commands are executed.

Chapter – 11
System administration

INTRODUCTION:

A System Administrator in Unix is a person responsible for the smooth operation of the system. He wields the all-powerful root account. Hence, he must be very knowledgeable about Unix and the particular installation he is responsible for. The main functions of a System Administrator include:

- Managing users and group.
- Managing and maintaining file systems and devices.
- Managing processes.
- System configuration.
- Installing and updating software.
- Backing up data.
- Booting up and shutting down the system.
- Disaster recovery.

All these functions require that you work as root, with the Permissions and
privileges. Hence, you must be extra careful when performing these tasks in your Unix system. In this and the next session, we will be discussing ways of accomplishing these tasks. Red Hat Unix 6.0 provides excellent graphical tools to accomplish most of these tasks. But, the majority of System Administrators prefer working at the commands at their disposal. Also, in certain cases, working at the command prompt gives you that extra amount of control that graphical tools cannot. We will discuss the command line approach to all these tasks and mention the graphical tools you can use to accomplish the same task in a GUI environment.

Becoming the super User

Although you can log in as root straightaway, the usual way to log in as root is with the su command. The su command allows you to assume the identification of another user. For example,

```
$  su Krishna
```

Will prompt you for the password for Krishna and if it is correct will set your user ID to that of Krishna a username argument, su will prompt you for the root password, validating your user ID as root. Once you are finished using the root account, you log out in the usual way and return to you own mortal identify.
Why not simply log in as root from the usual login prompt? This is desirable in some instances, but most of the time it's best to use su after logging in as yourself. On a system with many users, use of su records a message such as

```
Sep 8 10:44:52 ashok P. AM_pwdb [938]:  (su) session opened for user foot by ashok (uid=500)
```

In the system logs, such as the file /var/log/messages. This message indicates that the user ashok successfully issued the su command, in this case for root. If you were to log in directly as root, no such message would appear in the logs and you wouldn't be able to tell which user was mucking about with the root account.

In most cases, the prompt for the root account differs from that for normal users. Classically, the root prompt contains a hash mark (#), while normal user prompts contain a $ or %.

Managing Users and Groups:

User accounts serve a number of purposes on Unix systems. Most prominently, they give the system a way to distinguish between different people who use the system. Each user has a personal account with a separate username and password. Apart form these personal accounts, these are users on the system that provide administrative functions. As we have seen, the system administrator uses the root account to perform maintenance – but usually not for personal system use. Such accounts are accessed by using the su command.

Other accounts on the system may not be set aside for human interaction at all. These accounts are generally used by system daemons (background system process), which must access files on the system through a specific user ID other than root or one of the personal user accounts. For example, if you configure your system to receive a news feed form another site the news daemon must restore news articles in a spool directory anyone can access, but only one user (the news daemon) can write to. No human being is associated with news account; it is an 'imaginary' user set aside for the news daemon only. As the system administrator, it is your job to create and manage accounts for all users (real and virtual) on your machine. For this you need to understand two system files first.

The /etc/passed File:

Every account on the system has entry in the file /etc/passed. This contains entries, one line per user, that specify several attributes for the user, such the username, real name and so on. Each entry in this file is of the format:

```
Username : password : uid : gid : gecos: home dir: shell
```

Username is the account name by the user to log in or it may be the account name of some system daemon. On most systems, personal account usernames are limited to 8 characters in length.

Password is an encrypted representation of the user's password. This field is set by the passwd program; it cannot be set by hand. A one-way encryption scheme is used that is difficult to break. Note that if this field contains an "x" it means that the encrypted password is actually kept in another system file, /etc/shadow (discussed below). If the first character of the password field is an asterisk (*), it means that the account is disabled; the system will not allow logins as this user.

uid is the user ID. It is a unique integer the system uses to identify the account. The system uses the uid field internally in dealing with process and file permissions. Hence, both the username and uid fields identify a particular account; the uid is more important to the system, while username is more convenient for humans.

gid is the group ID, an integer identifying the user's default group. This is found in the file /etc/group (discussed below).

gecos contains miscellaneous information about the user, such as the user's real name, address telephone number etc. gecos is a historical name dating back to the 1970's; it stands for General Electric Comprehensive Operating System. This field was originally added to /etc/passwd to provide compatibility to some of its services.

homedir is the user's home directory for his or her personal use. In Unix, this is generally /home/<username> for ordinary users, and /root for the root account. When the user logs in, the shell finds the current working directory in the named homedir.

shell is the name of the program to run when the user logs in. Usually, this is the full pathname of a shell, such as /bin/bash.

Some of the above fields are optional. The required ones are username, uid, gid and homedir. Here are a couple of sample entries in the /etc/passwd file.

```
root : x : 0 : 0 : root : /root : /bin/bash
ashok : x : 500 : 500 : Ashok Sengupta : /home/ashok : /bin/bash
```

The first entry is for the root account. Notice that the uid of root is zero. This is what makes the root account special. The system knows that uid 0 does not have the usual security restrictions. The gid of root is also zero. This is by convention. The "x" in the password field indicates that the actual encrypted password is in the file /etc/shadow. The home directory or the root account is /root and the log in shell is /bin/bash. Note that individual fields are separated by colon (:) character.

The second entry is for an actual human user with username ashok. In this case the uid and gid are both 500. Technically, the uid can be any unique integer. It is the convention in RedHat Unix to start uid's from 500 for ordinary users. The gid is also 500, which just means that user ashok is in whatever group is numbered 500 in the file /etc/group.

The /etc/shadow File:

The /etc/shadow file will contain the actual encrypted passwords of accounts on the system, along with other information required by the PAM (Password Authentication Module) used by Red Hat Unix for additional security. Note that the use of /etc/shadow has to be confirmed at the time of installing the system. Here are two sample entries from the /etc/shadow file:

Root : $1$Avcx7gEH$LakZgj9902e77/k, F0dhn. : 10824 : 0 : 99999 : -l : - l : 134538444
Asok: $1$yjQATTFms$7typFMT77aXR62gUXHtqy1 : 10824 : -l :99999 : - 1 : - : - 1 : - 1 : 135348956

In this file also there will be one entry per user. Each entry will be identified by the account name in the first field. The second field contains the actual encrypted password. Rest of the fields are used internally by the system's PAM module. Like the /etc/passwd file, here also the fields are separated by the colon (:) character.

The /etc/group File

The file /etc/group contains a one-line entry for each group in the system. The format of this file is:

groupname : password : gid : members

groupname is a character string that identifies the group, password is an optional password associated with the group, gid is the unique integer identifying the group ID and members is a comma-separated list of users who are members of this group but who have different gid in /etc/passwd. That is, this line need not contain those users who have this group set as their "default" group in /etc/passwd. It's only for users who are additional members of the group. A few sample entries of /etc/group are shown below:

```
root: : 0 : root
bin : : l : root, bin, daemon
daemon : : 2 : root, bin, daemon
sys: : 3 : root, bin, adm
adm : : 4 : root, adm, deamon
ashok : x : 500 :
mohan : x : 501 :
```

The first line shows that the group root has a gid of 0 and root is the sole member of this group. Next few lines identify different system and administrative groups and shows the additional group members. The last two lines are entries for user group. In these two entries, the members field is blank, indicating that there are no additional members in these groups, except the user whose default group is this group as per the /etc/passwd file and this user is not a member of any other group.

Creating User Accounts

You use the useradd command to create new users in your system. The following command

```
# useradd   Krishna
```

will create a new user with log in name krishna, assign the default shell /bin/bash as the log in shell, create the home directory /home/Krishna, assign the next available uid and gid and assign the default group name Krishna. Moreover, it will create the necessary entries in /etc/passwd, /etc/shadow and /etc/group and copy skeleton configuration files (like .bashrc, .bash_profile, .Xdefaults etc.) in the user's home directory.

The next step is to assign an initial password to the user (which the user can later change)

```
# passwd Krishna
```

```
Changing password for user Krishna
New Unix password: <type initial password here>
Retype new Unix password: <re-type password here>
Passwd: all authentication tokens updated successfully
```

Now the new user can log in to the system with this username and password.

For most purposes, the defaults allocated by the system are enough. However, if you wish to assign the user particular groups, or home directory or login shell, the useradd command has command-line options for a lot of such requirements. It is possible to set an expiry date for the new account, using the – e mm/dd/yy option. Look up the man pages of the useradd command for the available options.

Deleting and Disabling User accounts

To delete an user account, you must remove the user's entry in /etc/passwd and /etc/shadow, remove any references to the user in /etc/group and delete the user's home directory as well as any additional files created and owned by the user. For example, if the user is having an incoming mailbox in /usr/spool/mail, it must be deleted as well.

The userdel command will delete an account:

```
#       userdel Krishna
```

Will delete the user account Krishna from the system and update the related files. You can use the –r option to remove the home directory of the user and all files in it also:

```
#       userdel –r Krishna
```

Other files owned by the user have to be manually deleted. To list all such files, use the command:

```
#       find / -user username – ls
```

This will give an ls – l listing of all files owned by the user, if the account haven't yet been deleted. If the account has already been deleted, use the –uid num option of the find command instead.

Disabling an user account, for whatever reason, is very simple. Just put an asterisk (*) in the password field of the user's entry in the /etc/passwd file.

GUI Tools for User Management

Red Hat Unix 9.0 provides you with the Red Hat –config –users utility that allows you to configure various aspects of your system, including user and group management. To start this utility, become the super user by using the su command and then type redhat –config –users. The left pane allows you to select the particular aspects of the system you wish to configure and the right pane will display details of that aspect and will allow you to configure the same.

To manage users and groups, you follow:

Config -> User accounts -> User accounts in the left pane to manage user accounts and
Config -> User accounts -> Normal -> Group definitions

To manage groups. Using text boxes, buttons and tabs that appear on the right pane, you can create, modify, delete user accounts and create delete and modify groups. Short explanations are available at the top of the right pane.

The suid and sgid Permissions:

The suid (set user ID) and sgid (set group ID) permission bits allow a program (not a script) to run with the permissions of it was owner and group, respectively, and group respectively and not the permissions for the account that is running the program. For example, say you have a program with suid set and its owner is root. Anyone running this program runs it with root permissions, instead of his or her own permissions. The passwd
Command is a good example of this. The file /etc/passwd is writable by root and readably by everyone. The passwd program has suid turned on. Therefore, anyone can run the passwd program and change their password. Because the program is running with root permissions and not those of the actual use, the /etc/passwd file can be written into.

The same concept holds true for sgid. The program is run with the permissions of the group associated with the program, not those of the group to which the user belongs. Both suid and sgid are powerful tools and sources of potential security risk for the system.
The suid and sgid permissions are added with the chmod command as shown below:

```
#       chmod u+s file (s)
#       chmod g+s file (s)
```

The first command sets suid for the files listed and the second sets sgid. These bits cannot be set for nonexecutables. This is reflected in a ls –l listing of the files by the letter "s" in place of "x" in the user and group permissions field. For example, an ls –l listing of the passwd command shows:

```
$ ls  -l /usr/bin/passwd
-r-s-x-x        1       root    root 10704 Apr         15      02:51 /usr/bin/passwd
```

Observe that the suid bit is set for this program and it is executable by everyone.

Managing File systems and Devices:

In Unix, a file system is a physical device, such as a floppy disk, a hard disk or a CD-ROM that has been formatted to hold files. The exact way in which files will actually be stored in a file system will depend upon the file system type. For example, a Unix native file system (the default Unix file system) will store files in a way that is different from the way a Windows 95 file system stores files. The default file system type used by Unix is called an ext2 file system (second extended file system). In order to hide the details of the underlying file system format from the user, Unix supports a range of file systems – both its own and those of other operating systems. The huge benefit that accrues from this is that you can access other operating system's file systems from with Unix and work on files stored there with your favorite Unix commands. The following table lists the commonly used file systems supported by Unix:

File system	Type	Description
Third Extended File system	ext3	Unix native File System
Second extended file system	ext2	Unix native filesystem
Network file system	NFS	Allows remote access to files on a network.
M-DOS file system	ms dos	Access files on a DOS filesystem
Win 95/98 file system	vfat	Accesses files stored on a Win (95/98 file system. Can also handle 32-bit FAT.)
Iso 9660 File system	iso9660	File system used by most CD-

ROM's. can handle the Rock ridge extension.

/proc file system	proc	Provides process info for ps command.
HPFS file system	hpfs	OS/2 file system.
System V file system	sys v	Access files on UNIX System V

Each file system type has its own attributes and limitations. For example, the MS-DOS file system limits you to an 8.3 filename convention and should be used to access MS-DOS floppies and partitions. The ext2 and ext3 file systems supports 256 - character filenames and a 4-terrabyte maximum file system size. /proc is a virtual file system. No actual disk space is associated with it. If you look into the /proc directory on your Unix system, you will see a number of "files" and "directories" in it. The contents of these changes with time. The Unix kernel provides system and process information through the /proc file system. When you access some files on this directory, the kernel recognizes your request and provides you the data. None of these files and directories are stored on disk. They are maintained by the Kernel to provide easy access to certain system and process information required by programs like ps and top.

Mounting File systems:

Under Unix (and UNIX), in order to access files in any file system, you must mount the file system on a certain directory. This makes the files on the file system appear as though they reside in that directory. The directory must be an existing directory on your Unix system. Once you mount a file system on a directory, the previous contents of that directory will become "hidden", i.e., you would not be able to access the previous contents until you un-mount the file system. Note that mounting a file system on a directory does not destroy the previous contents of that directory.

The mount command is used to mount file systems. This command usually can be used by root only.

The syntax is

```
mount –t type device mount –point
```

Where type is the file system type (as per table above), device is the name of the device containing the file system (such as /dev/hda1, /dev/fd0 etc.) and mount-point is the name of the existing directory on which you wish to mount the file system. For example, if you have a Windows 98 FAT-32 partition in /dev/hda1, you can mount the same on the directory /mnt with the command.

```
#  mount   –t     vfat    /devf/hda1     /mnt
```

After this command is executed, all the directories and files in the Windows 98 will be available for you to access under the /mnt directory. You can access these files now with your usual cp, mv, rm etc. commands. You can look at those files with your usual ls –l command and so on. Note that Unix is smart enough to understand that the file system under /mnt is a FAT -32 type. For instance, if you copy some of your existing Unix files onto the Win 98 partition, they will be correctly copied for Win 98 to access them later.

Red Hat Unix 9.0 during the installation process, creates two directories, floppy and cdrom, under the /mnt directory. These are intended as mount-points for a floppy and a CD-ROM, respectively. To mount a DOS floppy, use the command:

```
#       mount       -t      ms dos      /dev/fd0      /mnt/floppy
```

To mount a Unix floppy, use the command:

```
#       mount       -t      ext2    /dev/fd0      /mnt/floppy
```

Of course, you can use any other existing directory as your mount-point. A CD-ROM has to be mounted as read-only, for obvious reasons. This is done with the –r option of the mount command:

```
#       mount       -t      iso9660     -r      /dev/hdb      /mnt/cdrom
```

Assuming the CD-ROM drive is the second device on the primary IDE bus, as is usually the case.
If you issue the mount command without any arguments, it will output a list of all currently mounted file systems and their mount-points:

```
#       mount
/dev/hda3       on      /       type    ext2    (rw)
None    on      /proc   type    proc    (rw)
/dev/hda1       on      /dosc   type    vfat    (rw)
None    on      /dev/pts        type    devpts  (rw, mode=0622)
```

This output shows that two user file systems are mounted on / (root directory) and / dosc and two system virtual file systems are also mounted. It is also clear that the file system mounted on the root directory is a Unix native file system, and that on /dosc is a Windows 95/98 FAT-32 file system. Both are mounted as read-write, i.e., both reading and writing are permitted.

Un-mounting File systems:

The reverse process of mounting file systems is to un-mount them. Unmounting a file stem has two effects:
The system buffers are synchronized (sync –ed) with the actual contents of the
file system on disk.
The file system is no longer accessible at the mount-point. You can now access the
previous contents of the mount point or mount another file system on it.

Unmounting is done with the unmount command (note the spelling):

```
#       umount /mnt/floppy
```

This will un-mount the file system mounted on the directory /mnt/floppy.

It is very important to note that removable media, such as floppies or CD-ROM's, should not be removed from the drives or changed to another media while mounted. This will cause the system's information on the device to be out of sync with what is actually there and could lead to data loss. Whenever you want or remove or switch a media, un-mount it first.

Reads and writes on mounted file systems occur asynchronously. For example, if you have issued a cp command to copy a file on to a mounted floppy, the actual writing of data may not occur immediately. The system handles I/O asynchronously and reads and writes data only when absolutely necessary. You can, however, user the sync command to force the system to write all unwritten buffered data immediately:

```
# 	sync
```

Unmounting a file system also causes this to happen.

Lastly, note that you cannot un-mount a file system while your current directory is the mount-point or any of the mounted file system's directories. Change directory first using the cd command and then un-mount the file system.

The /etc/fstab File

During booting, several file systems are mounted automatically. This is handled by the special configuration file /etc/fstab (file system table). This file contains an entry for each file system to be mounted at booting time. Each entry has six fields of the following format:

```
Device mount-point type options dump order
```

Where device is the name of the device containing the file system, mount-point is the directory on which to mount the file system, type is the file system type, options specify mount options. The dump field specified whether the file system should be included in a backup using the dump command (a 0 means no, a 1 means yes). The order field contains a digit that specifies the order of checking this file system at boot time. A sample /etc/fstab is shown below:

```
# cat /etc/fstab
/dev/hda3        /              ext2    defaults              1  1
/dev/hda2        swap           swap        defaults          0  0
/dev/hda1        /dosc          vfat        defaults          1  2
/dev/fd0         /mnt/floppy    ext2        user, noauto      0  0
/dev/cdrom       /mnt/cdrom     iso9660     user, noauto,ro 0  0
none             /proc          proc        defaults          0  0
none             /dev/pts                   devpts  mode=0622     0  0
```

Observe that the Unix native ext2 partition at /dev/hda3 is mounted on the root directory and the Win 95/98 partition at /dev/hda1 is mounted on /dosc at boot time. Both of them will be included for backup by a dump command and the Unix partition will be checked first, followed by the Win 95 partition. We also observe some mount options in the fourth column of the /etc/fstab file. The commonly used mount options are described below. The following can either be specified with the – 0 option of the mount command followed by a comma-separated list of these option or these may be specified in the fourth field in the /etc/fstab file:

Option	Meaning
async	All I/O to be done with the mounted file system asynchronously.
auto	The file systems with this option specified in the /etc/fstab file will be mounted automatically with the –a option of the mount command. The inverse of this option is noauto.
dev	Interpret character or block special devices on the file system.

exec	Permits execution of binaries on the file system. Opposite is noexec.
rw	Mount the file system in read-write mode.
ro	Mount the file system in read-only mode.
user	Allow ordinary users to mount the file system. Opposite in nouser.
suid	Allow suid and sgid bits to take effect on the filesystem.
Defaults	Specifies default options that include: asyn, nouser, exec, rw, dev, suid and auto.

The command

```
Mount –a
```

Will mount all file systems listed in /etc/fstab. This command is executed at boot time by the script /etc/rc.d/rc.sysinit. This way, all file systems listed in /etc/fstab will be available when the system starts up.

There is one exception to this rule: the root file system. The root file system, mounted at /, usually will contain the files /etc/fstab and /etc/rc.d/rc.sysinit. Hence, the kernel mounts the root file system directly. The name of the device containing the foot file system (such as /dev/hda3) is hard coded into the kernel itself. This can be later altered by using the rdev (root device) command. While booting, the kernel attempts to mount this device.

A file system need not be listed in /etc/fstab in order to be mounted. But, it needs to be listed in order to be mounted automatically with the mount –a command or to use the user mount option.

Creating File systems

You use the command fd0 to create file systems. This is equivalent to formatting a floppy or a hard disk partition, so that the floppy or the partition can store files. The syntax is

```
mkfs –t type device blocks
```

Where type specifies the file system type, device is the name of the device and blocks is the size of the file system in 1KB blocks. Actually, the mkfs command is a front-end for the commands mkfs.ext2 for Unix file systems, mkfs.msdos for MS-DOS file systems etc. To create a ext2 file system on a floppy, use the command:

```
#       mkfs  –t ext2   /dev/fd0 1440
```

Here, 1440 indicates the size of the file system 1440 KB or 1.44 MB, i.e., a high-density, 3.5" floppy. You can create an MS-DOS file system INSTEAD USING THE –T MSDOS OPTION.

After creating the file system, the floppy can be mounted and used. Note that the mkfs command will destroy all existing data on the device, without warning. File systems on hard disk partitions are also created in exactly the same way, except the device name should be that of a hard disk partition (such as /dev/hda2 or /dev/sda4) and blocks should be the number of 1 KB blocks in the partition.

Using Floppies

For preparing floppies for use, it is always best to perform a low-level format on the floppy first. This can be done by the fdformat command of Unix. This command accepts the name of the raw device (unformatted floppy representation in /dev) as its argument. For the first 3.5” 1.44 MB floppy drive (A: in MS-DOS), the raw device name is fdoH1440. Thus, the following command will format a floppy:

```
#       fdformat  /dev/fdoH1440
```

After formatting, you must create a file system on the floppy. For example, the following command will create an ext2 file system on the formatted floppy:

```
#       mkfs –f  ext2 –c /dev/fd0  1440
```

Note that this time we are using the usual device name, fd0. The –c option of the mkfs command will check for bad blocks on the device. Next, we must mount the floppy:

```
#       mount –t ext2 /dev/fd0  /mnt/floppy
```

Now, you may access the file system on the floppy. An MS-DOS floppy can also be similarly created and used.

Checking Free Space: The df Command

The df (disk free) command shows you the details of space in all mounted file systems, including mounted floppies and CD-ROM’s. Invoked without any options, this command displays the total, occupied and free space sizes in un its of 1 KB:

```
#       df

File system     1 k-blocks     used      Available        use%      Mounted on
/dev/hda3          2004833   1429040       472174         75%       /
/dev/hda1          2044232    987148      1057084         48%       /dosc
/dev/fd0              1423       842          581         59%       /mnt/floppy
```

This output shows details of three file systems mounted. The second column shows the total size of the file system in 1 KB blocks. Other columns are self-explanatory. The df command has the –h (human) option that makes its output easier to comprehend:

```
#       df –h

File system     Size      used      Avail     Use%      Mounted on
/dev/hda3       1.9G      1.4g      461M      75%       /
/dev/hda1       1.9G      964M      1.0G      48%       /dosc
/dev/fd0        1.4M      842K      581k      59%       /mnt/floppy
```

Checking and Repairing File systems

Sometimes, it is necessary to check a file system and repair errors on it, if any. Such errors may occur in case of a sudden power failure when the kernel did not get a chance to write buffered data to disk and hence, the disk remained out of sync or due to a system crash or due to an error in using powerful commands such as dd etc. In most cases file system errors will be minor and can be repaired.

You use the fsck command to check and optionally repair Unix file systems. Like mkfs, this command is also a front-end for the actual command, fsck.exst2 or e2fsck. The basic syntax of the command is:

```
Fsck –t type device
```

Where type is the type of file system to check and device is the device containing the file system. For example, to check the Unix file system on the file system on the partition /dev/hdc2, we issue the command:

```
#         /sbin/fsck –t ext2  /dev/hdc2
Parallel zing fsck version 1.14  (9-Jan-1999)
E2fsck 1.14, 9-Jan-1999 for EXT2 FS 0.5b, 95/08/09
/dev/hdc2:      clean, 45490/491520 files,      904803/1963584 blocks
```

In this case fsck does not report any error. Most of the errors will be repaired automatically by fsck and output of the command will report the same, giving details of the error. Note that fsck can handle multiple file systems residing on separate disks by running fsck on all of them in parallel. It takes advantage of the parallelism of disk spindle rotation for doing so and this saves the total run time for the command.

You should avoid running fsck on mounted file systems. This is due to the fact that the changes made by fsck in correcting problems on the mounted file system in question makes the state of the file system different from that maintained by the kernel. Hence, the file system becomes out of sync with the system. If at all you attempt to run fsck on a mounted file system, the command will ask for confirmation before that the file system is in sync. But, it is best to avoid running fsck on mounted file systems. Un-mount the file system first and then run fsck on it.

So, how to check the root file system? You cannot un-mount the root file system, and to run fsck, the foot file system must be mounted (as fsck will reside there), but it will not be advisable to run the command on the mounted file system. There are several ways to do this. The best method is to use a boot floppy that we will discuss shortly.

There are several useful options to the fsck command. These are described below:

Option	**Meaning**
-v	Prints verbose information during the checking process.
-c	Performs bad block checking also
-R	Excludes the root file system from checking. Used along with the -A option.
-A	Checks all file systems listed in the /etc/fstab file.

Note that you cannot use the fsck command to check MS-DOS or Windows 95 partitions or floppies. For checking and repairing such file systems, use tools such as Norton Utilities or Microsoft Scandisk.

Managing Swap Space

Swap Space is a generic term for disk storage used to increase the amount of apparent memory available in the system. In Unix, Swap space is used to implement paging, a process in which memory pages (each page is usually 4 KB) are written out to the area when availability of physical memory is low and read back into physical memory when these pages are needed again. The virtual memory system used by Unix allows memory pages to be shared by running programs. For instance, if you have several instances of the vi editor running at any time, there will be only one copy of vi code in memory.

The use of swap space cannot completely make up for the lack of physical memory, since memory access is much faster than disk access. Swap is primarily used to allow multiple programs to run simultaneously, which, otherwise would not fit into the available physical memory.

Unix supports swap space in two forms: a separate swap partition on the disk or swap file located some where on the file system. You can have up to 32 swap areas, with each area being a disk partition or file up to 128 MB in size. This allows an amazing total of up to 2 gigabytes of swap space. Generally, a disk swap partition will yield a much faster operation than a swap file, since the blocks in a disk swap partition are guaranteed to be contiguous, but the blocks in a swap file may be scattered across the disk. Swap files are primarily added in case your existing swap partition does not contain enough swap space.

During installation, you generally create a swap partition on disk. Use the free command to check the amount of swap space you are currently having:

```
    # free
        Total      used          free     shared       buffers      cached

Mem:    63200      64244         1956     62272          1708        33368
-/+ buffers/cache:          26168          37032
Swap: 104416                 3476           100940
```

All the numbers above are in units of 1 KB blocks. Here, we see a system with 63200 blocks (about 62 MB) of physical RAM, with 61244 blocks (bout 60 MB) currently in use. Note that the system actually has more physical RAM than given in the “total” column, as this number does not include the memory used by the kernel for its own need. The “shared” column shows the amount of physical RAM that is shared between processes. The “cached” column indicates RAM (about 32 MB) that is being used by the kernel buffer caches. Memory from this amount will be reclaimed if it’s needed by applications. Thus, we see that although about 60 MB of RAM is being shown as used, not all of it is actually being used by application programs.

On the last line, we see that the total swap space is 104416 blocks (about 102 MB), in which about 3 MB is being currently used. If new applications are now launched, RAM from the kernel buffer caches will be allotted. Swap memory is used as a last resort when the system cannot claim physical RAM.

Creating Swap Space

You use the fdisk command to create a swap partition on disk. We will discuss fdisk in the next session. In order to create a swap file, you first need to create a blank file of size equal to the intended size of the swap area. This can easily be done with the dd command. For example,

```
    #       dd if=/dev/zero        of=/swap       bs=lk   count=16348
```

Here, we are writing 16 MB of null data in the file /swap, making this a 16 MB file containing nothing. The /dev/zero file is a special file which returns null bytes (bytes with ASCII code 0) for every read request on it. Next, we sync the filesystem:

```
    #       sync
```

Now, we can use the mkswap command to “format” this file as a swap file:

```
    #       mkswap –c  /swap  16384
```

The –c option will check the space for bad blocks, the /swap argument specifies the just-created file and the last argument specifies the number of blocks in the swap area. After running this command, we should run the sync command once more to ensure that the format information has indeed been written to the file.

The mkswap command can also be used to create swap partitions on disk. In that case, the /swap argument above should be replaced by the disk partition name (such as /dev/hda2) and the last argument should be the number of blocks in the partition.

Enabling the Swap Space

After the swap space has been created, you must enable the same in order that the system can start to utilize the space as a swap space. This is done by the swapon command:

```
#	swapon /swap
```

This should add this swap file to the available swap areas of the system; verify this with the free command.

Swap spaces are automatically enabled at system boot time, if they are listed in the /etc/fstab file with the work swap in the mount-point and type columns. This is done by the swapon –a command from the system startup file, /etc/rc.d/rc.sysinit.

Disabling Swap Space

Swap spaces are disabled using the swapoff command, followed by the name of the swap partition or file. To disable the swap partition /dev/hda2, we issue the command:

```
#	swapoff  /dev/hda2
```

If it’s a swap file instead, you can delete the file after disabling swap on it. Deleting a file that is being used as a swap file will cause disaster for the system.

Disaster Recovery

As a system administrator, you must be well prepared to meet emergency situation s in the system. Most of the time, this situation will be in the form of a corrupted kernel image (vmlinuz) or a corrupted root filesystem that refuses to be mounted. In both these cases, you simply cannot boot up your system. The most effective solution is to create a boot disk and use this disk to boot up the system and then trying to repair the error.

Red Hat Unix comes with an excellent utility, mkbootdisk, for creating boot disks. This utility will create a boot floppy compatible with the current kernel and root file system. The only required argument is the kernel version found in your /boot directory. The kernel image file is kept in the /boot directory. The name of the kernel used in the system will be vmlinuz followed by the version number of the kernel. This version number will have to be supplied as an argument to the mkbootdisk command. To find out the version number, use the ls –l command:

```
$ ls –l /boot/vmlinuz -*
-rw –r –r –  1 root root 617431  Apr  20 08:06   /boot/vmlinuz –2.2.5 –15
-rw –r –r –  1 root root 530928  Apr  20 07:07   /boot/vmlinuz –2.2.5 –15Boo
-rw –r –r –  1 root root  640798 Apr   20 07:49  /boot/vmlinuz –2.2.5 –15 smp
```

Ignoring the BOO and smp files, we note that the working kernel for our system is a file named vmlinuz –2.2.5. So, the version number is –2.2.5 –15. Now, we invoke the mkbootdisk command with the –verbose option, so that we can watch what the command is doing:

```
#       /sbin/mkbootdisk        –verbose 2.2.5 –15 (kernel version)
Insert a disk in /dev/fd0.  Any information on the disk will be lost.
Press <Enter> to continue or ^c to abort: <press Enter here>
Formatting .dev/fd0… done.
Copying /boot/vmlinuz –2.2.5 –15… done.
Setting up lilo… done.
Added Unix  *
Added rescue
```

After you insert a floppy and press the Enter key, it is first formats the floppy. Next, it copies the kernel image to the floppy. Then, an initial ramdisk (a disk drive implemented in RAM) image is created that will load any SCSI modules, if needed. This is called an initial image. Lastly, the LILO boot loader is added that'll provide the LILO boot: prompt when you boot your system with this boot floppy.

At the LILO prompt, you will have two choices: you simply press the Enter key (or key in the word Unix) or you key in the work rescue. In the former case, the machine will boot with the kernel image on the floppy, but with your usual root file system on your hard disk. Hence, this should be done when you have a corrupted kernel image on your hard disk. After the system is booted like this, you can correct the damaged kernel image by copying a previously saved kernel image (from the boot floppy created during installation) or by re-compiling a new kernel image. After doing this, you should be able to boot your system normally.

If you type rescue at the LILO prompt, the kernel image from the boot floppy will load into memory, and then you will be prompted to insert a floppy containing a compressed root file system. You can easily prepare such a floppy from your Red Hat installation CD-ROM. The compressed root file system is kept on the CD-ROM in the file resue.img in the /images directory on the CD. Issue the following commands to create a rescue disk containing the root filesystem:

```
#       mount –t iso9660 –r /dev/hdb bmnt/cdrom
#       cd /mnt/cdrom/images
#       dd if= rescue.img of=/devg/fd0  bs=lk
```

This will create the necessary rescue disk. Insert this disk when prompted and you machine will boot with this root file system. This file system contains most of the commands you will need for any repair work, like fsck. It even contains a small text editor, pico. Now you can check and repair your root file system by running fsck on the relevant hard disk partition.

GUI Tool for Reviewing Your File system

You may use the Unixconf utility for a graphical view of your current file system and add or edit entries in /etc/fstab. For this, follow the following steps:

Become super user with the su command and type Unixconf. This will start the utility.
Follow Config -> File systems -> Access local drive and click on it.
The right pane will change to show the current /etc/fstab.

Now, you can click on an entry to edit the same or use the Add button to add new entries in /etc/fstab. Moreover, when you select a mountable entry, the resulting screen allows you to mount or un-mount the file system.

Appendix

Unix Command Summary

Unix Commands

In the table below we summarize the more frequently used commands on a Unix system. In this table, as in general, for most Unix commands, *file*, could be an actual file name, or a list of file names, or input/output could be redirected to or from the command.

TABLE : Unix Commands

Command/Syntax	**What it will do**
awk/nawk [options] *file*	scan for patterns in a file and process the results
cat [options] *file*	*concatenate (list) a file*
cd [directory]	change directory
chgrp [options] *group file*	change the group of the file
chmod [options] *file*	change file or directory access permissions
chown [options] *owner file*	change the ownership of a file; can only be done by the superuser
chsh (*passwd -e/-s*) *username login_shell*	change the user's login shell (often only by the superuser)
cmp [options] *file1 file2*	compare two files and list where differences occur (text or binary files)
compress [options] *file*	compress file and save it as *file.Z*
cp [options] *file1 file2*	copy *file1* into *file2*; *file2* shouldn't already exist. This command creates or overwrites *file2*.
cut (options) [*file*(s)]	cut specified field(s)/character(s) from lines in file(s)

date [options]	report the current date and time
dd [if=infile] [of=outfile] [operand=value]	copy a file, converting between ASCII and EBCDIC or swapping byte order, as specified
diff [options] *file1 file2*	compare the two files and display the differences (text files only)
df [options] [resource]	report the summary of disk blocks and inodes free and in use
du [options] [*directory* or *file*]	report amount of disk space in use
echo [text string]	echo the text string to stdout
ed or *ex* [options] *file*	Unix line editors
emacs [options] *file*	full-screen editor
expr arguments	evaluate the arguments. Used to do arithmetic, etc. in the shell.
file [options] *file*	classify the file type

Command/Syntax	**What it will do**
find directory [options] [actions]	find files matching a type or pattern
finger [options] *user[@hostname]*	report information about users on local and remote machines
ftp [options]	*host* transfer file(s) using file transfer protocol
grep [options] 'search string' *argument* *egrep* [options] 'search string' *argument* *fgrep* [options] 'search string' *argument*	search the argument (in this case probably a file) for all occurrences of the search string, and list them.

gzip [options] *file* *gunzip* [options] *file* *zcat* [options] *file*	compress or uncompress a file. Compressed files are stored with a .gz ending
head [-number] *file*	display the first 10 (or number of) lines of a file
hostname	display or set (super-user only) the name of the current machine
kill [options] [-SIGNAL] [pid#] [%job]	send a signal to the process with the process id number (pid#) or job control number (%n). The default signal is to kill the process.
ln [options] *source_file target*	link the *source_file* to the *target*
lpq [options] *lpstat* [options]	show the status of print jobs
lpr [options] *file* *lp* [options] *file*	print to defined printer
lprm [options] *cancel* [options]	remove a print job from the print queue
ls [options] [*directory* or *file*]	list *directory* contents or *file* permissions
mail [options] [user] *mailx* [options] [user] *Mail* [options] [user]	simple email utility available on Unix systems. Type a period as the first character on a new line to send message out, question mark for help.
man [options] *command*	show the manual (man) page for a command
mkdir [options] *directory*	make a *directory*
more [options] *file* *less* [options] *file* *pg* [options] *file*	page through a text file

mv [options] *file1 file2*	move *file1* into *file2*
od [options] *file*	octal dump a binary file, in octal, ASCII, hex, decimal, or character mode.
passwd [options]	set or change your password
paste [options] *file*	paste field(s) onto the lines in *file*
pr [options] *file*	filter the file and print it on the terminal
ps [options]	show status of active processes

Command/Syntax	**What it will do**
pwd	print working (current) directory
rcp [options] *hostname*	remotely copy files from this machine to another machine
rlogin [options] *hostname*	login remotely to another machine
rm [options] *file*	remove (delete) a file or directory (-r recursively deletes the directoryand its contents) (-i prompts before removing files)
rmdir [options] *directory*	remove a *directory*
rsh [options] *hostname*	remote shell to run on another machine
script file	saves everything that appears on the screen to file until *exit* is executed
sed [options] *file*	stream editor for editing files from a script or from the command line
sort [options] *file*	sort the lines of the *file* according to the options chosen
source file *. file*	read commands from the *file* and execute them in the current shell.
source: C shell, .: Bash shell.	

strings [options] *file*	report any sequence of 4 or more printable characters ending in <NL> or <NULL>. Usually used to search binary files for ASCII strings.
stty [options]	set or display terminal control options
tail [options] *file*	display the last few lines (or parts) of a file
tar key[options] [*file*(s)]	tape archiver--refer to man pages for details on creating, listing, and
retrieving from archive files. Tar files can be stored on tape or disk.	
tee [options] *file*	copy stdout to one or more files
telnet [host [port]]	communicate with another host using telnet protocol
touch [options] [date] *file*	create an empty file, or update the access time of an existing file
tr [options] *string1 string2*	translate the characters in string1 from stdin into those in string2 in stdout
uncompress file.Z	uncompress *file.Z* and save it as a file
uniq [options] *file*	remove repeated lines in a file
uudecode [*file*]	decode a uuencoded file, recreating the original file
uuencode [*file*] *new_name*	encode binary file to 7-bit ASCII, useful when sending via email, to be decoded as new_name at destination
vi [options] *file*	visual, full-screen editor
wc [options] [*file*(s)]	display word (or character or line) count for *file*(s)
whereis [options] *command*	report the binary, source, and man page locations for the command named

Which	*command* reports the path to the command or the shell alias in use
who or *w*	report who is logged in and what processes are running
zcat file.Z	concatenate (list) uncompressed file to screen, leaving file compressed on disk

Lab Exercises for Session 1

1. Log in to the Unix system for the first time with your assigned log in name and password.
2. Change your password to some other password and remember your new password.
3. Log out of Unix.
4. Log in again with the new password.
5. Check out which virtual terminal you are using and who else are logged in at the moment.
6. Switch to the other virtual terminals in your machine and observe the screens.
7. Check your present working directory.
8. Create to the test1, test2, and test3 under your home directory.
9. Change to the test1 directory.
10. Create a text file file1.txt with the following contents:

Many programmers – experienced and novice alike – have begun their adventure with Java.

11. Display the contents of file1.txt on the screen.
12. Use the ls –l command to check the size of the file file1.txt.
13. Now append the following text to the file file1.txt.

In a flurry of well-deserved attention Java has brought life to the web, whipped the public and professional press into a frenzy, and is on the verge of challenging our assumptions of what networked computing is all about.

14. Display the contents of file1.txt and check whether the above text has been appended.

15. Remove the directory test3.
16. Switch to the directory test2 with one command.
17. Switch back to your home directory.
18. Browse through the man and info pages of the commands cat, ls and cp.
19. Log out of Unix.

Lab Exercises for Session 2

1. Rename the test1 directory and the file1.txt file your had created in the precious session to exercises and java.txt, respectively.

2. Change the permissions of java.txt to rw - - - - - - -, using the octal notation of the chmod command.

3. Add the read permission for the group and other, using the symbolic notation of chmod.
Check whether the permissions have been set correctly.

4. Check the current value of your mask by issuing the umask command without any parameters.

5. Create a text file outputs containing the outputs of the command date, followed by the output of the command who, followed by the output of the command ls –l.

6. Create a hard ling outputs. hard link to the file outputs and a symbolic link outputs.sym to the Same file. Check whether the links have been created correctly.

7. Now, delete the file outputs. Display the contents of the files outputs.sym and outputs.hard.

8. Explain the difference.

9. Create suitable DOS-equivalent aliases for the commands mv, ls –l, cp and rm. Check whether these aliases have been set correctly. Now, try to execute these alias commands. (Copy existing files to temporary files to experiment).

10. Set your primary prompt so that it shows your current directory followed by the > sign.

11. Switch to the /etc directory. List all the files there whose names end with conf, whose names start with ftp, whose names start with a capital letter and whose names have the letter i in the second position.

12. Switch back to your home directory.

13. Check the value of your HOME, LOG NAME, PATH and SHELL variables.

14. Check the prompt sting set for your log in account. Interpret the meaning of the string and compare it with the prompt you are getting.

15. Browse the man and info pages of the commands ln, chmod and umask.

Lab Exercises for Session 4

1. Use the wc command to count the number of words, lines and characters in the UNIX.doc file you created in the last session. Check the character count reported by wc with that reported by ls –l listing of the same file.
2. Display lines 11 through 19 of the UNIX.doc.
3. Construct a command pipe to report the number of entries in the /dev directory.
4. Construct a command pipe to display only the directory entries in the directory / etc.
5. Construct a command pipe permissions, file sizes and file names from an ls –l listing.
6. Construct a command pipe to display only the names of users currently logged in
7. Log in at a few virtual consoles. Fire different jobs, like the vi editor, the less command, the joe editor etc. in each of these consoles. Now issue the ps and ps –a commands from a fresh console and observe the outputs.
8. Construct a command pipe to display only the process ID's and the names of the processes running on different virtual consoles.
9. Kill the vi process, switch to the console in which vi was running and observe what happened.
10. Using a text editor, create a text file inventory and put the following data in it:

Item No.	Item Name	Unit Cost	Quantity
Coo2	Color TV 36cm.	9000/-	10
Bw20	B/W TV 36cm.	2,900/-	5
C009	Color TV 53cm.	12000/-	12
MS05	Music System 500W	11700/-	8
BW31	B/W TV 51cm.	5200/-	2
T006	Two-in-one	1800/-	7

Display only the item no. and item name columns sorted by item no. in the ascending order.

11. Construct a command pipe to display a count of only the B/W TV entries.

12. Construct a command pipe to display only the item name and unit cost fields by using the field mode of the cut command.

Lab Exercises for Session 5

1. Log in with your usual account and become super user with the su command. Add a couple of users with the useradd command, with default user ID, group, home directory and shell. Assign passwords to these accounts.

2. Check the /etc/passwd and /etc/shadow files for entries of these new accounts. Now, log in from a virtual terminal into these new accounts and observe what skeleton files are automatically copied into their home directories. Log out of these accounts and return to your graphical screen.

3. Start the Unixconf utility and display the user accounts screen. Check the entries of the newly created accounts. Try editing these accounts and then delete one of these accounts.

4. Now, disable the remaining account by editing /etc/passwd. Confirm this by trying to log in to this account.

5. Use the df and free commands to check currently mounted filesystems and free RAM and swap spaces.

6. Format a floppy and create a Unix filesystem on it. Mount the floppy and copy some files on it. Now use the df command again to confirm the floppy is mounted and the occupied and free spaces on it. Also, taken an ls –l listing of the mount-point to check the files are copied. Un-mount the floppy.

7. Check whether any Windows 95/98 partitions are mounted. If not, mount it and copy some text files from that partition into your home directory. Open these text files in Unix to verify the cross-partition copy.

8. Create emergency boot and rescue diskettes and boot up the PC with these. First, instruct the LILO prompt to use the root filesystem on the hard disk. Observe that you can now work normally on Unix. Next, boot up the root filesystem on the rescue disk and check what tools and programs are available for you (to use in an emergency) in the /bin directory.