UNIX AND SHELL PROGRAMMING
Scheme and Syallbus

Subject Code: 10CS44	**I.A. Marks : 25**	**Hours/Week : 04**
Exam Hours: 03	**Total Hours : 52**	**Exam Marks: 100**

PART – A

UNIT 1:

1. The UNIX Operating System, the UNIX architecture and Command Usage, The File System **6 Hours**

UNIT 2:

2. Basic File Attributes, The vi Editor **6 Hours**

UNIT 3:

3. The Shell, The Process, Customizing the environment **7 Hours**

UNIT 4:

4. More file attributes, Simple filters **7 Hours**

PART – B

UNIT 5:

5. Filters using regular expressions, **6 Hours**

UNIT 6:

6. Essential Shell Programming **6 Hours**

UNIT 7:

7. awk – An Advanced Filter **7 Hours**

UNIT 8:

8. perl - The Master Manipulator **7 Hours**

Text Book

1. **"UNIX – Concepts and Applications",** Sumitabha Das, 4th Edition, Tata McGraw Hill, 2006.

(Chapters 1.2, 2, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 18, 19).

Reference Books

UNIX and Shell Programming, Behrouz A. Forouzan and Richard F. Gilberg, Thomson, 2005.

Unix & Shell Programming, M.G. Venkateshmurthy, Pearson Education, 2005.

Table of Contents

Sl No	Unit description	Page no
1	Unit 1 The Unix Operating System	1-19
2	Unit 2 Basic File Attributes	20-34
3	Unit 3 The Shell, The Process	35-62
4	Unit 4 More file attributes	63-77
5	Unit 5 Filters using regular expressions	78-89
6	Unit 6 Essential Shell Programming	90-124
7	Unit 7 awk – An Advanced Filter	125-146
8	Unit 8 perl - The Master Manipulator	147-160

UNIT 1

. The Unix Operating System, The UNIX architecture and Command Usage, The File System

6 Hours

Text Book

1. **"UNIX – Concepts and Applications",** Sumitabha Das, 4th Edition, Tata McGraw Hill, 2006.

(Chapters 1.2, 2, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 18, 19).

Reference Books

UNIX and Shell Programming, Behrouz A. Forouzan and Richard F. Gilberg, Thomson, 2005.

Unix & Shell Programming, M.G. Venkateshmurthy, Pearson Education, 2005.

The UNIX Operating System

Introduction

This chapter introduces you to the UNIX operating system. We first look at what is an operating system and then proceed to discuss the different features of UNIX that have made it a popular operating system.

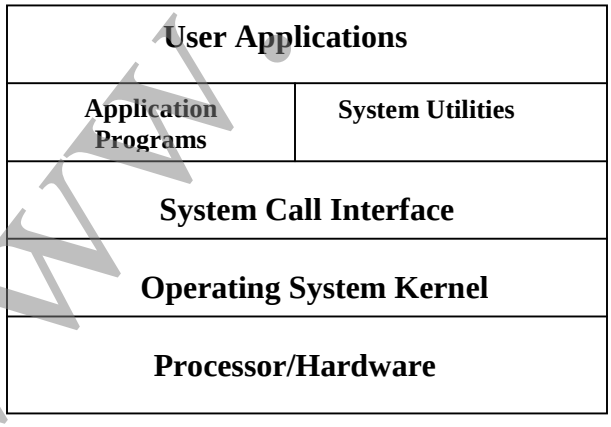
Objectives

- What is an operating system (OS)?
- Features of UNIX OS
- A Brief History of UNIX OS, POSIX and Single Unix Specification (SUS)

1. What is an operating system (OS)?

An operating system (OS) is a resource manager. It takes the form of a set of software routines that allow users and application programs to access system resources (e.g. the CPU, memory, disks, modems, printers, network cards etc.) in a **safe, efficient** and **abstract** way.

For example, an OS ensures **safe** access to a printer by allowing only one application program to send data directly to the printer at any one time. An OS encourages **efficient** use of the CPU by suspending programs that are waiting for I/O operations to complete to make way for programs that can use the CPU more productively. An OS also provides convenient **abstractions** (such as files rather than disk locations) which isolate application programmers and users from the details of the underlying hardware.



UNIX Operating system allows complex tasks to be performed with a few keystrokes. It doesn't tell or warn the user about the consequences of the command.
Kernighan and Pike (The UNIX Programming Environment) lamented long ago that "as the UNIX system has spread, the fraction of its users who are skilled in its application has decreased." However, the capabilities of UNIX are limited only by your imagination.

2. Features of UNIX OS

Several features of UNIX have made it popular. Some of them are:

Portable
UNIX can be installed on many hardware platforms. Its widespread use can be traced to the decision to develop it using the C language.

Multiuser
The UNIX design allows multiple users to concurrently share hardware and software

Multitasking
UNIX allows a user to run more than one program at a time. In fact more than one program can be running in the background while a user is working foreground.

Networking
While UNIX was developed to be an interactive, multiuser, multitasking system, networking is also incorporated into the heart of the operating system. Access to another system uses a standard communications protocol known as Transmission Control Protocol/Internet Protocol (TCP/IP).

Organized File System
UNIX has a very organized file and directory system that allows users to organize and maintain files.

Device Independence
UNIX treats input/output devices like ordinary files. The source or destination for file input and output is easily controlled through a UNIX design feature called redirection.

Utilities
UNIX provides a rich library of utilities that can be use to increase user productivity.

3. A Brief History of UNIX

In the late 1960s, researchers from General Electric, MIT and Bell Labs launched a joint project to develop an ambitious multi-user, multi-tasking OS for mainframe computers known as MULTICS (Multiplexed Information and Computing System). MULTICS failed, but it did inspire Ken Thompson, who was a researcher at Bell Labs, to have a go at writing a simpler operating system himself. He wrote a simpler version of MULTICS on a PDP7 in assembler and called his attempt UNICS (Uniplexed Information and Computing System). Because memory and CPU power were at a premium in those days, UNICS (eventually shortened to UNIX) used short commands to minimize the space needed to store them and the time needed to decode them - hence the tradition of short UNIX commands we use today, e.g. `ls, cp, rm, mv` etc.

Ken Thompson then teamed up with Dennis Ritchie, the author of the first C compiler in 1973. They rewrote the UNIX kernel in C - this was a big step forwards in terms of the system's portability - and released the Fifth Edition of UNIX to universities in 1974. The Seventh Edition, released in 1978, marked a split in UNIX development into two main branches: SYSV (System 5) and BSD (Berkeley Software Distribution). BSD arose from the University of California at Berkeley where Ken Thompson spent a sabbatical year. Its development was continued by students at Berkeley and other research institutions. SYSV was developed by AT&T and other commercial companies. UNIX flavors based on SYSV have traditionally been more conservative, but better supported than BSD-based flavors.

Until recently, UNIX standards were nearly as numerous as its variants. In early days, AT&T published a document called System V Interface Definition (SVID). X/OPEN (now The Open Group), a consortium of vendors and users, had one too, in the X/Open Portability Guide (XPG). In the US, yet another set of standards, named Portable Operating System Interface for Computer Environments (POSIX), were developed at the behest of the Institution of Electrical and Electronics Engineers (IEEE).
In 1998, X/OPEN and IEEE undertook an ambitious program of unifying the two standards. In 2001, this joint initiative resulted in a single specification called the Single UNIX Specification, Version 3 (SUSV3), that is also known as IEEE 1003.1:2001 (POSIX.1). In 2002, the International Organization for Standardization (ISO) approved SUSV3 and IEEE 1003.1:2001.

Some of the commercial UNIX based on system V are:

- IBM's AIX
- Hewlett-Packard's HPUX
- SCO's Open Server Release 5
- Silicon Graphics' IRIS
- DEC's Digital UNIX
- Sun Microsystems' Solaris 2

Some of the commercial UNIX based on BSD are:

- SunOS 4.1.X (now Solaris)
- DEC's Ultris
- BSD/OS, 4.4BSD

Some Free UNIX are:

- Linux, written by Linus Torvalds at University of Helsinki in Finland.
- FreeBSD and NetBSD, a derivative of 4.4BSD

Conclusion

In this chapter we defined an operating system. We also looked at history of UNIX and features of UNIX that make it a popular operating system. We also discussed the convergence of different flavors of UNIX into Single Unix Specification (SUS) and Portable Operating System Interface for Computing Environments (POSIX).

The UNIX Architecture and Command Usage

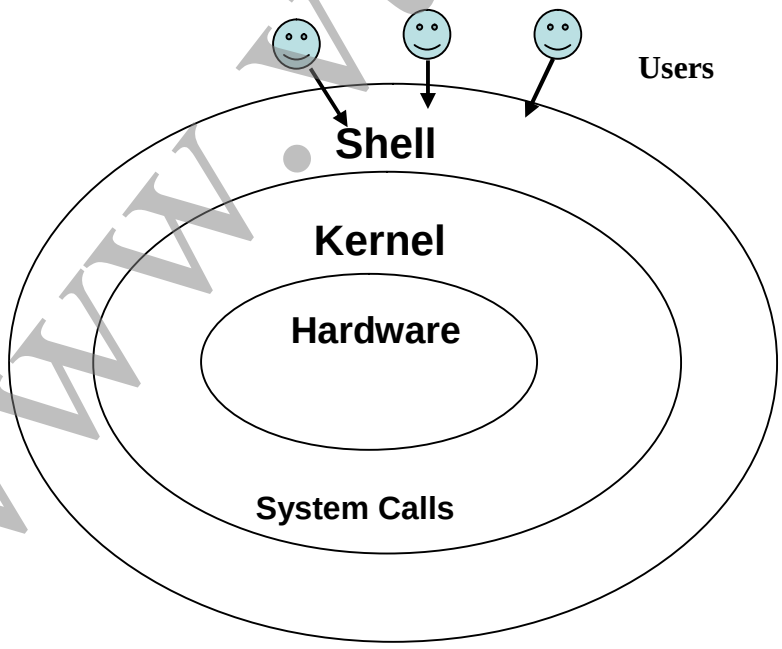
Introduction

In order to understand the subsequent chapters, we first need to understand the architecture of UNIX and the concept of division of labor between two agencies viz., the shell and the kernel. This chapter introduces the architecture of UNIX. Next we discuss the rich collection of UNIX command set, with a specific discussion of command structure and usage of UNIX commands. We also look at the man command, used for obtaining online help on any UNIX command. Sometimes the keyboard sequences don't work, in which case, you need to know what to do to fix them. Final topic of this chapter is troubleshooting some terminal problems.

Objectives

- The UNIX Architecture
- Locating Commands
- Internal and External Commands
- Command Structure and usage
- Flexibility of Command Usage
- The man Pages, apropos and whatis
- Troubleshooting the terminal problems

1. The UNIX Architecture



UNIX architecture comprises of two major components viz., the shell and the kernel. The kernel interacts with the machine's hardware and the shell with the user.

The kernel is the core of the operating system. It is a collection of routines written in C. It is loaded into memory when the system is booted and communicates directly with the hardware. User programs that need to access the hardware use the services of the kernel via use of system calls and the kernel performs the job on behalf of the user. Kernel is also responsible for managing system's memory, schedules processes, decides their priorities.

The shell performs the role of command interpreter. Even though there's only one kernel running on the system, there could be several shells in action, one for each user who's logged in. The shell is responsible for interpreting the meaning of metacharacters if any, found on the command line before dispatching the command to the kernel for execution.

The File and Proces

A file is an array of bytes that stores information. It is also related to another file in the sense that both belong to a single hierarchical directory structure.

A process is the second abstraction UNIX provides. It can be treated as a time image of an executable file. Like files, processes also belong to a hierarchical structure. We will be discussing the processes in detain in a subsequent chapter.

2. Locating Files

All UNIX commands are single words like ls, cd, cat, etc. These names are in lowercase. These commands are essentially *files* containing programs, mainly written in C. Files are stored in directories, and so are the binaries associated with these commands. You can find the location of an executable program using type command:

```
$ type ls
ls is /bin/ls
```

This means that when you execute ls command, the shell locates this file in /bin directory and makes arrangements to execute it.

The Path

The sequence of directories that the shell searches to look for a command is specified in its own PATH variable. These directories are colon separated. When you issue a command, the shell searches this list in the sequence specified to locate and execute it.

3. Internal and External Commands

Some commands are implemented as part of the shell itself rather than separate executable files. Such commands that are built-in are called internal commands. If a command exists both as an internal command of the shell as well as an external one (in /bin or /usr/bin), the shell will accord top priority to its own internal command with the same name. Some built-in commands are echo, pwd, etc.

4. Command Structure

UNIX commands take the following general form:

verb [options] [arguments]

where verb is the command name that can take a set of optional options and one or more optional arguments.

Commands, options and arguments have to be separated by spaces or tabs to enable the shell to interpret them as words. A contiguous string of spaces and tabs together is called a whitespace. The shell compresses multiple occurrences of whitespace into a single whitespace.

Options

An option is preceded by a minus sign (-) to distinguish it from filenames.

Example: $ ls –l

There must not be any whitespaces between – and l. Options are also arguments, but given a special name because they are predetermined. Options can be normally compined with only one – sign. i.e., instead of using

$ ls –l –a –t

we can as well use,

$ ls –lat

Because UNIX was developed by people who had their own ideas as to what options should look like, there will be variations in the options. Some commands use + as an option prefix instead of -.

Filename Arguments

Many UNIX commands use a filename as argument so that the command can take input from the file. If a command uses a filename as argument, it will usually be the last argument, after all options.

Example: cp file1 file2 file3 dest_dir
rm file1 file2 file3

The command with its options and argumens is known as the command line, which is considered as complete after *[Enter]* key is pressed, so that the entire line is fed to the shell as its input for interpretation and execution.

Exceptions

Some commands in UNIX like pwd do not take any options and arguments. Some commands like who may or may not be specified with arguments. The ls command can run without arguments (ls), with only options (ls –l), with only filenames (ls f1 f2), or using a combination of both (ls –l f1 f2). Some commands compulsorily take options (cut). Some commands like grep, sed can take an expression as an argument, or a set of instructions as argument.

5. Flexibility of Command Usage

UNIX provides flexibility in using the commands. The following discussion looks at how permissive the shell can be to the command usage.

Combining Commands

Instead of executing commands on separate lines, where each command is processed and executed before the next could be entered, UNIX allows you to specify more than one command in the single command line. Each command has to be separated from the other by a ; (semicolon).

```
wc sample.txt ; ls –l sample.txt
```

You can even group several commands together so that their combined output is redirected to a file.

```
(wc sample.txt ; ls –l sample.txt) > newfile
```

When a command line contains a semicolon, the shell understands that the command on each side of it needs to be processed separately. Here ; is known as a metacharacter.

Note: When a command overflows into the next line or needs to be split into multiple lines, just press enter, so that the secondary prompt (normally >) is displayed and you can enter the remaining part of the command on the next line.

Entering a Command before previous command has finished

You need not have to wait for the previous command to finish before you can enter the next command. Subsequent commands entered at the keyboard are stored in a buffer (a temporary storage in memory) that is maintained by the kernel for all keyboard input. The next command will be passed on to the shell for interpretation after the previous command has completed its execution.

6. man: Browsing The Manual Pages Online

UNIX commands are rather cryptic. When you don't remember what options are supported by a command or what its syntax is, you can always view man (short for manual) pages to get online help. The man command displays online documentation of a specified command.

A pager is a program that displays one screenful information and pauses for the user to view the contents. The user can make use of internal commands of the pager to scroll up and scroll down the information. The two popular pagers are more and less. more is the Berkeley's pager, which is a superior alternative to original pg command. less is the standard pager used on Linux systems. less if modeled after a popular editor called vi and is more powerful than more as it provides vi-like navigational and search facilities. We can use pagers with commands like ls | more. The man command is configured to work with a pager.

7. Understanding The man Documentation

The man documentation is organized in eight (08) sections. Later enhancements have added subsections like 1C, 1M, 3N etc.) References to other sections are reflected as SEE ALSO section of a man page.

When you use man command, it starts searching the manuals starting from section 1. If it locates a keyword in one section, it won't continue the search, even if the keyword occurs in another section. However, we can provide the section number additionally as argument for man command.

For example, passwd appears in section 1 and section 4. If we want to get documentation of passwd in section 4, we use,

$ man 4 passwd OR $ man –s4 passwd (on Solaris)

Understanding a man Page

A typical man page for wc command is shown below:

```
User Commands                          wc(1)
NAME
     wc – displays a count of lines, words and characters
in a file
SYNOPSIS
     wc [-c | -m | -C] [-lw] [file ...]
DESCRIPTION
     The wc utility reads one or more input files and, by
default, writes the      number of newline characters,
words and bytes  contained in each input file     to the
standard output. The utility also writes a total count for
all named      files, if more than one input file is
specified.
OPTIONS
     The following options are supported:
     -c   Count bytes.
     -m   Count characters.
     -C   same as –m.
     -l   Count lines.
     -w   Count words delimited by white spaces or new line
characters ...
OPERANDS
     The following operand is supported:
     file A path name of an input file. If no file operands
are specified,          the standard input will be used.
EXIT STATUS
     See largefile(5) for the description of the behavior
of wc when     encountering files greater than or equal to
2 Gbyte (2 **31 bytes)
SEE ALSO
     cksum(1), isspace(3C), iswalpha(3C), iswspace(3C),
largefile(5), ...
```

A man page is divided into a number of compulsory and optional sections. Every command doesn't need all sections, but the first three (NAME, SYNOPSIS and DESCRIPTION) are generally seen in all man pages. NAME presents a one-line introduction of the command. SYNOPSIS shows the syntax used by the command and DESCRIPTION provides a detailed description.

The SYNOPSIS follows certain conventions and rules:

- If a command argument is enclosed in rectangular brackets, then it is optional; otherwise, the argument is required.
- The ellipsis (a set if three dots) implies that there can be more instances of the preceding word.
- The | means that only one of the options shows on either side of the pipe can be used.

All the options used by the command are listed in OPTIONS section. There is a separate section named EXIT STATUS which lists possible error conditions and their numeric representation.

Note: You can use man command to view its own documentation ($ man man). You can also set the pager to use with man ($ PAGER=less ; export PAGER). To understand which pager is being used by man, use $ echo $PAGER.

The following table shows the organization of man documentation.

Section	Subject (SVR4)	Subject (Linux)
1	User programs	User programs
2	Kernel's system calls	Kernel's system calls
3	Library functions	Library functions
4	Administrative file formats	Special files (in /dev)
5	Miscellaneous	Administrative file formats
6	Games	Games
7	Special files (in /dev)	Macro packages and conventions
8	Administration commands	Administration commands

8. Further Help with man –k, apropos and whatis

man –k: Searches a summary database and prints one-line description of the command.
Example:

```
$ man –k awk
awk  awk(1)    -pattern scanning and processing language
nawk nawk(1)   -pattern scanning and processing language
```

apropos: lists the commands and files associated with a keyword.
Example:

```
$ apropos FTP
ftp  ftp(1)          -file transfer program
ftpd in.ftpd(1m)     -file transfer protocol server
```

```
ftpusers  ftpusers(4) -file listing users to be disallowed
                      ftp login privileges
```

whatis: lists one-liners for a command.
Example:

```
$ whatis cp
cp   cp(1)          -copy files
```

9. When Things Go Wrong

Terminals and keyboards have no uniform behavioral pattern. Terminal settings directly impact the keyboard operation. If you observe a different behavior from that expected, when you press certain keystrokes, it means that the terminal settings are different. In such cases, you should know which keys to press to get the required behavior. The following table lists keyboard commands to try when things go wrong.

Keystroke or command	**Function**
[Ctrl-h]	Erases text
[Ctrl-c] or Delete	Interrupts a command
[Ctrl-d]	Terminates login session or a program that expects its input from keyboard
[Ctrl-s]	Stops scrolling of screen output and locks keyboard
[Ctrl-q]	Resumes scrolling of screen output and unlocks keyboard
[Ctrl-u]	Kills command line without executing it
[Ctrl-\]	Kills running program but creates a core file containing the memory image of the program
[Ctrl-z]	Suspends process and returns shell prompt; use **fg** to resume job
[Ctrl-j]	Alternative to *[Enter]*
[Ctrl-m]	Alternative to *[Enter]*
stty sane	Restores terminal to normal status

Conclusion

In this chapter, we looked at the architecture of UNIX and the division of labor between two agencies viz., the shell and the kernel. We also looked at the structure and usage of UNIX commands. The man documentation will be the most valuable source of documentation for UNIX commands. Also, when the keyboard sequences won't sometimes work as expected because of different terminal settings. We listed the possible remedial keyboard sequences when that happens.

The File System

Introduction

In this chapter we will look at the file system of UNIX. We also look at types of files their significance. We then look at two ways of specifying a file viz., with absolute pathnames and relative pathnames. A discussion on commands used with directory files viz., cd, pwd, mkdir, rmdir and ls will be made. Finally we look at some of the important directories contained under UNIX file system.

Objectives

- Types of files
- UNIX Filenames
- Directories and Files
- Absolute and Relative Pathnames
- pwd – print working directory
- cd – change directory
- mkdir – make a directory
- rmdir – remove directory
- The PATH environmental variable
- ls – list directory contents
- The UNIX File System

1. Types of files

A simple description of the UNIX system is this:
"On a UNIX system, everything is a file; if something is not a file, it is a process."
A UNIX system makes no difference between a file and a directory, since a directory is just a file containing names of other files. Programs, services, texts, images, and so forth, are all files. Input and output devices, and generally all devices, are considered to be files, according to the system.

Most files are just files, called *regular* files; they contain normal data, for example text files, executable files or programs, input for or output from a program and so on.
While it is reasonably safe to suppose that everything you encounter on a UNIX system is a file, there are some exceptions.
Directories: files that are lists of other files.
Special files or Device Files: All devices and peripherals are represented by files. To read or write a device, you have to perform these operations on its associated file. Most special files are in /dev.
Links: a system to make a file or directory visible in multiple parts of the system's file tree.
(Domain) sockets: a special file type, similar to TCP/IP sockets, providing inter−process networking protected by the file system's access control.
Named pipes: act more or less like sockets and form a way for processes to communicate with each other, without using network socket semantics.

Ordinary (Regular) File

This is the most common file type. An ordinary file can be either a text file or a binary file.

A text file contains only printable characters and you can view and edit them. All C and Java program sources, shell scripts are text files. Every line of a text file is terminated with the *newline* character.

A binary file, on the other hand, contains both printable and nonprintable characters that cover the entire ASCII range. The object code and executables that you produce by compiling C programs are binary files. Sound and video files are also binary files.

Directory File

A directory contains no data, but keeps details of the files and subdirectories that it contains. A directory file contains one entry for every file and subdirectory that it houses. Each entry has two components namely, the filename and a unique identification number of the file or directory (called the *inode number).*

When you create or remove a file, the kernel automatically updates its corresponding directory by adding or removing the entry (filename and inode number) associated with the file.

Device File

All the operations on the devices are performed by reading or writing the file representing the device. It is advantageous to treat devices as files as some of the commands used to access an ordinary file can be used with device files as well.

Device filenames are found in a single directory structure, /dev. A device file is not really a stream of characters. It is the attributes of the file that entirely govern the operation of the device. The kernel identifies a device from its attributes and uses them to operate the device.

2. Filenames in UNIX

On a UNIX system, a filename can consist of up to 255 characters. Files may or may not have extensions and can consist of practically any ASCII character except the / and the Null character. You are permitted to use control characters or other nonprintable characters in a filename. However, you should avoid using these characters while naming a file. It is recommended that only the following characters be used in filenames:

Alphabets and numerals.

The period (.), hyphen (-) and underscore (_).

UNIX imposes no restrictions on the extension. In all cases, it is the application that imposes that restriction. Eg. A C Compiler expects C program filenames to end with .c, Oracle requires SQL scripts to have .sql extension.

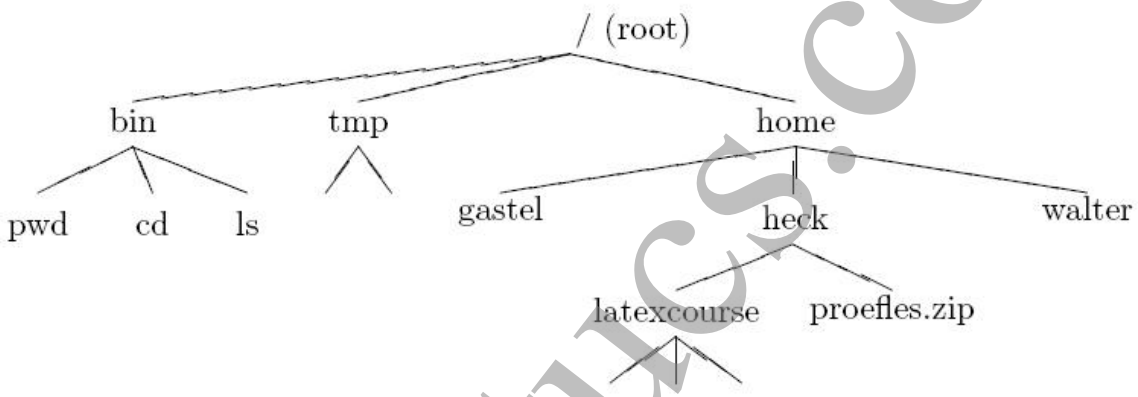
A file can have as many dots embedded in its name. A filename can also begin with or end with a dot.

UNIX is case sensitive; cap01, Chap01 and CHAP01 are three different filenames that can coexist in the same directory.

3. Directories and Files

A file is a set of data that has a name. The information can be an ordinary text, a user-written computer program, results of a computation, a picture, and so on. The file name may consist of ordinary characters, digits and special tokens like the underscore, except the forward slash (/). It is permitted to use special tokens like the ampersand (&) or spaces in a filename.

Unix organizes files in a tree-like hierarchical structure, with the *root directory*, indicated by a forward slash (/), at the top of the tree. See the Figure below, in which part of the hierarchy of files and directories on the computer is shown.



4. Absolute and relative paths

A path, which is the way you need to follow in the tree structure to reach a given file, can be described as starting from the trunk of the tree (the / or root directory). In that case, the path starts with a slash and is called an absolute path, since there can be no mistake: only one file on the system can comply.

Paths that don't start with a slash are always relative to the current directory. In relative paths we also use the . and .. indications for the current and the parent directory.

The HOME variable

When you log onto the system, UNIX automatically places you in a directory called the *home directory*. The shell variable HOME indicates the home directory of the user.

E.g.,

```
$ echo $HOME
/home/kumar
```

What you see above is an absolute pathname, which is a sequence of directory names starting from root (/). The subsequent slashes are used to separate the directories.

5. 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

6. 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 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)

7. 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.

Examples

mkdir patch Creates a directory *patch* under current directory
mkdir patch dbs doc Creates three directories under current directory
mkdir pis pis/progs pis/data Creates a directory tree with *pis* as a directory under the current directory and *progs* and *data* as subdirectories under *pis*

Note the order of specifying arguments in example 3. The parent directory should be specified first, followed by the subdirectories to be created under it.

The system may refuse to create a directory due to the following reasons:
1. The directory already exists.
2. There may be an ordinary file by the same name in the current directory.
3. The permissions set for the current directory don't permit the creation of files and directories by the user.

8. 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 that directory. You cannot remove a subdirectory unless you are placed in a directory which is hierarchically *above* the one you have chosen to remove.

E.g.

rmdir patch Directory must be empty
rmdir pis pis/progs pis/data Shows error as *pis* is not empty. However rmdir silently deletes the lower level subdirectories *progs* and *data*.

9. The PATH environment variable

Environmental variables are used to provide information to the programs you use. We have already seen one such variable called HOME.
A command runs in UNIX by executing a disk file. When you specify a command like *date*, the system will locate the associated file from a list of directories specified in the PATH variable and then executes it. The PATH variable normally includes the current directory also.

Whenever you enter any UNIX command, you are actually specifying the name of an executable file located somewhere on the system. The system goes through the following steps in order to determine which program to execute:
1. Built in commands (such as cd and history) are executed within the shell.
2. If an absolute path name (such as /bin/ls) or a relative path name (such as ./myprog), the system executes the program from the specified directory.
3. Otherwise the PATH variable is used.

10. 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: executables with *, directories with / and symbolic links with @
-R Recursive list
-u Sorts filenames by last access time
-t Sorts filenames by last modification time
-i Displays inode number
-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

Examples

1. To list the files in a directory:

```
$ ls
```

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

```
$ ls -a
```

3. 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 .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
```

11. The UNIX File System

The root directory has many subdirectories. The following table describes some of the subdirectories contained under root.

Directory	Content
/bin	Common programs, shared by the system, the system administrator and the users.
/dev	Contains references to all the CPU peripheral hardware, which are represented as files with special properties.
/etc	Most important system configuration files are in /etc, this directory contains data similar to those in the Control Panel in Windows
/home	Home directories of the common users.
/lib	Library files, includes files for all kinds of programs needed by the system and the users.

/sbin	Programs for use by the system and the system administrator.
/tmp	Temporary space for use by the system, cleaned upon reboot, so don't use this for saving any work!
/usr	Programs, libraries, documentation etc. for all user-related programs.
/var	Storage for all variable files and temporary files created by users, such as log files, the mail queue, the print spooler area, space for temporary storage of files downloaded from the Internet, or to keep an image of a CD before burning it.

Conclusion

In this chapter we looked at the UNIX file system and different types of files UNIX understands. We also discussed different commands that are specific to directory files viz., pwd, mkdir, cd, rmdir and ls. These commands have no relevance to ordinary or device files. We also saw filenaming conventions in UNIX. Difference between the absolute and relative pathnames was highlighted next. Finally we described some of the important subdirectories contained under root (/).

UNIT 2

2. Basic File Attributes, The vi Editor 6 Hours

Text Book

2. **"UNIX – Concepts and Applications",** Sumitabha Das, 4th Edition, Tata McGraw Hill, 2006.

(Chapters 1.2, 2, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 18, 19).

Reference Books

UNIX and Shell Programming, Behrouz A. Forouzan and Richard F. Gilberg, Thomson, 2005.

Unix & Shell Programming, M.G. Venkateshmurthy, Pearson Education, 2005.

Basic File Attributes

The UNIX file system allows the user to access other files not belonging to them and without infringing on security. A file has a number of attributes (properties) that are stored in the inode. In this chapter, we discuss,

- ls –l to display file attributes (properties)
- Listing of a specific directory
- Ownership and group ownership
- Different file permissions

Listing File Attributes

ls command is used to obtain a list of all filenames in the current directory. The output in UNIX lingo is often referred to as the listing. Sometimes we combine this option with other options for displaying other attributes, or ordering the list in a different sequence. ls look up the file's inode to fetch its attributes. It lists seven attributes of all files in the current directory and they are:

- File type and Permissions
- Links
- Ownership
- Group ownership
- File size
- Last Modification date and time
- File name

The file type and its permissions are associated with each file. Links indicate the number of file names maintained by the system. This does not mean that there are so many copies of the file. File is created by the owner. Every user is attached to a group owner. File size in bytes is displayed. Last modification time is the next field. If you change only the permissions or ownership of the file, the modification time remains unchanged. In the last field, it displays the file name.

For example,

```
$ ls –l
total 72
-rw-r--r--    1 kumar  metal  19514  may  10  13:45    chap01
-rw-r--r--    1 kumar  metal   4174  may  10  15:01    chap02
-rw-rw-rw-    1 kumar  metal     84  feb  12  12:30    dept.lst
-rw-r--r--    1 kumar  metal   9156  mar  12   1999    genie.sh
drwxr-xr-x    2 kumar  metal    512  may   9  10:31    helpdir
drwxr-xr-x    2 kumar  metal    512  may   9  09:57    progs
```

Listing Directory Attributes

ls -d will not list all subdirectories in the current directory
For example,

ls –ld helpdir progs
drwxr-xr-x 2 kumar metal 512 may 9 10:31 helpdir
drwxr-xr-x 2 kumar metal 512 may 9 09:57 progs

Directories are easily identified in the listing by the first character of the first column, which here shows a d. The significance of the attributes of a directory differs a good deal from an ordinary file. To see the attributes of a directory rather than the files contained in it, use ls –ld with the directory name. Note that simply using ls –d will not list all subdirectories in the current directory. Strange though it may seem, ls has no option to list only directories.

File Ownership

When you create a file, you become its owner. Every owner is attached to a group owner. Several users may belong to a single group, but the privileges of the group are set by the owner of the file and not by the group members. When the system administrator creates a user account, he has to assign these parameters to the user:

The user-id (UID) – both its name and numeric representation
The group-id (GID) – both its name and numeric representation

File Permissions

UNIX follows a three-tiered file protection system that determines a file's access rights. It is displayed in the following format:

Filetype owner (rwx) groupowner (rwx) others (rwx)

For Example:

-rwxr-xr-- 1 kumar metal 20500 may 10 19:21 chap02

r w x r - x r - -

owner/user group owner others

The first group has all three permissions. The file is readable, writable and executable by the owner of the file. The second group has a hyphen in the middle slot, which indicates the absence of write permission by the group owner of the file. The third group has the write and execute bits absent. This set of permissions is applicable to others.

You can set different permissions for the three categories of users – owner, group and others. It's important that you understand them because a little learning here can be a dangerous thing. Faulty file permission is a sure recipe for disaster

Changing File Permissions

A file or a directory is created with a default set of permissions, which can be determined by umask. Let us assume that the file permission for the created file is -rw-r--r--. Using **chmod** command, we can change the file permissions and allow the owner to execute his file. The command can be used in two ways:

In a relative manner by specifying the changes to the current permissions
In an absolute manner by specifying the final permissions

Relative Permissions

chmod only changes the permissions specified in the command line and leaves the other permissions unchanged. Its syntax is:

chmod category operation permission filename(s)

chmod takes an expression as its argument which contains:
user category (user, group, others)
operation to be performed (assign or remove a permission)
type of permission (read, write, execute)

Category	**operation**	**permission**
u - user	+ assign	r - read
g - group	- remove	w - write
o - others	= absolute	x - execute
a - all (ugo)		

Let us discuss some examples:
Initially,

-rw-r--r-- 1 kumar metal 1906 sep 23:38 xstart

chmod u+x xstart

-rwxr--r-- 1 kumar metal 1906 sep 23:38 xstart

The command assigns (+) execute (x) permission to the user (u), other permissions remain unchanged.

chmod ugo+x xstart or
chmod a+x xstart or
chmod +x xstart

-rwxr-xr-x 1 kumar metal 1906 sep 23:38 xstart

chmod accepts multiple file names in command line

chmod u+x note note1 note3

Let initially,

-rwxr-xr-x 1 kumar metal 1906 sep 23:38 xstart

chmod go-r xstart

Then, it becomes

-rwx--x--x 1 kumar metal 1906 sep 23:38 xstart

Absolute Permissions

Here, we need not to know the current file permissions. We can set all nine permissions explicitly. A string of three octal digits is used as an expression. The permission can be represented by one octal digit for each category. For each category, we add octal digits. If we represent the permissions of each category by one octal digit, this is how the permission can be represented:

- Read permission – 4 (octal 100)
- Write permission – 2 (octal 010)
- Execute permission – 1 (octal 001)

Octal	Permissions	Significance
0	- - -	no permissions
1	- - x	execute only
2	- w -	write only
3	- w x	write and execute
4	r - -	read only
5	r - x	read and execute
6	r w -	read and write
7	r w x	read, write and execute

We have three categories and three permissions for each category, so three octal digits can describe a file's permissions completely. The most significant digit represents user and the least one represents others. chmod can use this three-digit string as the expression.

Using relative permission, we have,

chmod a+rw xstart

Using absolute permission, we have,

chmod 666 xstart

chmod 644 xstart

chmod 761 xstart

will assign all permissions to the owner, read and write permissions for the group and only execute permission to the others.

777 signify all permissions for all categories, but still we can prevent a file from being deleted. 000 signifies absence of all permissions for all categories, but still we can delete a file. It is the directory permissions that determine whether a file can be deleted or not. Only owner can change the file permissions. User can not change other user's file's permissions. But the system administrator can do anything.

The Security Implications

Let the default permission for the file xstart is

-rw-r--r--

chmod u-rw, go-r xstart or

chmod 000 xstart

This is simply useless but still the user can delete this file
On the other hand,

chmod a+rwx xstart

chmod 777 xstart

-rwxrwxrwx

The UNIX system by default, never allows this situation as you can never have a secure system. Hence, directory permissions also play a very vital role here

We can use chmod Recursively.

chmod -R a+x shell_scripts

This makes all the files and subdirectories found in the shell_scripts directory, executable by all users. When you know the shell meta characters well, you will appreciate that the * doesn't match filenames beginning with a dot. The dot is generally a safer but note that both commands change the permissions of directories also.

Directory Permissions

It is possible that a file cannot be accessed even though it has read permission, and can be removed even when it is write protected. The default permissions of a directory are,

rwxr-xr-x (755)

A directory must never be writable by group and others

Example:

mkdir c_progs

ls –ld c_progs

drwxr-xr-x 2 kumar metal 512 may 9 09:57 c_progs

If a directory has write permission for group and others also, be assured that every user can remove every file in the directory. As a rule, you must not make directories universally writable unless you have definite reasons to do so.

Changing File Ownership

Usually, on BSD and AT&T systems, there are two commands meant to change the ownership of a file or directory. Let kumar be the owner and metal be the group owner. If sharma copies a file of kumar, then sharma will become its owner and he can manipulate the attributes

chown changing file owner and **chgrp** changing group owner

On BSD, only system administrator can use chown
On other systems, only the owner can change both

chown

Changing ownership requires superuser permission, so use **su** command

ls -l note

-rwxr----x 1 kumar metal 347 may 10 20:30 note

chown sharma note; ls -l note

-rwxr----x 1 sharma metal 347 may 10 20:30 note

Once ownership of the file has been given away to sharma, the user file permissions that previously applied to Kumar now apply to sharma. Thus, Kumar can no longer edit *note* since there is no write privilege for group and others. He can not get back the ownership either. But he can copy the file to his own directory, in which case he becomes the owner of the copy.

chgrp

This command changes the file's group owner. No superuser permission is required.

ls –l dept.lst

-rw-r--r-- 1 kumar metal 139 jun 8 16:43 dept.lst

chgrp dba dept.lst; ls –l dept.lst

-rw-r--r-- 1 kumar dba 139 jun 8 16:43 dept.lst

In this chapter we considered two important file attributes – permissions and ownership. After we complete the first round of discussions related to files, we will take up the other file attributes.

- Source: Sumitabha Das, "UNIX – Concepts and Applications", 4th edition, Tata McGraw Hill, 2006

UNIT 3

3. The Shell, The Process, Customizing the environment 7 Hours

Text Book

3. **"UNIX – Concepts and Applications",** Sumitabha Das, 4th Edition, Tata McGraw Hill, 2006.

(Chapters 1.2, 2, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 18, 19).

Reference Books

UNIX and Shell Programming, Behrouz A. Forouzan and Richard F. Gilberg, Thomson, 2005.

Unix & Shell Programming, M.G. Venkateshmurthy, Pearson Education, 2005.

The Shell

Introduction

In this chapter we will look at one of the major component of UNIX architecture – The Shell. Shell acts as both a command interpreter as well as a programming facility. We will look at the interpretive nature of the shell in this chapter.

Objectives

- The Shell and its interpretive cycle
- Pattern Matching – The wild-cards
- Escaping and Quoting
- Redirection – The three standard files
- Filters – Using both standard input and standard output
- /dev/null and /dev/tty – The two special files
- Pipes
- tee – Creating a tee
- Command Substitution
- Shell Variables

1. The shell and its interpretive cycle

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 Bourne shell, ***sh***. Every Unix platform will either have the Bourne shell, or a Bourne compatible shell available.

Numerous other shells are available. Some of the more well known of these may be on your Unix system: the Korn shell, ***ksh***, by David Korn, C shell, ***csh***, by Bill Joy and the Bourne 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***.

Even though the shell appears not to be doing anything meaningful when there is no activity at the terminal, it swings into action the moment you key in something.

The following activities are typically performed by the shell in its interpretive cycle:

- The shell issues the prompt and waits for you to enter a command.
- After a command is entered, the shell scans the command line for metacharacters and expands abbreviations (like the * in rm *) to recreate a simplified command line.
- It then passes on the command line to the kernel for execution.
- The shell waits for the command to complete and normally can't do any work while the command is running.

- After the command execution is complete, the prompt reappears and the shell returns to its waiting role to start the next cycle. You are free to enter another command.

2. Pattern Matching – The Wild-Cards

A pattern is framed using ordinary characters and a metacharacter (like *) using well-defined rules. The pattern can then be used as an argument to the command, and the shell will expand it suitably before the command is executed.

The metacharacters that are used to construct the generalized pattern for matching filenames belong to a category called wild-cards. The following table lists them:

Wild-Card	Matches
*	Any number of characters including none
?	A single character
[ijk]	A single character – either an i, j or k
[x-z]	A single character that is within the ASCII range of characters x and x
[!ijk]	A single character that is not an i,j or k (Not in C shell)
[!x-z]	A single character that is not within the ASCII range of the characters x and x (Not in C Shell)
{pat1,pat2…}	Pat1, pat2, etc. (Not in Bourne shell)

Examples:

To list all files that begin with *chap*, use

```
$ ls chap*
```

To list all files whose filenames are six character long and start with chap, use

```
$ ls chap??
```

Note: Both * and ? operate with some restrictions. for example, the * doesn't match all files beginning with a . (dot) ot the / of a pathname. If you wish to list all hidden filenames in your directory having at least three characters after the dot, the dot must be matched explicitly.

```
$ ls .???*
```

However, if the filename contains a dot anywhere but at the beginning, it need not be matched explicitly.

Similarly, these characters don't match the / in a pathname. So, you cannot use

```
$ cd /usr?local        to change to /usr/local.
```

The character class

You can frame more restrictive patterns with the character class. The character class comprises a set of characters enclosed by the rectangular brackets, [and], but it matches a single character in the class. The pattern [abd] is character class, and it matches a single character – an a,b or d.

Examples:

```
$ls chap0[124]        Matches chap01, chap02, chap04 and lists if found.
$ ls chap[x-z]        Matches chapx, chapy, chapz and lists if found.
```

You can negate a character class to reverse a matching criteria. For example,

- To match all filenames with a single-character extension but not the .c ot .o files, use *.[!co]
- To match all filenames that don't begin with an alphabetic character, use [!a-zA-Z]*

Matching totally dissimilar patterns

This feature is not available in the Bourne shell. To copy all the C and Java source programs from another directory, we can delimit the patterns with a comma and then put curly braces around them.

```
$ cp $HOME/prog_sources/*.{c,java} .
```

The Bourne shell requires two separate invocations of cp to do this job.

```
$ cp /home/srm/{project,html,scripts/* .
```

The above command copies all files from three directories (project, html and scripts) to the current directory.

3. Escaping and Quoting

Escaping is providing a \ (backslash) before the wild-card to remove (escape) its special meaning.

For instance, if we have a file whose filename is chap* (Remember a file in UNIX can be names with virtually any character except the / and null), to remove the file, it is dangerous to give command as *rm chap**, as it will remove all files beginning with chap. Hence to suppress the special meaning of *, use the command *rm chap**

To list the contents of the file chap0[1-3], use

```
$ cat chap0\[1-3\]
```

A filename can contain a whitespace character also. Hence to remove a file named My Documend.doc, which has a space embedded, a similar reasoning should be followed:

```
$ rm My\ Document.doc
```

Quoting is enclosing the wild-card, or even the entire pattern, within quotes. Anything within these quotes (barring a few exceptions) are left alone by the shell and not interpreted.

When a command argument is enclosed in quotes, the meanings of all enclosed special characters are turned off.

Examples:

$ rm 'chap*'	Removes fil *chap**
$ rm "My Document.doc"	Removes file *My Document.doc*

4. Redirection : The three standard files

The shell associates three files with the terminal – two for display and one for the keyboard. These files are streams of characters which many commands see as input and output. When a user logs in, the shell makes available three files representing three streams. Each stream is associated with a default device:

Standard input: The file (stream) representing input, connected to the keyboard.

Standard output: The file (stream) representing output, connected to the display.

Standard error: The file (stream) representing error messages that emanate from the command or shell, connected to the display.

The standard input can represent three input sources:
The keyboard, the default source.
A file using redirection with the < symbol.
Another program using a pipeline.

The standard output can represent three possible destinations:
The terminal, the default destination.
A file using the redirection symbols > and >>.
As input to another program using a pipeline.

A file is opened by referring to its pathname, but subsequent read and write operations identify the file by a unique number called a file descriptor. The kernel maintains a table of file descriptors for every process running in the system. The first three slots are generally allocated to the three standard streams as,

0 – Standard input
1 – Standard output
2 – Standard error

These descriptors are implicitly prefixed to the redirection symbols.
Examples:
Assuming file2 doesn't exist, the following command redirects the standard output to file *myOutput* and the standard error to file *myError*.

```
$ ls –l file1 file2 1>myOutput 2>myError
```

To redirect both standard output and standard error to a single file use:

```
$ ls –l file1 file2 1>| myOutput 2>| myError  OR
$ ls –l file1 file2 1> myOutput 2>& 1
```

5. Filters: Using both standard input and standard output

UNIX commands can be grouped into four categories viz.,

1. Directory-oriented commands like mkdir, rmdir and cd, and basic file handling commands like cp, mv and rm use neither standard input nor standard output.
2. Commands like ls, pwd, who etc. don't read standard input but they write to standard output.
3. Commands like lp that read standard input but don't write to standard output.
4. Commands like cat, wc, cmp etc. that use both standard input and standard output.

Commands in the fourth category are called filters. Note that filters can also read directly from files whose names are provided as arguments.
Example: To perform arithmetic calculations that are specified as expressions in input file calc.txt and redirect the output to a file result.txt, use

```
$ bc < calc.txt   > result.txt
```

6. /dev/null and /dev/tty : Two special files

/dev/null: If you would like to execute a command but don't like to see its contents on the screen, you may wish to redirect the output to a file called /dev/null. It is a special file that can accept any stream without growing in size. It's size is always zero.

/dev/tty: This file indicates one's terminal. In a shell script, if you wish to redirect the output of some select statements explicitly to the terminal. In such cases you can redirect these explicitly to /dev/tty inside the script.

7. Pipes

With piping, the output of a command can be used as input (piped) to a subsequent command.

```
$ command1 | command2
```

Output from command1 is piped into input for command2.

This is equivalent to, but more efficient than:

```
$ command1 > temp
$ command2 < temp
$ rm temp
```

Examples

```
$ ls -al | more
$ who | sort | lpr
```

When a command needs to be ignorant of its source

If we wish to find total size of all C programs contained in the working directory, we can use the command,

```
$ wc –c *.c
```

However, it also shows the usage for each file(size of each file). We are not interested in individual statistics, but a single figure representing the total size. To be able to do that, we must make wc ignorant of its input source. We can do that by feeding the concatenated output stream of all the .c files to wc –c as its input:

```
$ cat *.c | wc –c
```

8. Creating a tee

tee is an external command that handles a character stream by duplicating its input. It saves one copy in a file and writes the other to standard output. It is also a filter and hence can be placed anywhere in a pipeline.

Example: The following command sequence uses tee to display the output of who and saves this output in a file as well.

```
$ who | tee users.lst
```

9. Command substitution

The shell enables the connecting of two commands in yet another way. While a pipe enables a command to obtain its standard input from the standard output of another command, the shell enables one or more command arguments to be obtained from the standard output of another command. This feature is called command substitution.
Example:

```
$ echo Current date and time is `date`
```

Observe the use of backquotes around date in the above command. Here the output of the command execution of date is taken as argument of echo. The shell executes the enclosed command and replaces the enclosed command line with the output of the command.

Similarly the following command displays the total number of files in the working directory.

```
$ echo "There are 'ls | wc –l' files in the current directory"
```

Observe the use of double quotes around the argument of echo. If you use single quotes, the backquote is not interpreted by the shell if enclosed in single quotes.

10. Shell 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.

To declare a local shell variable we use the form *variable=value* (no spaces around =) and its evaluation requires the $ as a prefix to the variable.
Example:

```
$  count=5
$  echo $count
5
```

A variable can be removed with **unset** and protected from reassignment by **readonly**. Both are shell internal commands.

Note: In C shell, we use **set** statement to set variables. Here, there either has to be whitespace on both sides of the = or none at all.

```
$ set count=5
$ set size = 10
```

Uses of local shell variables

1. Setting pathnames: If a pathname is used several times in a script, we can assign it to a variable and use it as an argument to any command.
2. Using command substitution: We can assign the result of execution of a command to a variable. The command to be executed must be enclosed in backquotes.
3. Concatenating variables and strings: Two variables can be concatenated to form a new variable.
   Example: $ base=foo ; ext=.c

```
$ file=$base$ext
$ echo $file        // prints foo.c
```

Conclusion

In this chapter we saw the major interpretive features of the shell. The following is a summary of activities that the shell performs when a command line is encountered at the prompt.

- Parsing: The shell first breaks up the command line into words using spaces and tabs as delimiters, unless quoted. All consecutive occurrences of a space or tab are replaced with a single space.
- Variable evaluation: All $-prefixed strings are evaluated as variables, unless quoted or escaped.
- Command substitution: Any command surrounded by backquotes is executed by the shell, which then replaces the standard output of the command into the command line.
- Redirection: The shell then looks for the characters >, < and >> to open the files they point to.
- Wild-card interpretation: The shell then scans the command line for wild-cards (the characters *, ?, [and]). Any word containing a wild-card is replaced by a sorted list of filenames that match the pattern. The list of these filenames then forms the arguments to the command.
- PATH evaluation: It finally looks for the PATH variable to determine the sequence of directories it has to search in order to find the associated binary.

The Process

UNIT 4

4. More file attributes, Simple filters **7 Hours**

Text Book

4. **"UNIX – Concepts and Applications",** Sumitabha Das, 4th Edition, Tata McGraw Hill, 2006.

(Chapters 1.2, 2, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 18, 19).

Reference Books

UNIX and Shell Programming, Behrouz A. Forouzan and Richard F. Gilberg, Thomson, 2005.

Unix & Shell Programming, M.G. Venkateshmurthy, Pearson Education, 2005.

MORE FILE ATTRIBUTES

Apart from permissions and ownership, a UNIX file has several other attributes, and in this chapter, we look at most of the remaining ones. A file also has properties related to its time stamps and links. It is important to know how these attributes are interpreted when applied to a directory or a device.

This chapter also introduces the concepts of file system. It also looks at the inode, the lookup table that contained almost all file attributes. Though a detailed treatment of the file systems is taken up later, knowledge of its basics is essential to our understanding of the significance of some of the file attributes. Basic file attributes has helped us to know about - ls –l to display file attributes (properties), listing of a specific directory, ownership and group ownership and different file permissions. ls –l provides attributes like – permissions, links, owner, group owner, size, date and the file name.

File Systems and inodes

The hard disk is split into distinct partitions, with a separate file system in each partition. Every file system has a directory structure headed by root.

n partitions = n file systems = n separate root directories

All attributes of a file except its name and contents are available in a table – inode (index node), accessed by the inode number. The inode contains the following attributes of a file:

- File type
- File permissions
- Number of links
- The UID of the owner
- The GID of the group owner
- File size in bytes
- Date and time of last modification
- Date and time of last access
- Date and time of last change of the inode
- An array of pointers that keep track of all disk blocks used by the file

Please note that, neither the name of the file nor the inode number is stored in the inode. To know inode number of a file:

```
ls -il tulec05
```

```
9059  -rw-r--r-- 1 kumar  metal  51813 Jan 31 11:15  tulec05
```

Where, 9059 is the inode number and no other file can have the same inode number in the same file system.

Hard Links

The link count is displayed in the second column of the listing. This count is normally 1, but the following files have two links,

```
-rwxr-xr-- 2 kumar metal 163 Jull 13 21:36 backup.sh
-rwxr-xr-- 2 kumar metal 163 Jul 13 21:36 restore.sh
```

All attributes seem to be identical, but the files could still be copies. It's the link count that seems to suggest that the files are linked to each other. But this can only be confirmed by using the –i option to ls.

```
ls -li backup.sh restore.sh
```

```
478274 -rwxr-xr-- 2 kumar metal163 jul 13 21:36 backup.sh
478274 -rwxr-xr-- 2 kumar metal163 jul 13 21:36 restore.sh
```

ln: Creating Hard Links

A file is linked with the ln command which takes two filenames as arguments (cp command). The command can create both a hard link and a soft link and has syntax similar to the one used by cp. The following command links emp.lst with employee:

```
ln emp.lst employee
```

The –i option to ls shows that they have the same inode number, meaning that they are actually one end the same file:

```
ls -li emp.lst employee
```

```
29518 -rwxr-xr-x 2 kumar metal 915 may 4 09:58 emp.lst
29518 -rwxr-xr-x 2 kumar metal 915 may 4 09:58 employee
```

The link count, which is normally one for unlinked files, is shown to be two. You can increase the number of links by adding the third file name emp.dat as:

```
ln employee emp.dat ; ls -l emp*
```

```
29518 -rwxr-xr-x 3 kumar metal 915 may 4 09:58 emp.dat
29518 -rwxr-xr-x 3 kumar metal 915 may 4 09:58 emp.lst
29518 -rwxr-xr-x 3 kumar metal 915 may 4 09:58 employee
```

You can link multiple files, but then the destination filename must be a directory. A file is considered to be completely removed from the file system when its link count drops to zero. ln returns an error when the destination file exists. Use the –f option to force the removal of the existing link before creation of the new one

Where to use Hard Links

ln data/ foo.txt input_files

It creates link in directory *input_files.* With this link available, your existing programs will continue to find foo.txt in the *input_files* directory. It is more convenient to do this that modifies all programs to point to the new path. Links provide some protection against accidental deletion, especially when they exist in different directories. Because of links, we don't need to maintain two programs as two separate disk files if there is very little difference between them. A file's name is available to a C program and to a shell script. A single file with two links can have its program logic make it behave in two different ways depending on the name by which it is called.

We can't have two linked filenames in two file systems and we can't link a directory even within the same file system. This can be solved by using symbolic links (soft links).

Symbolic Links

Unlike the hard linked, a symbolic link doesn't have the file's contents, but simply provides the pathname of the file that actually has the contents.

ln -s note note.sym

ls -li note note.sym

9948 -rw-r--r-- 1 kumar group 80 feb 16 14:52 note
9952 lrwxrwxrwx 1 kumar group 4 feb16 15:07note.sym ->note

Where, l indicate symbolic link file category. -> indicates note.sym contains the pathname for the filename note. Size of symbolic link is only 4 bytes; it is the length of the pathname of note.

It's important that this time we indeed have two files, and they are not identical. Removing note.sym won't affect us much because we can easily recreate the link. But if we remove note, we would lose the file containing the data. In that case, note.sym would point to a nonexistent file and become a dangling symbolic link.

Symbolic links can also be used with relative pathnames. Unlike hard links, they can also span multiple file systems and also link directories. If you have to link all filenames in a directory to another directory, it makes sense to simply link the directories. Like other files, a symbolic link has a separate directory entry with its own inode number. This means that rm can remove a symbolic link even if its points to a directory.

A symbolic link has an inode number separate from the file that it points to. In most cases, the pathname is stored in the symbolic link and occupies space on disk.

However, Linux uses a fast symbolic link which stores the pathname in the inode itself provided it doesn't exceed 60 characters.

The Directory

A directory has its own permissions, owners and links. The significance of the file attributes change a great deal when applied to a directory. For example, the size of a directory is in no way related to the size of files that exists in the directory, but rather to the number of files housed by it. The higher the number of files, the larger the directory size. Permission acquires a different meaning when the term is applied to a directory.

```
ls -l -d progs
```

```
drwxr-xr-x  2  kumar  metal 320 may 9 09:57 progs
```

The default permissions are different from those of ordinary files. The user has all permissions, and group and others have read and execute permissions only. The permissions of a directory also impact the security of its files. To understand how that can happen, we must know what permissions for a directory really mean.

Read permission

Read permission for a directory means that the list of filenames stored in that directory is accessible. Since ls reads the directory to display filenames, if a directory's read permission is removed, ls wont work. Consider removing the read permission first from the directory *progs*,

```
ls -ld progs
```

```
drwxr-xr-x  2  kumar  metal 128 jun 18 22:41 progs
```

```
chmod -r progs ; ls progs
```

```
progs: permission denied
```

Write permission

We can't write to a directory file. Only the kernel can do that. If that were possible, any user could destroy the integrity of the file system. Write permission for a directory implies that you are permitted to create or remove files in it. To try that out, restore the read permission and remove the write permission from the directory before you try to copy a file to it.

```
chmod 555 progs ; ls –ld progs
```

```
dr-xr-xr-x 2 kumar metal 128 jun 18 22:41 progs
```

```
cp emp.lst progs
```

cp: cannot create progs/emp.lst: permission denied

- The write permission for a directory determines whether we can create or remove files in it because these actions modify the directory
- Whether we can modify a file depends on whether the file itself has write permission. Changing a file doesn't modify its directory entry

Execute permission

If a single directory in the pathname doesn't have execute permission, then it can't be searched for the name of the next directory. That's why the execute privilege of a directory is often referred to as the search permission. A directory has to be searched for the next directory, so the cd command won't work if the search permission for the directory is turned off.

```
chmod 666 progs ; ls –ld progs
```

drw-rw-rw- 2 kumar metal 128 jun 18 22:41 progs

```
cd progs
```

permission denied to search and execute it

umask: DEFAULT FILE AND DIRECTORY PERMISSIONS

When we create files and directories, the permissions assigned to them depend on the system's default setting. The UNIX system has the following default permissions for all files and directories.

rw-rw-rw- (octal 666) for regular files

rwxrwxrwx (octal 777) for directories

The default is transformed by subtracting the user mask from it to remove one or more permissions. We can evaluate the current value of the mask by using *umask* without arguments,

```
$ umask
022
```

This becomes 644 (666-022) for ordinary files and 755 (777-022) for directories *umask* 000. This indicates, we are not subtracting anything and the default permissions will remain unchanged. Note that, changing system wide default permission settings is possible using *chmod* but not by *umask*

MODIFICATION AND ACCESS TIMES

A UNIX file has three time stamps associated with it. Among them, two are:

- Time of last file modification ls -l
- Time of last access ls –lu

The access time is displayed when ls -l is combined with the -u option. Knowledge of file‘s modification and access times is extremely important for the system administrator. Many of the tools used by them look at these time stamps to decide whether a particular file will participate in a backup or not.

TOUCH COMMAND – changing the time stamps

To set the modification and access times to predefined values, we have,

touch options expression filename(s)

touch emp.lst (without options and expression)

Then, both times are set to the current time and creates the file, if it doesn’t exist.

touch command (without options but with expression) can be used. The expression consists of MMDDhhmm (month, day, hour and minute).

touch 03161430 emp.lst ; ls -l emp.lst

-rw-r--r-- 1 kumar metal 870 mar 16 14:30 emp.lst

ls -lu emp.lst

-rw-r--r-- 1 kumar metal 870 mar 16 14:30 emp.lst

It is possible to change the two times individually. The –m and –a options change the modification and access times, respectively:

touch command (with options and expression)

-m for changing modification time
-a for changing access time

touch -m 02281030 emp.lst ; ls -l emp.lst

-rw-r--r-- 1 kumar metal 870 feb 28 10:30 emp.lst

touch -a 01261650 emp.lst ; ls -lu emp.lst

-rw-r--r-- 1 kumar metal 870 jan 26 16:50 emp.lst

find : locating files

It recursively examines a directory tree to look for files matching some criteria, and then takes some action on the selected files. It has a difficult command line, and if you have ever wondered why UNIX is hated by many, then you should look up the cryptic *find* documentation. How ever, *find* is easily tamed if you break up its arguments into three components:

find path_list selecton_criteria action

where,

- Recursively examines all files specified in path_list
- It then matches each file for one or more selection-criteria
- It takes some action on those selected files

The path_list comprises one or more subdirectories separated by white space. There can also be a host of selection_criteria that you use to match a file, and multiple actions to dispose of the file. This makes the command difficult to use initially, but it is a program that every user must master since it lets him make file selection under practically any condition.

- Source: Sumitabha Das, "UNIX – Concepts and Applications", 4th edition, Tata McGraw Hill, 2006

SIMPLE FILTERS

Filters are the commands which accept data from standard input manipulate it and write the results to standard output. Filters are the central tools of the UNIX tool kit, and each filter performs a simple function. Some commands use delimiter, pipe (|) or colon (:). Many filters work well with delimited fields, and some simply won't work without them. The piping mechanism allows the standard output of one filter serve as standard input of another. The filters can read data from standard input when used without a filename as argument, and from the file otherwise

The Simple Database

Several UNIX commands are provided for text editing and shell programming. (emp.lst) - each line of this file has six fields separated by five delimiters. The details of an employee are stored in one single line. This text file designed in fixed format and containing a personnel database. There are 15 lines, where each field is separated by the delimiter |.

$ cat emp.lst

2233 | a.k.shukla | g.m | sales | 12/12/52 | 6000
9876 | jai sharma | director | production | 12/03/50 | 7000
5678 | sumit chakrobarty | d.g.m. | marketing | 19/04/43 | 6000
2365 | barun sengupta | director | personnel | 11/05/47 | 7800
5423 | n.k.gupta | chairman | admin | 30/08/56 | 5400
1006 | chanchal singhvi | director | sales | 03/09/38 | 6700
6213 | karuna ganguly | g.m. | accounts | 05/06/62 | 6300
1265 | s.n. dasgupta | manager | sales | 12/09/63 | 5600
4290 | jayant choudhury | executive | production | 07/09/50 | 6000
2476 | anil aggarwal | manager | sales | 01/05/59 | 5000
6521 | lalit chowdury | directir | marketing | 26/09/45 | 8200
3212 | shyam saksena | d.g.m. | accounts | 12/12/55 | 6000
3564 | sudhir agarwal | executive | personnel | 06/07/47 | 7500
2345 | j. b. sexena | g.m. | marketing | 12/03/45 | 8000
0110 | v.k.agrawal | g.m.| marketing | 31/12/40 | 9000

pr : paginating files

We know that,

cat dept.lst

01|accounts|6213
02|progs|5423
03|marketing|6521

04|personnel|2365
05|production|9876
06|sales|1006

pr command adds suitable headers, footers and formatted text. pr adds five lines of margin at the top and bottom. The header shows the date and time of last modification of the file along with the filename and page number.

pr dept.lst

May 06 10:38 1997 dept.lst page 1

01:accounts:6213
02:progs:5423
03:marketing:6521
04:personnel:2365
05:production:9876
06:sales:1006

…*blank lines.*…

pr options

The different options for pr command are:

-k prints k (integer) columns
-t to suppress the header and footer
-h to have a header of user's choice
-d double spaces input
-n will number each line and helps in debugging
-on offsets the lines by n spaces and increases left margin of page

pr +10 chap01

starts printing from page 10

pr -l 54 chap01

this option sets the page length to 54

head – displaying the beginning of the file

The command displays the top of the file. It displays the first 10 lines of the file, when used without an option.

head emp.lst

-n to specify a line count
head -n 3 emp.lst

will display the first three lines of the file.

tail – displaying the end of a file

This command displays the end of the file. It displays the last 10 lines of the file, when used without an option.

tail emp.lst

-n to specify a line count

tail -n 3 emp.lst

displays the last three lines of the file. We can also address lines from the beginning of the file instead of the end. The +count option allows to do that, where count represents the line number from where the selection should begin.

tail +11 emp.lst
Will display 11th line onwards

Different options for tail are:

- Monitoring the file growth (-f)
- Extracting bytes rather than lines (-c)

Use tail –f when we are running a program that continuously writes to a file, and we want to see how the file is growing. We have to terminate this command with the interrupt key.

cut – slitting a file vertically

It is used for slitting the file vertically. head -n 5 emp.lst | tee shortlist will select the first five lines of emp.lst and saves it to *shortlist.* We can cut by using -c option with a list of column numbers, delimited by a comma (cutting columns).

cut -c 6-22,24-32 shortlist

cut -c -3,6-22,28-34,55- shortlist

The expression 55- indicates column number 55 to end of line. Similarly, -3 is the same as 1-3.

Most files don't contain fixed length lines, so we have to cut fields rather than columns (cutting fields).

-d for the field delimiter
-f for the field list

cut -d \| -f 2,3 shortlist | tee cutlist1

will display the second and third columns of *shortlist* and saves the output in *cutlist1*. here | is escaped to prevent it as pipeline character

- To print the remaining fields, we have

cut –d \| -f 1,4- shortlist > cutlist2

paste – pasting files

When we cut with *cut*, it can be pasted back with the *paste* command, *vertically* rather than horizontally. We can view two files side by side by pasting them. In the previous topic, cut was used to create the two files cutlist1 and cutlist2 containing two cut-out portions of the same file.

paste cutlist1 cutlist2

We can specify one or more delimiters with -d

paste -d "|" cutlist1 cutlist2

Where each field will be separated by the delimiter |. Even though paste uses at least two files for concatenating lines, the data for one file can be supplied through the standard input.

Joining lines (-s)

Let us consider that the file *address book* contains the details of three persons

cat addressbook

paste -s addressbook -to print in one single line

paste -s -d "| | \n" addressbook -are used in a circular manner

sort : ordering a file

Sorting is the ordering of data in ascending or descending sequence. The sort command orders a file and by default, the entire line is sorted

sort shortlist

This default sorting sequence can be altered by using certain options. We can also sort one or more keys (fileds) or use a different ordering rule.

sort options

The important sort options are:

-tchar	uses delimiter *char* to identify fields
-k n	sorts on nth field
-k m,n	starts sort on mth field and ends sort on nth field
-k m.n	starts sort on nth column of mth field
-u	removes repeated lines
-n	sorts numerically
-r	reverses sort order
-f	folds lowercase to equivalent uppercase
-m list	merges sorted files in list
-c	checks if file is sorted
-o flname	places output in file flname

sort –t"|" –k 2 shortlist

sorts the second field (name)

sort –t"|" –r –k 2 shortlist or

sort –t"|" –k 2r shortlist

sort order can be revered with this –r option.

sort –t"|" –k 3,3 –k 2,2 shortlist

sorting on secondary key is also possible as shown above.

sort –t"|" –k 5.7,5.8 shortlist

we can also specify a character position with in a field to be the beginning of sort as shown above (sorting on columns).

sort –n numfile

when sort acts on numericals, strange things can happen. When we sort a file containing only numbers, we get a curious result. This can be overridden by –n (numeric) option.

cut –d "|" –f3 emp.lst | sort –u | tee desigx.lst

Removing repeated lines can be possible using –u option as shown above. If we cut out the designation filed from emp.lst, we can pipe it to sort to find out the unique designations that occur in the file.

Other sort options are:

sort –o sortedlist –k 3 shortlist

sort –o shortlist shortlist

sort –c shortlist

sort –t "|" –c –k 2 shortlist

sort –m foo1 foo2 foo3

uniq command – locate repeated and nonrepeated lines

When we concatenate or merge files, we will face the problem of duplicate entries creeping in. we saw how sort removes them with the –u option. UNIX offers a special tool to handle these lines – the uniq command. Consider a sorted dept.lst that includes repeated lines:

cat dept.lst

displays all lines with duplicates. Where as,

uniq dept.lst

simply fetches one copy of each line and writes it to the standard output. Since uniq requires a sorted file as input, the general procedure is to sort a file and pipe its output to uniq. The following pipeline also produces the same output, except that the output is saved in a file:

sort dept.lst | uniq – uniqlist

Different uniq options are :

Selecting the nonrepeated lines (-u)

cut –d "|" –f3 emp.lst | sort | uniq –u

Selecting the duplicate lines (-d)

```
cut –d "|" –f3 emp.lst | sort | uniq –d
```

Counting frequency of occurrence (-c)

```
cut –d "|" –f3 emp.lst | sort | uniq –c
```

tr command – translating characters

The tr filter manipulates the individual characters in a line. It translates characters using one or two compact expressions.

tr options expn1 expn2 standard input

It takes input only from standard input, it doesn't take a filename as argument. By default, it translates each character in expression1 to its mapped counterpart in expression2. The first character in the first expression is replaced with the first character in the second expression, and similarly for the other characters.

```
tr '|/' '~-' < emp.lst | head –n 3

exp1='|/' ; exp2='~-'

tr "$exp1" "$exp2" < emp.lst
```

Changing case of text is possible from lower to upper for first three lines of the file.

```
head –n 3 emp.lst | tr '[a-z]' '[A-Z]'
```

Different **tr options** are:
Deleting charecters (-d)

```
tr –d '|/' < emp.lst | head –n 3
```

Compressing multiple consecutive charecters (-s)

```
tr –s ' ' < emp.lst | head –n 3
```

Complementing values of expression (-c)

```
tr –cd '|/' < emp.lst
```

Using ASCII octal values and escape sequences

```
tr '|' '\012' < emp.lst | head –n 6
```

UNIT 5

5. Filters using regular expressions, 6 Hours

Text Book

5. **"UNIX – Concepts and Applications",** Sumitabha Das, 4th Edition, Tata McGraw Hill, 2006.

(Chapters 1.2, 2, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 18, 19).

Reference Books

UNIX and Shell Programming, Behrouz A. Forouzan and Richard F. Gilberg, Thomson, 2005.

Unix & Shell Programming, M.G. Venkateshmurthy, Pearson Education, 2005.

Filters Using Regular Expression
grep and sed

We often need to search a file for a pattern, either to see the lines containing (or not containing) it or to have it replaced with something else. This chapter discusses two important filters that are specially suited for these tasks – grep and sed. grep takes care of all search requirements we may have. sed goes further and can even manipulate the individual characters in a line. In fact sed can de several things, some of then quite well.

grep – searching for a pattern

It scans the file / input for a pattern and displays lines containing the pattern, the line numbers or filenames where the pattern occurs. It's a command from a special family in UNIX for handling search requirements.

grep *options pattern filename(s)*

grep "sales" emp.lst

will display lines containing sales from the file emp.lst. Patterns with and without quotes is possible. It's generally safe to quote the pattern. Quote is mandatory when pattern involves more than one word. It returns the prompt in case the pattern can't be located.

grep president emp.lst

When grep is used with multiple filenames, it displays the filenames along with the output.

grep "director" emp1.lst emp2.lst

Where it shows filename followed by the contents

grep options

grep is one of the most important UNIX commands, and we must know the options that POSIX requires grep to support. Linux supports all of these options.

-i	ignores case for matching
-v	doesn't display lines matching expression
-n	displays line numbers along with lines
-c	displays count of number of occurrences
-l	displays list of filenames only
-e exp	specifies expression with this option
-x	matches pattern with entire line
-f file	takes pattrens from file, one per line
-E	treats pattren as an extended RE

-F matches multiple fixed strings

grep -i ‘agarwal’ emp.lst

grep -v ‘director’ emp.lst > otherlist

wc -l otherlist will display 11 otherlist

grep –n ‘marketing’ emp.lst

grep –c ‘director’ emp.lst

grep –c ‘director’ emp*.lst

will print filenames prefixed to the line count

grep –l ‘manager’ *.lst

will display filenames *only*

grep –e ‘Agarwal’ –e ‘aggarwal’ –e ‘agrawal’ emp.lst

will print matching multiple patterns

grep –f pattern.lst emp.lst

all the above three patterns are stored in a separate file *pattern.lst*

Basic Regular Expressions (BRE) – An Introduction

It is tedious to specify each pattern separately with the -e option. grep uses an expression of a different type to match a group of similar patterns. If an expression uses meta characters, it is termed a regular expression. Some of the characters used by regular expression are also meaningful to the shell.

BRE character subset

The basic regular expression character subset uses an elaborate meta character set, overshadowing the shell’s wild-cards, and can perform amazing matches.

*****	Zero or more occurrences
g*	nothing or g, gg, ggg, etc.
.	A single character
.*	nothing or any number of characters
[pqr]	a single character p, q or r
[c1-c2]	a single character within the ASCII range represented by c1 and c2

The character class

grep supports basic regular expressions (BRE) by default and extended regular expressions (ERE) with the –E option. A regular expression allows a group of characters enclosed within a pair of [ ], in which the match is performed for a single character in the group.

grep "[aA]g[ar][ar]wal" emp.lst

A single pattern has matched two similar strings. The pattern [a-zA-Z0-9] matches a single alphanumeric character. When we use range, make sure that the character on the left of the hyphen has a lower ASCII value than the one on the right. Negating a class (^) (caret) can be used to negate the character class. When the character class begins with this character, all characters other than the ones grouped in the class are matched.

The *

The asterisk refers to the immediately preceding character. * indicates zero or more occurrences of the previous character.

g* nothing or g, gg, ggg, etc.

grep "[aA]gg*[ar][ar]wal" emp.lst

Notice that we don't require to use –e option three times to get the same output!!!!!

The dot

A dot matches a single character. The shell uses ? Character to indicate that.

.* signifies any number of characters or none

grep "j.*saxena" emp.lst

Specifying Pattern Locations (^ and $)

Most of the regular expression characters are used for matching patterns, but there are two that can match a pattern at the beginning or end of a line. Anchoring a pattern is often necessary when it can occur in more than one place in a line, and we are interested in its occurance only at a particular location.

^ for matching at the beginning of a line
$ for matching at the end of a line

grep "^2" emp.lst

Selects lines where emp_id starting with 2

grep “7…$” emp.lst

Selects lines where emp_salary ranges between 7000 to 7999

grep “^[^2]” emp.lst

Selects lines where emp_id doesn’t start with 2

When meta characters lose their meaning

It is possible that some of these special characters actually exist as part of the text. Sometimes, we need to escape these characters. For example, when looking for a pattern g*, we have to use \
To look for [, we use \[
To look for .*, we use \.*

Extended Regular Expression (ERE) and grep

If current version of grep doesn’t support ERE, then use egrep but without the –E option. -E option treats pattern as an ERE.

+ matches one or more occurrences of the previous character

? Matches zero or one occurrence of the previous character

b+ matches b, bb, bbb, etc.

b? matches either a single instance of b or nothing

These characters restrict the scope of match as compared to the *

grep –E “[aA]gg?arwal” emp.lst

?include +<stdio.h>

The ERE set

ch+	matches one or more occurrences of character ch
ch?	Matches zero or one occurrence of character ch
exp1|exp2	matches exp1 or exp2
(x1|x2)x3	matches x1x3 or x2x3

Matching multiple patterns (|, (and))

grep –E ‘sengupta|dasgupta’ emp.lst

We can locate both without using –e option twice, or

grep –E ‘(sen|das)gupta’ emp.lst

sed – The Stream Editor

sed is a multipurpose tool which combines the work of several filters. sed uses **instructions** to act on text. An instruction combines an **address** for selecting lines, with an **action** to be taken on them.

sed options ‘address action’ file(s)

sed supports only the BRE set. *Address* specifies either one line number to select a single line or a set of two lines, to select a group of contiguous lines. *action* specifies print, insert, delete, substitute the text.

sed processes several instructions in a sequential manner. Each instruction operates on the output of the previous instruction. In this context, two options are relevant, and probably they are the only ones we will use with sed – the –e option that lets us use multiple instructions, and the –f option to take instructions from a file. Both options are used by grep in identical manner.

Line Addressing

sed ‘3q’ emp.lst

Just similar to *head –n 3 emp.lst*. Selects first three lines and quits

sed –n ‘1,2p’ emp.lst

p prints selected lines as well as all lines. To suppress this behavior, we use –n whenever we use p command

sed –n ‘$p’ emp.lst

Selects last line of the file

sed –n ‘9,11p’ emp.lst

Selecting lines from anywhere of the file, between lines from 9 to 11

sed –n ‘1,2p
7,9p

$p' emp.lst

Selecting multiple groups of lines

sed –n '3,$!p' emp.lst

Negating the action, just same as 1,2p

Using Multiple Instructions (-e and –f)

There is adequate scope of using the –e and –f options whenever sed is used with multiple instructions.

sed –n –e '1,2p' –e '7,9p' –e '$p' emp.lst

Let us consider,

cat instr.fil
1,2p
7,9p
$p

-f option to direct the sed to take its instructions from the file

sed –n –f instr.fil emp.lst

We can combine and use –e and –f options as many times as we want

sed –n –f instr.fil1 –f instr.fil2 emp.lst

sed –n –e '/saxena/p' –f instr.fil1 –f instr.fil2 emp.lst

Context Addressing

We can specify one or more patterns to locate lines

sed –n '/director/p' emp.lst

We can also specify a comma-separated pair of context addresses to select a group of lines.

sed –n '/dasgupta/,/saxena/p' emp.lst

Line and context addresses can also be mixed

sed –n '1,/dasgupta/p' emp.lst

Using regular expressions
Context addresses also uses regular expressions.

```
Sed –n '/[aA]gg*[ar][ar]wal/p' emp.lst
```

Selects all agarwals.

```
Sed –n '/sa[kx]s*ena/p
        /gupta/p' emp.lst
```

Selects saxenas and gupta.

We can also use ^ and $, as part of the regular expression syntax.

```
sed –n '/50…..$/p' emp.lst
```

Selects all people born in the year 1950.

Writing Selected Lines to a File (w)

We can use w command to write the selected lines to a separate file.

```
sed –n '/director/w dlist' emp.lst
```

Saves the lines of directors in *dlist* file

```
sed –n '/director/w dlist
        /manager/w mlist
        /executive/w elist' emp.lst
```

Splits the file among three files

```
sed –n '1,500w foo1
501,$w foo2' foo.main
```

Line addressing also is possible. Saves first 500 lines in foo1 and the rest in foo2

Text Editing

sed supports inserting (i), appending (a), changing (c) and deleting (d) commands for the text.

```
$ sed '1i\
> #include <stdio.h>\
> #include <unistd.h>
> 'foo.c > $$
```

Will add two include lines in the beginning of foo.c file. Sed identifies the line without the \ as the last line of input. Redirected to $$ temporary file. This technique has to be followed when using the a and c commands also. To insert a blank line *after* each line of the file is printed (*double spacing text*), we have,

```
sed 'a\

' emp.lst
```

Deleting lines (d)

```
sed '/director/d' emp.lst > olist          or

sed –n '/director/!p' emp.lst > olist
```

Selects all lines except those containing *director*, and saves them in *olist*

Note that –n option not to be used with d

Substitution (s)

Substitution is the most important feature of sed, and this is one job that sed does exceedingly well.

```
[address]s/expression1/expression2/flags
```

Just similar to the syntax of substitution in vi editor, we use it in sed also.

```
sed 's/|/:/' emp.lst | head –n 2
```

2233:a.k.shukla |gm |sales |12/12/52|6000

9876:jai sharma |director|production|12/03/50|7000

Only the first instance of | in a line has been replaced. We need to use the g (global) flag to replace all the pipes.

```
sed 's/|/:/g' emp.lst | head –n 2
```

We can limit the vertical boundaries too by specifying an address (for first three lines only).

```
sed '1,3s/|/:/g' emp.lst
```

Replace the word director with member in the first five lines of emp.lst

```
sed '1,5s/director/member/' emp.lst
```

sed also uses regular expressions for patterns to be substituted. To replace all occurrence of agarwal, aggarwal and agrawal with simply Agarwal, we have,

```
sed 's/[Aa]gg*[ar][ar]wal/Agarwal/g' emp.lst
```

We can also use ^ and $ with the same meaning. To add 2 prefix to all emp-ids,

```
sed 's/^/2/' emp.lst | head –n 1
```

22233 | a.k.shukla | gm | sales | 12/12/52 | 6000

To add .00 suffix to all salary,

```
sed 's/$/.00/' emp.lst | head –n 1
```

2233 | a.k.shukla | gm | sales | 12/12/52 | 6000.00

Performing multiple substitutions

```
sed 's/<I>/<EM>/g
s/<B>/<STRONG>/g
s/<U>/<EM>/g' form.html
```

An instruction processes the output of the previous instruction, as sed is a stream editor and works on data stream

```
sed 's/<I>/<EM>/g
s/<EM>/<STRONG>/g' form.html
```

When a 'g' is used at the end of a substitution instruction, the change is performed globally along the line. Without it, only the left most occurrence is replaced. When there are a group of instructions to execute, you should place these instructions in a file instead and use sed with the –f option.

Compressing multiple spaces

```
sed 's/*|/|/g' emp.lst  |  tee empn.lst  |  head –n 3
```

2233|a.k.shukla|g.m|sales|12/12/52|6000
9876|jai sharma|director|production|12/03/50|7000
5678|sumit chakrobarty|dgm|mrking|19/04/43|6000

The remembered patterns

Consider the below three lines which does the same job

sed ‘s/director/member/’ emp.lst

sed ‘/director/s//member/’ emp.lst

sed ‘/director/s/director/member/’ emp.lst

The // representing an empty regular expression is interpreted to mean that the search and substituted patterns are the same

sed ‘s/|//g’ emp.lst removes every | from file

Basic Regular Expressions (BRE) – Revisited

Three more additional types of expressions are:
The repeated patterns - &
The interval regular expression (IRE) – { }
The tagged regular expression (TRE) – ( )

The repeated patterns - &

To make the entire source pattern appear at the destination also

sed ‘s/director/executive director/’ emp.lst

sed ‘s/director/executive &/’ emp.lst

sed ‘/director/s//executive &/’ emp.lst

Replaces director with executive director where & is a repeated pattern

The interval RE - { }

sed and grep uses IRE that uses an integer to specify the number of characters preceding a pattern. The IRE uses an escaped pair of curly braces and takes three forms:

ch\{m\} – the ch can occur m times

ch\{m,n\} – ch can occur between m and n times

ch\{m,\} – ch can occur at least m times

The value of m and n can't exceed 255. Let teledir.txt maintains landline and mobile phone numbers. To select only mobile numbers, use IRE to indicate that a numerical can occur 10 times.

grep ‘[0-9]\{10\}’ teledir.txt

Line length between 101 and 150

```
grep '^.\{101,150\}$' foo
```

Line length at least 101

```
sed –n '/.{101,\}/p' foo
```

The Tagged Regular Expression (TRE)

You have to identify the segments of a line that you wish to extract and enclose each segment with a matched pair of escaped parenthesis. If we need to extract a number, \([0-9]*\). If we need to extract non alphabetic characters,

```
\([^a-zA-Z]*\)
```

Every grouped pattern automatically acquires the numeric label n, where n signifies the nth group from the left.

```
sed 's/ \ (a-z]*\) *\ ([a-z]*\) / \2, \1/' teledir.txt
```

To get surname first followed by a , and then the name and rest of the line. sed does not use compulsorily a / to delimit patterns for substitution. We can use only any character provided it doesn't occur in the entire command line. Choosing a different delimiter has allowed us to get away without escaping the / which actually occurs in the pattern.

UNIT 6

6. Essential Shell Programming 6 Hours

Text Book

6. **"UNIX – Concepts and Applications",** Sumitabha Das, 4th Edition, Tata McGraw Hill, 2006.

(Chapters 1.2, 2, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 18, 19).

Reference Books

UNIX and Shell Programming, Behrouz A. Forouzan and Richard F. Gilberg, Thomson, 2005.

Unix & Shell Programming, M.G. Venkateshmurthy, Pearson Education, 2005.

Essential Shell Programming

Definition:

Shell is an agency that sits between the user and the UNIX system.

Description:

Shell is the one which understands all user directives and carries them out. It processes the commands issued by the user. The content is based on a type of shell called Bourne shell.

Shell Scripts

When groups of command have to be executed regularly, they should be stored in a file, and the file itself executed as a shell script or a shell program by the user. A shell program runs in interpretive mode. It is not complied with a separate executable file as with a C program but each statement is loaded into memory when it is to be executed. Hence shell scripts run slower than the programs written in high-level language. .sh is used as an extension for shell scripts. However the use of extension is not mandatory.

Shell scripts are executed in a separate child shell process which may or may not be same as the login shell.

Example: script.sh

```
#! /bin/sh
# script.sh: Sample Shell Script
echo “Welcome to Shell Programming”
echo “Today’s date : ‘date‘”
echo “This months calendar:”
cal ‘date “+%m 20%y”‘                    #This month’s calendar.
echo “My Shell :$ SHELL”
```

The # character indicates the comments in the shell script and all the characters that follow the # symbol are ignored by the shell. However, this does not apply to the first line which beings with #. This because, it is an interpreter line which always begins with #! followed by the pathname of the shell to be used for running the script. In the above example the first line indicates that we are using a Bourne Shell.

To run the script we need to first make it executable. This is achieved by using the chmod command as shown below:

$ chmod +x script.sh

Then invoke the script name as:

$ script.sh

Once this is done, we can see the following output :

Welcome to Shell Programming

Today's date: Mon Oct 8 08:02:45 IST 2007

This month's calendar:

October 2007

Su	Mo	Tu	We	Th	Fr	Sa
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

My Shell: /bin/Sh

As stated above the child shell reads and executes each statement in interpretive mode. We can also explicitly spawn a child of your choice with the script name as argument:

sh script.sh

Note: Here the script neither requires a executable permission nor an interpreter line.

Read: Making scripts interactive

The read statement is the shell's internal tool for making scripts interactive (i.e. taking input from the user). It is used with one or more variables. Inputs supplied with the standard input are read into these variables. For instance, the use of statement like

read name

causes the script to pause at that point to take input from the keyboard. Whatever is entered by you will be stored in the variable *name*.

Example: A shell script that uses read to take a search string and filename from the terminal.

```
#! /bin/sh
# emp1.sh: Interactive version, uses read to accept two inputs
#
echo "Enter the pattern to be searched: \c"          # No newline
read pname
echo "Enter the file to be used: \c"                 # use echo –e in bash
read fname
echo "Searching for pattern $pname from the file $fname"
grep $pname $fname
echo "Selected records shown above"
```

Running of the above script by specifying the inputs when the script pauses twice:

```
$ emp1.sh
Enter the pattern to be searched : director
Enter the file to be used: emp.lst
Searching for pattern director from the file emp.lst

9876    Jai Sharma      Director    Productions
2356    Rohit           Director    Sales
Selected records shown above
```

Using Command Line Arguments

Shell scripts also accept arguments from the command line. Therefore e they can be run non interactively and be used with redirection and pipelines. The arguments are assigned to special shell variables. Represented by $1, $2, etc; similar to C command arguments argv[0], argv[1], etc. The following table lists the different shell parameters.

Shell parameter	Significance
$1, $2…	Positional parameters representing command line arguments
$ #	No. of arguments specified in command line
$ 0	Name of the executed command
$ *	Complete set of positional parameters as a single string
"$ @"	Each quoted string treated as separate argument
$?	Exit status of last command
$$	Pid of the current shell
$!	PID of the last background job.

Table: shell parameters

exit and Exit Status of Command

To terminate a program exit is used. Nonzero value indicates an error condition.

Example 1:

$ cat foo

Cat: can't open foo

Returns nonzero exit status. The shell variable $? Stores this status.

Example 2:

grep director emp.lst > /dev/null:echo $?
0

Exit status is used to devise program logic that braches into different paths depending on success or failure of a command

The logical Operators && and ||

The shell provides two operators that aloe conditional execution, the && and ||.

Usage:

cmd1 && cmd2

cmd1 || cmd2

&& delimits two commands. cmd 2 executed only when cmd1 succeeds.

Example1:

$ grep ‘director’ emp.lst && echo “Pattern found”

Output:

9876 Jai Sharma Director Productions

2356 Rohit Director Sales

Pattern found

Example 2:

$ grep ‘clerk’ emp.lst || echo “Pattern not found”

Output:

Pattern not found

Example 3:

grep “$1” $2 || exit 2

echo “Pattern Found Job Over”

The if Conditional

The if statement makes two way decisions based on the result of a condition. The following forms of if are available in the shell:

Form 1

```
if command is successful
then
  execute commands
fi
```

Form 2

```
if command is successful
then
  execute commands
else
  execute commands
fi
```

Form 3

```
if command is successful
then
  execute commands
elif command is successful
then...
else...
fi
```

If the command succeeds, the statements within if are executed or else statements in else block are executed (if else present).

Example:

```
#! /bin/sh
if grep "^$1" /etc/passwd 2>/dev/null
then
        echo "Pattern Found"
else
        echo "Pattern Not Found"
fi
```

Output1:

```
$ emp3.sh ftp
ftp: *.325:15:FTP User:/Users1/home/ftp:/bin/true
Pattern Found
```

Output2:

$ emp3.sh mail

Pattern Not Found

While: Looping

To carry out a set of instruction repeatedly shell offers three features namely while, until and for.

Syntax:

while condition is true

do

Commands

done

The commands enclosed by do and done are executed repeatedly as long as condition is true.

Example:

```
  #! /bin/usr
 ans=y
while [“$ans”=”y”]
do
        echo “Enter the code and description : \c” > /dev/tty
        read code description
        echo “$code $description” >>newlist
        echo “Enter any more [Y/N]”
        read any
        case $any in
        Y* | y* ) answer =y;;
        N* | n*) answer = n;;
                *) answer=y;;
        esac
done
```

Input:

Enter the code and description : 03 analgestics

Enter any more [Y/N] :y

Enter the code and description : 04 antibiotics

Enter any more [Y/N] : [Enter]

Enter the code and description : 05 OTC drugs

Enter any more [Y/N] : n

Output:

$ cat newlist

03 | analgestics

04 | antibiotics

05 | OTC drugs

Using test and [ ] to Evaluate Expressions

Test statement is used to handle the true or false value returned by expressions, and it is not possible with if statement. Test uses certain operators to evaluate the condition on its right and returns either a true or false exit status, which is then used by if for making decisions. Test works in three ways:

- Compare two numbers
- Compares two strings or a single one for a null value
- Checks files attributes

Test doesn't display any output but simply returns a value that sets the parameters $?

Numeric Comparison

Operator	Meaning
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-ge	Greater than or equal to
-lt	Less than
-le	Less than or equal

Table: Operators

Operators always begin with a – (Hyphen) followed by a two word character word and enclosed on either side by whitespace.

Numeric comparison in the shell is confined to integer values only, decimal values are simply truncated.

Ex:
$x=5;y=7;z=7.2

1. $test $x –eq $y; echo $?
   1 *Not equal*

2. $test $x –lt $y; echo $?
   0 *True*

3. $test $z –gt $y; echo $?
   1 *7.2 is not greater than 7*
   2
4. $test $z –eq $y ; echo $y
   0 *7.2 is equal to 7*
   1

Example 3 and 4 shows that test uses only integer comparison.

The script emp.sh uses test in an if-elif-else-fi construct (Form 3) to evaluate the shell parameter $#

```
#!/bin/sh
#emp.sh: using test, $0 and $# in an if-elif-else-fi construct
#
If test $# -eq 0; then
Echo “Usage : $0 pattern file” > /dev/tty
Elfi test $# -eq 2 ;then
Grep “$1” $2 || echo “$1 not found in $2”>/dev/tty
Else
echo “You didn’t enter two arguments” >/dev/tty
fi
```

It displays the usage when no arguments are input, runs grep if two arguments are entered and displays an error message otherwise.

Run the script four times and redirect the output every time

```
$emp31.sh>foo
Usage : emp.sh pattern file
```

$emp31.sh ftp>foo
You didn't enter two arguments
$emp31.sh henry /etc/passwd>foo
Henry not found in /etc/passwd
$emp31.sh ftp /etc/passwd>foo
ftp:*:325:15:FTP User:/user1/home/ftp:/bin/true

Shorthand for test

[and] can be used instead of test. The following two forms are equivalent

Test $x –eq $y

and

[$x –eq $y]

String Comparison

Test command is also used for testing strings. Test can be used to compare strings with the following set of comparison operators as listed below.

Test	True if
s1=s2	String s1=s2
s1!=s2	String s1 is not equal to s2
-n stg	String stg is not a null string
-z stg	String stg is a null string
stg	String stg is assigned and not null
s1= =s2	String s1=s2

Table: String test used by test

Example:

#!/bin/sh
#emp1.sh checks user input for null values finally turns emp.sh developed previously
#

if [$# -eq 0] ; then
echo "Enter the string to be searched :\c"
read pname
if [-z "$pname"] ; then
echo "You have not entered th e string"; exit 1

```
fi
echo “Enter the filename to be used :\c”
read flname
if [ ! –n “$flname” ] ; then
echo “ You have not entered the flname” ; exit 2
fi
emp.sh “$pname” “$flname”
else
emp.sh $*
fi
```

Output1:
$emp1.sh
Enter the string to be searched :[Enter]
You have not entered the string

Output2:
$emp1.sh
Enter the string to be searched :root
Enter the filename to be searched :/etc/passwd
Root:x:0:1:Super-user:/:/usr/bin/bash

When we run the script with arguments emp1.sh bypasses all the above activities and calls emp.sh to perform all validation checks

$emp1.sh jai
You didn’t enter two arguments

$emp1.sh jai emp,lst
9878|jai sharma|director|sales|12/03/56|70000

$emp1.sh “jai sharma” emp.lst
You didn’t enter two arguments

Because $* treats jai and sharma are separate arguments. And $# makes a wrong argument count. Solution is replace $* with “$@” (with quote” and then run the script.

File Tests

Test can be used to test various file attributes like its type (file, directory or symbolic links) or its permission (read, write. Execute, SUID, etc).

Example:

```
$ ls –l emp.lst
-rw-rw-rw-      1 kumar group          870 jun 8 15:52 emp.lst
$ [-f emp.lst] ; echo $?                          → Ordinary file
0
$ [-x emp.lst] ; echo $?                          → Not an executable.
1
$ [! -w emp.lst]  || echo “False that file not writeable”
False that file is not writable.
```

Example: filetest.sh

```
#! /bin/usr
#
if [! –e $1] : then
        Echo “File doesnot exist”
elif [! –r S1]; then
        Echo “File not readable”
elif[! –w $1]; then
        Echo “File not writable”
else
        Echo “File is both readable and writable”\
fi
```

Output:

```
$ filetest.sh emp3.lst
```

File does not exist

$ filetest.sh emp.lst

File is both readable and writable

The following table depicts file-related Tests with test:

Test	True if
-f file	File exists and is a regular file
-r file	File exists and readable
-w file	File exists and is writable
-x file	File exists and is executable
-d file	File exists and is a directory
-s file	File exists and has a size greater than zero
-e file	File exists (Korn & Bash Only)
-u file	File exists and has SUID bit set
-k file	File exists and has sticky bit set
-L file	File exists and is a symbolic link (Korn & Bash Only)
f1 –nt f2	File f1 is newer than f2 (Korn & Bash Only)
f1 –ot f2	File f1 is older than f2 (Korn & Bash Only)
f1 –ef f2	File f1 is linked to f2 (Korn & Bash Only)

Table: file-related Tests with test

The case Conditional

The case statement is the second conditional offered by the shell. It doesn't have a parallel either in C (Switch is similar) or perl. The statement matches an expression for more than one alternative, and uses a compact construct to permit multiway branching. case also handles string tests, but in a more efficient manner than if.

Syntax:

```
case expression in
        Pattern1) commands1 ;;
        Pattern2) commands2 ;;
```

```
        Pattern3) commands3 ;;
        …
Esac
```

Case first matches expression with pattern1. if the match succeeds, then it executes commands1, which may be one or more commands. If the match fails, then pattern2 is matched and so forth. Each command list is terminated with a pair of semicolon and the entire construct is closed with esac (reverse of case).

Example:

```
#! /bin/sh
#
echo "          Menu\n
1. List of files\n2. Processes of user\n3. Today's Date
4. Users of system\n5.Quit\nEnter your option: \c"
read choice
case "$choice" in
            1) ls –l;;
            2) ps –f ;;
            3) date ;;
            4) who ;;
            5) exit ;;
            *) echo "Invalid option"
esac
```

Output

```
$ menu.sh

        Menu

1. List of files
2. Processes of user
3. Today's Date
4. Users of system
5. Quit
Enter your option: 3
Mon Oct 8 08:02:45 IST 2007
```

Note:

- case can not handle relational and file test, but it matches strings with compact code. It is very effective when the string is fetched by command substitution.
- case can also handle numbers but treats them as strings.

Matching Multiple Patterns:

case can also specify the same action for more than one pattern . For instance to test a user response for both y and Y (or n and N).

Example:

```
Echo “Do you wish to continue? [y/n]: \c”
Read ans
Case “$ans” in
        Y | y );;
        N | n) exit ;;
esac
```

Wild-Cards: case uses them:

case has a superb string matching feature that uses wild-cards. It uses the filename matching metacharacters *, ? and character class (to match only strings and not files in the current directory).

Example:

```
Case “$ans” in
        [Yy] [eE]* );;                  Matches YES, yes, Yes, yEs, etc
        [Nn] [oO]) exit ;;              Matches no, NO, No, nO
              *)  echo “Invalid Response”
esac
```

expr: Computation and String Handling

The Broune shell uses expr command to perform computations. This command combines the following two functions:

- Performs arithmetic operations on integers
- Manipulates strings

Computation:

expr can perform the four basic arithmetic operations (+, -, *, /), as well as modulus (%) functions.

Examples:

```
$ x=3 y=5

$ expr  3+5
8

$ expr  $x-$y
-2

$ expr  3 \* 5          Note:\ is used to prevent the shell from interpreting * as metacharacter
15

$ expr  $y/$x
1

$ expr  13%5
3
```

expr is also used with command substitution to assign a variable.

Example1:

```
$ x=6 y=2 : z=`expr $x+$y`
$ echo $z
8
```

Example2:

```
$ x=5
$ x=`expr $x+1`
$ echo $x
6
```

String Handling:

expr is also used to handle strings. For manipulating strings, expr uses two expressions separated by a colon (:). The string to be worked upon is closed on the left of the colon and a regular expression is placed on its right. Depending on the composition of the expression expr can perform the following three functions:

1. Determine the length of the string.
2. Extract the substring.
3. Locate the position of a character in a string.

1. Length of the string:

The regular expression .* is used to print the number of characters matching the pattern .

Example1:

```
$ expr “abcdefg” : ‘.*’
7
```

Example2:

```
while echo “Enter your name: \c” ;do
        read name
        if [‘expe “$name” :’.*’‘    -gt  20] ; then
                echo “Name is very long”
        else
                break
        fi
done
```

2. Extracting a substring:

expr can extract a string enclosed by the escape characters \ (and \).

Example:

```
$ st=2007
$ expr “$st” :’..\(..\)’
07                                  Extracts last two characters.
```

3. **Locating position of a character:**

   expr can return the location of the first occurrence of a character inside a string.

   Example:

   $ stg = abcdefgh ; expr "$stg" : '[^d]*d'
   4 Extracts the p*osition of character d*

$0: Calling a Script by Different Names

There are a number of UNIX commands that can be used to call a file by different names and doing different things depending on the name by which it is called. $0 can also be to call a script by different names.

Example:

```
#! /bin/sh
#
lastfile=`ls –t *.c |head -1`
command=$0
exe='expr $lastfile: '\(.*\).c''
case $command in
        *runc) $exe ;;
        *vic) vi $lastfile;;
        *comc) cc –o $exe $lastfile &&
                Echo "$lastfile compiled successfully";;
esac
```

After this create the following three links:

```
ln comc.sh comc
ln comc.sh runc
ln comc.sh vic
```

Output:

```
$ comc
hello.c compiled successfully.
```

While: Looping

To carry out a set of instruction repeatedly shell offers three features namely while, until and for.

Synatx:

```
while condition is true
do
        Commands
done
```

The commands enclosed by do and done are executed repadetedly as long as condition is true.

Example:

```
  #! /bin/usr
 ans=y
while [“$ans”=”y”]
do
        echo “Enter the code and description : \c” > /dev/tty
        read code description
        echo “$code $description” >>newlist
        echo “Enter any more [Y/N]”
        read any
        case $any in
        Y* | y* ) answer =y;;
        N* | n*) answer = n;;
                *) answer=y;;
        esac
done
```

Input:

Enter the code and description : 03 analgestics

Enter any more [Y/N] :y

Enter the code and description : 04 antibiotics

Enter any more [Y/N] : [Enter]

Enter the code and description : 05 OTC drugs

Enter any more [Y/N] : n

Output:

$ cat newlist

03 | analgestics

04 | antibiotics

05 | OTC drugs

Other Examples: An infinite/semi-infinite loop

(1)

```
while true ; do
   [ -r $1 ] && break
   sleep $2
done
```

(2)

```
while [ ! -r $1 ] ; do
   sleep $2
done
```

for: Looping with a List

for is also a repetitive structure.

Synatx:

for variable in list

do

Commands

done

list here comprises a series of character strings. Each string is assigned to variable specified.

Example:

```
for file in ch1 ch2; do
> cp $file ${file}.bak
> echo $file copied to $file.bak
done
```

Output:

```
ch1 copied to ch1.bak
ch2 copied to ch2.bak
```

Sources of list:

- **List from variables**: Series of variables are evaluated by the shell before executing the loop

  Example:

  $ for var in $PATH $HOME; do echo "$var" ; done

  Output:
  /bin:/usr/bin;/home/local/bin;
  /home/user1

- **List from command substitution**: Command substitution is used for creating a list. This is used when list is large.

  Example:

  $ for var in `cat clist`

- **List from wildcards**: Here the shell interprets the wildcards as filenames.

  Example:

```
for file in *.htm *.html ; do
        sed 's/strong/STRONG/g
        s/img src/IMG SRC/g' $file > $$
        mv $$ $file
done
```

- **List from positional parameters**:

  Example: emp.sh

  ```
  #! /bin/sh
  for pattern in "$@"; do
  grep "$pattern" emp.lst || echo "Pattern $pattern not found"
  done
  ```

  Output:

  $emp.sh 9876 "Rohit"

9876	Jai Sharma	Director	Productions
2356	Rohit	Director	Sales

basename: Changing Filename Extensions

They are useful in chaining the extension of group of files. Basename extracts the base filename from an absolute pathname.

Example1:

$basename /home/user1/test.pl

Ouput:

test.pl

Example2:

$basename test2.doc doc

Ouput:

test2

Example3: Renaming filename extension from .txt to .doc

```
for file in *.txt ; do
    leftname=`basename $file .txt` Stores left part of filename
     mv $file ${leftname}.doc
done
```

set and shift: Manipulating the Positional Parameters

The set statement assigns positional parameters $1, $2 and so on, to its arguments. This is used for picking up individual fields from the output of a program.

Example 1:

$ set 9876 2345 6213

$

This assigns the value 9876 to the positional parameters $1, 2345 to $2 and 6213 to $3. It also sets the other parameters $# and $*.

Example 2:

$ set `date`

$ echo $*

Mon Oct 8 08:02:45 IST 2007

Example 3:

$ echo “The date today is $2 $3, $6”

The date today is Oct 8, 2007

Shift: Shifting Arguments Left

Shift transfers the contents of positional parameters to its immediate lower numbered one. This is done as many times as the statement is called. When called once, $2 becomes $1, $3 becomes S2 and so on.

Example 1:

$ echo “$@” *$@ and $* are interchangeable*

Mon Oct 8 08:02:45 IST 2007

$ echo $1 $2 $3

Mon Oct 8

```
$shift
$echo $1 $2 $3
Mon Oct 8 08:02:45
$shift 2                                    Shifts 2 places
$echo $1 $2 $3
08:02:45 IST 2007
```

Example 2: emp.sh

```
#! /bin/sh
Case $#  in
        0|1) echo "Usage: $0 file pattern(S)" ;exit ;;
          *) fname=$1
           shift
           for pattern in "$@" ; do
                  grep "$pattern" $fname || echo "Pattern $pattern not found"
        done;;
esac
```

Output:

```
$emp.sh emp.lst
Insufficient number of arguments
$emp.sh emp.lst Rakesh  1006 9877

9876    Jai Sharma    Director    Productions
2356    Rohit         Director    Sales

Pattern 9877 not found.
```

Set -- : Helps Command Substitution

Inorder for the set to interpret - and null output produced by UNIX commands the – option is used . If not used – in the output is treated as an option and set will interpret it wrongly. In case of null, all variables are displayed instead of null.

Example:

$set `ls –l chp1`

Output:

-rwxr-xr-x: bad options

Example2:

$set `grep usr1 /etc/passwd`

Correction to be made to get correct output are:

$set -- `ls –l chp1`

$set -- `grep usr1 /etc/passwd`

The Here Document (<<)

The shell uses the << symbol to read data from the same file containing the script. This is referred to as a here document, signifying that the data is here rather than in aspirate file. Any command using standard input can slo take input from a here document.

Example:

```
mailx kumar << MARK
Your program for printing the invoices has been executed
on `date`.Check the print queue
The updated file is $flname
MARK
```

The string (MARK) is delimiter. The shell treats every line following the command and delimited by MARK as input to the command. Kumar at the other end will see three lines of message text with the date inserted by command. The word MARK itself doesn't show up.

Using Here Document with Interactive Programs:

A shell script can be made to work non-interactively by supplying inputs through here document.

Example:

```
$ search.sh << END
> director
>emp.lst
>END
```

Output:

Enter the pattern to be searched: Enter the file to be used: Searching for director from file emp.lst

9876	Jai Sharma	Director	Productions
2356	Rohit	Director	Sales

Selected records shown above.

The script search.sh will run non-interactively and display the lines containing “director” in the file emp.lst.

trap: interrupting a Program

Normally, the shell scripts terminate whenever the interrupt key is pressed. It is not a good programming practice because a lot of temporary files will be stored on disk. The trap statement lets you do the things you want to do when a script receives a signal. The trap statement is normally placed at the beginning of the shell script and uses two lists:

trap ‘command_list’ signal_list

When a script is sent any of the signals in signal_list, trap executes the commands in command_list. The signal list can contain the integer values or names (without SIG prefix) of one or more signals – the ones used with the kill command.

Example: To remove all temporary files named after the PID number of the shell:

trap ‘rm $$* ; echo “Program Interrupted” ; exit’ HUP INT TERM

trap is a signal handler. It first removes all files expanded from $$*, echoes a message and finally terminates the script when signals SIGHUP (1), SIGINT (2) or SIGTERM(15) are sent to the shell process running the script.

A script can also be made to ignore the signals by using a null command list.

Example:

trap ‘’ 1 2 15

Programs

1)

```
#!/bin/sh
IFS="|"
While echo "enter dept code:\c"; do
Read dcode
Set -- ‘grep "^$dcode"<<limit
01|ISE|22
02|CSE|45
03|ECE|25
04|TCE|58
limit`
Case $# in
3) echo "dept name :$2 \n  emp-id:$3\n"
*) echo "invalid code";continue
esac
done
```

Output:

```
$valcode.sh
Enter dept code:88
Invalid code
Enter dept code:02
Dept name  : CSE
Emp-id :45
Enter dept code:<ctrl-c>
```

2)
```
#!/bin/sh
x=1
While [$x –le 10];do
 echo “$x”
 x=`expr $x+1`
done
#!/bin/sh
sum=0
for I in “$@” do
 echo “$I”
sum=`expr $sum + $I`
done
Echo “sum is $sum”
```

3)
```
#!/bin/sh
sum=0
for I in `cat list`; do
 echo “string is $I”
x= ‘expr “$I”:’.*’‘
Echo “length is $x”
Done
```

4)
This is a non-recursive shell script that accepts any number of arguments and prints them in a reverse order.

For example if A B C are entered then output is C B A.

```
#!/bin/sh
if [ $# -lt 2 ]; then
echo "please enter 2 or more arguments"
exit
fi
for x in $@
do
y=$x" "$y
done
echo "$y"
```

Run1:
```
[root@localhost shellprgms]# sh sh1a.sh 1 2 3 4 5 6 7

7 6 5 4 3 2 1
```

Run2: [root@localhost shellprgms]# sh ps1a.sh this is an argument

argument an is this

5)

The following shell script to accept 2 file names checks if the permission for these files are identical and if they are not identical outputs each filename followed by permission.

```
#!/bin/sh
if [ $# -lt 2 ]
then
echo "invalid number of arguments"
exit
fi
str1=`ls -l $1|cut -c 2-10`
str2=`ls -l $2|cut -c 2-10`

if [ "$str1" = "$str2" ]
then
echo "the file permissions are the same: $str1"
else
echo " Different file permissions "
echo -e "file permission for $1 is $str1\nfile permission for $2 is  $str2"
fi
```

Run1:

```
[root@localhost shellprgms]# sh 2a.sh ab.c xy.c
file permission for ab.c is rw-r--r--
file permission for xy.c is  rwxr-xr-x
```

Run2:

```
[root@localhost shellprgms]# chmod +x ab.c
[root@localhost shellprgms]# sh 2a.sh ab.c xy.c
```

the file permissions are the same: rwxr-xr-x

6)This shell function that takes a valid directory name as an argument and recursively descends all the subdirectories, finds the maximum length of any file in that hierarchy and writes this maximum value to the standard output.

```
#!/bin/sh
if [ $# -gt 2 ]
then
echo "usage sh flname dir"
exit
fi
if [ -d $1 ]
then
ls -lR $1|grep -v ^d|cut -c 34-43,56-69|sort -n|tail -1>fn1
echo "file name is `cut -c 10- fn1`"
echo " the size is `cut -c -9 fn1`"
else
echo "invalid dir name"
fi
```

Run1:

```
[root@localhost shellprgms]# sh 3a.sh
file name is   a.out
the size is     12172
```

7)This shell script that accepts valid log-in names as arguments and prints their corresponding home directories. If no arguments are specified, print a suitable error message.

```
if [ $# -lt 1 ]
then
echo " Invlaid Arguments..... "
exit
fi
for x in "$@"
```

```
do
  grep -w "^$x" /etc/passwd | cut -d ":" -f 1,6
done
```

Run1:

```
[root@localhost shellprgms]# sh 4a.sh root
root:/root
```

Run2:

```
[root@localhost shellprgms]# sh 4a.sh
Invalid Arguments.....
```

8) This shell script finds and displays all the links of a file specified as the first argument to the script. The second argument, which is optional, can be used to specify the directory in which the search is to begin. If this second argument is not present .the search is to begin in current working directory.

```
#!/bin/bash
if [ $# -eq 0 ]
then
        echo "Usage:sh 8a.sh[file1] [dir1(optional)]"
        exit
fi
if [ -f $1 ]
then
        dir="."
if [ $# -eq 2 ]
then
        dir=$2
fi
inode=`ls -i $1|cut -d " " -f 2`
echo "Hard links of $1 are"
find $dir -inum $inode -print
```

```
echo "Soft links of $1 are"
find $dir -lname $1 -print
else
echo "The file $1 does not exist"
fi
```

Run1:

```
[root@localhost shellprgms]$ sh 5a.sh  hai.c
Hard links of hai.c are
./hai.c
Soft links of hai.c are
./hai_soft
```

9) This shell script displays the calendar for current month with current date replaced by * or ** depending on whether date has one digit or two digits.

```
#!/bin/bash
n=` date +%d`
echo " Today's date is : `date +%d%h%y` ";
cal > calfile
if [ $n -gt 9 ]
then
 sed "s/$n/\**/g" calfile
else
 sed "s/$n/\*/g" calfile
[root@localhost shellprgms]# sh 6a.sh
 Today's date is : 10 May 05
      May 2005
Su  Mo  Tu  We  Th  Fr  Sa
 1   2    3   4   5   6   7
 8   9   **  11  12  13  14
15  16   17  18  19  20  21
```

22 23 24 25 26 27 28

29 30 31

10) This shell script implements terminal locking. Prompt the user for a password after accepting, prompt for confirmation, if match occurs it must lock and ask for password, if it matches terminal must be unlocked

```
trap “ " 1 2 3 5 20
clear
echo -e “\nenter password to lock terminal:"
stty -echo
read keynew
stty echo
echo -e “\nconfirm password:"
stty -echo
read keyold
stty echo
if [ $keyold = $keynew ]
then
echo "terminal locked!"
while [ 1 ]
do
echo "retype the password to unlock:"
stty -echo
read key
if [ $key = $keynew ]
then
stty echo
echo "terminal unlocked!"
stty sane
exit
fi
echo "invalid password!"
done
else
```

```
echo " passwords do not match!"
fi
stty sane
```

Run1:

```
[root@localhost shellprgms]# sh 13.sh
enter password:
confirm password:
terminal locked!
retype the password to unlock:
invalid password!
retype the password to unlock:
terminal unlocked!
```

****