Miracle – UnixExercises

Key People:

TrainingAd visor: Raju Cherukuri

Date: Nov/20/2005

Author Details:
Prepared by: Raju Cherukuri
Email: *scherukuri@miraclesoft.com*
Phone no: 1-248-233-3654

Document Details:
Version: 3.0
Date: Thursday, January 12, 2006

Table Of Contents

Chapter 1

1. UNIX : INTRODUCTION

1.1. Objective of Chapter 1

The objective of this chapter is to introduce you to an Operating System and in particular to UNIX. First you get introduced to the software called operating system. Then you will be taught about the UNIX operating system – its history, features, architecture. You also get to know about the concept of files and the UNIX file system. Finally, you are taught how to use UNIX with a few simple and often used commands.

In this chapter you will learn the following:

- ❖ Introduction to operating system
- ❖ History of UNIX operating system
- ❖ Features of UNIX operating system
- ❖ Unix architecture
- ❖ Unix file system
- ❖ Getting started using UNIX operating system
- ❖ Simple UNIX commands

1.2. INTRODUCTION TO OPERATING SYTEM (OS)

An operating system is an important part of a computer system. A computer system can be described as a collection of hardware and software components. Hardware components are those components which you can touch physically, like the Central Processing Unit (CPU), memory, keyboard, hard disk, printer, and so on. The software components are sets of instructions, also referred as programs, to the hardware (CPU) to perform certain operations to bring out specific results like play a game, prepare a letter, send electronic mail etc.. The most important software component of a computer system is the Operating System (OS).

The Operating system is a supervisor program that takes care of coordinating the operations of the machine from the time it is switched on till it is switched off. When the computer is turned on, the OS takes care of all the starting functions that must occur to get the computer to a usable state. Various pieces of hardware are initialized.

The main activities of an operating system are:

Running a program. When you enter a filename on your terminal at the command line, the operating system interprets your command, loads the program into computer's memory and executes the program. When more than one program/process can run simultaneously on the system, the operating system takes care of scheduling the CPU for various processes, managing the computer's memory and other related tasks.

Controlling the input and output operations of the computer: When you do any file related operations like deleting a file from the hard disk or saving a file on to the disk, the operating system ensures that the operation is carried out properly without erasing a different file or overwriting an existing file etc. Similarly, it takes care of displaying the output on your terminal and accepting input from devices like keyboard or mouse and printing the documents you have specified on the printer connected to the system.

When the computer is shutdown, the operating system makes sure all the hardware is shutdown correctly. Without an Operating system, no application can run, and the computer is just an expensive paperweight.

1.3. THE HISTORY OF UNIX OPERATING SYSTEM

IDG) – The operating system took 30 years to evolve into what it is today. Here are some key events that led to its development.

1971 The first edition of the Unix server operating system emerges from Bell Labs. Although Linux does not include any Unix code, it is a Unix clone, which means it shares a number of technical features with Unix, which might be considered the forerunner of the open-source operating system. During the 1970s, Unix code was distributed to people at various universities and companies, and they created their own Unix varieties, which ultimately evolved into Sun Microsystems' Solaris, Berkeley's FreeBSD and silicon Graphics' IRIX.

1985 Richard Stallman publishes his famous "GNU Manifesto", one of the first documents of the open-source revolution. Stallman began working on the GNU operating (GNU stands for "GNU is Not Unix.") Stallman's Free Software Foundation later created the GNU General Public License, the widely adopted, fully legal "anticopyright" treatise that today allows Linux and other software to remain completely free.

1987 Professor Aandrew S. Tanenbaum invents Minix, an open-source operating system that's clone of Unix. Young Linus Torvalds, at the time a computer science student in Finland, is introduced to Minix, and bases his plans for Linux on the Minix example.

1991 In August, Torvalds announces his plans to create a free operating system on the Minix users newsgroup. He modestly notes in his posting that his OS is "just a hobby. [It] won't be big and professional like GNU. " In Octorber, Linux 0.01 is released on the Internet under a GNU public license. In the Minix newsgroup, Torvalds asks his fellow programmers to lend a hand in making the system more workable. He gets enough help to release version 0.1 by December. Over the next several years, Linux developers swell Vendors like Red Hat, Caldera and Debian create popular distributions of Linux that bundle the operating system with useful programs and a graphical interface.

1997 Torvalds moves to Silicon Valley and goes to work at Transmeta.

'1999 In August, Red Hat completes its initial public offering, making it the first Linux-oriented company to successfully go public. In December, Andover.net, a consortium of Web site resources largely devoted to Linux, and VA Linux, a manufacturer of Linux hardware, have wildly successful IPOs. Linuxcare, a leading Linux service provider, announces alliances with such industry giants as IBM, Dell, Motorola and Informix.

1.4. FEATURES OF UNIX

UNIX is a comprehensive OS with a number of features and capabilities. Several reasons Were identified for the popularity and success of the UNIX System.

UNIX Operating System is written in "C" language which makes it easy to read, understand, update and port to other machines.

Multi-user OS: Multi-user refers to an OS that allows multiple users to use the system simultaneously. The theory of multi-user system is to approach 100% computer utilization while reducing the cost per user. A single user cannot use the printer, disk, memory or CPU 100% of the time. But multiple users can increase the use of these devices and resources by having an OS that manages the resources for them.

Multitasking system: Multitasking refers to an OS that executes multiple tasks simultaneously. UNIX refers to a task as a process. A user can run several commands in background while executing another command in the foreground. When a background task is being executed, user can continue doing another task e.g., printing a large document can be performed in the background while editing some other document in the foreground.

Portability: UNIX is highly portable. Portability is the ability of the software operating on one machine to operate as efficiently on another, different machine.

Job control: Job Control on UNIX refers to the ability to control which job is executed in the foreground, background or is suspended. Using Job control can increase the productivity of a user by allowing multiple tasks to be juggled back and forth between background, foreground and suspended states.

Hierarchical Structure: UNIX uses a hierarchical structure to store and maintain files. This structure allows maximum flexibility for storing information to resemble its real life structure. Multiple users may be grouped by corporate departments. As an individual user, you may group your data by project or subject.

The UNIX Shell: The shell is a very powerful and dynamic UNIX utility. It is the primary interface to the OS (kernel). It can be interactively programmed or it can be used to write scripts to solve simple to complex problems.

Pipes and Filters: Pipes and Filters contribute to the power of UNIX. These enable several commands or utilities to be combined to perform complex functions.

Device Independence: UNIX system considers all devices connected to it as files. It hides the machine architecture from the user, making it easier to write programs that run on different hardware implementations

System Security: Being a multi-user OS, UNIX offers protection to one user's information from other users. It maintains a list of users who are allowed to access the system and keeps track of what files and resources each user is authorized to use.

Communication: The UNIX system has several built in programs, enabling the user to communicate, transfer files across different UNIX systems and between UNIX and other OS system.

1.5. UNIX ARCHITECTURE

Like an onion, the UNIX system is built up of layer, with each layer representing a building block that can be used to build other building blocks and so on. Most programs and commands supplied with the UNIX system can be used in combination with each other to build other tools, and complex mechanisms can be built up from a single line of commands to perform vast assortments of functions.

The hardware at the center of the diagram provides the operating system with the basic hardware services. The Operating system interacts directly with the hardware, providing common services to programs and insulating them from hardware specifics.

The UNIX operating system is commonly called the **kernel.** It provides an interface for all other UNIX programs to use the hardware independent.. Whenever you log in to UNIX, you communicate with the kernel via shell program. It does not make itself available for the end user. The kernel provides the basic services of System initialization, Process management, Memory management, File System management, Communication facilities and Programmatic interface through system calls.

Programs such as the shell and editors (ed and vi) shown in the outer layers interact with the kernel by invoking a well defined set of *system* calls. The system calls instruct the Kernal to do various operations for the calling program and exchange of data between the kernel and the program.

One of the prominent features of UNIX system is its wide variety of powerful utility programs. You can use a utility to locate system information, manage files or the contents of files and manipulate the output of other utilities.

The shell is a utility program that acts as a command interpreter. It is primary interface to the kernel. The shell is a command language and a programming language. As a command language, it can be used to communicate interactively with the kernel. As a programming language, users can write shell scripts to solve simple to complex problems.

Several programs shown in the figure are in standard system configuration and are known as commands. Over 200 utility program (commands) are supplied with UNIX system that service various day to day processing requirement. These are also used to solve complex problems.

In addition to utility programs, there are a number of UNIX-based application programs like word processors, spreadsheets, database managers and language processors which form the outer most ring in the architecture.

1.6. UNIX FILE SYSTEM

Information in UNIX system is stored in files, which are much like ordinary office files. Each file has a name, contents, a place to keep it and some administrative information such as who owns it and how big it is. A file might contain a letter, or a list of names and addresses, or the source statements of a program, or data to be used by a program.

Files are kept in storage devices, usually disks. You can have a number of files on the disk. To organize these files, UNIX operating system divides the disk into various logical units, where each unit can contain a group of related files. The logical units are called ***file systems.***

Unix uses the hierarchical structure to store its files. This structure is referred to as 'inverted' tree structure from its resemblance to an upside down tree. The file system starts with one main directory called ***root***. Since the tree is upside down, the root is at the top. The root directory symbolized by '/' (forward slash) has multiple directories under it.

The name of a file is given by a path name that describes how to locate the file in the file system hierarchy. A path name is a sequence of component names separated by slash characters; a component is a sequence of characters that designates a file name that is uniquely contained in the previous (directory) component. A full path name starts with a slash character and specifies a file that can be found by starting at the file system root and traversing the file tree, following the branches that lead to successive component names of the path name.

1.6.1. Features of UNIX file system:

Hierarchical structure: Allows grouping of related information and efficient manipulation of these groups.

Dynamic file growth: Files grow as needed. Disk space is not wasted since only the amount required to store the current contents of the file is only used.

Stucture less files: There is no internal structure imposed on the contents of the files. Any structure given by the user can be used.

Security: Unauthorized users can be restricted from using a file.

Device independent: Input and output from a device are processed as if it were in a file. Therefore, programs that process data can also process data to and from a device.

1.6.2. Types of files

A UNIX file system may contain six different types of files:

1. Regular files
2. Directories
3. Special files
4. Named pipes (FIFO)
5. Links
6. Symbolic Links

Regular files: A regular file (also known as an ordinary file) contains arbitrary data in zero or more data blocks stored within a file system. These files may simply contain ASCII text, or binary data. Individual applications may store their files in a specific format. There is no structure imposed by the operating system about how a regular file must be made up.

Data blocks belonging to a regular file may not necessarily reside on disk in a contiguous order. However, the UNIX operating system hides this side effect from the user and presents a file as if it were a contiguous stream of bytes; the user does not need to be concerned with a file's underlying storage structure.

UNIX identifies files by a unique number called the index node (inode) number. A file has one and only on inode number although it may have many filenames. The inode numbers are maintained in a directory file along with the related filename.

Directory files: Directories are collection of files. For instance, a user may need to group all his project files into one directory. Each directory has a name and each file within the directory has a filename. Directories are special types of files since they provide mapping between the names of files and the files themselves. As a result, the structure of directories define the structure of the file system as a whole.

The directory consists of a table containing two fields: an inode number and a file name used to symbolically reference the inode.

Special files: Special files contain no data. Instead, they provide a mechanism to map physical devices to file names in a file system. Each device supported by the system, including memory, is associated with at least one special file. Special files have associated software incorporated into the kernel called device drivers.

There are two types of special files : block-special and character-special. A block-special file is associated with a block structured device such as a disk, which transfers data to te machine's memory in blocks, typically made up of 512, 1024 bytes. A character-special file is associated with any device that is not necessarily block structured. Terminals, system console, serial devices, tape drives are character-special files.

Links: The UNIX file system provides a facility for linking files together with different file names. This facility is called linking. The purpose of linking files together is to allow a single program to administer different names. Actually only one copy of data is stored in the file system. The linked files share the same inode number and only a directory entry is made for the file.

Symbolic links: A symbolic link is a data file containing the name of the file it is supposed to be linked to. A symbolic link can be created even if the file it is supposed to be linked to does not exist. The advantage of having symbolic link is when the file system is low in space but a new software package has to be installed in it, a directory can be made on another file system which is then symbolically linked to the name of the expected installation directory.

With symbolic links, both a directory entry and new inode are created. Additionally, a single data block is reserved for it containing the full pathname of the file it references.

FIFOs (Pipes): Pipes are used to join two or more UNIX processes together allowing the data to flow from now process to another without being stored on the desk. A pipe file is a special file that buffers up data received in its input so that a process that reads from its output receives the data on a first-in-first-out basis(FIFO). No data is associated with a pipe special file although it does use up a directory entry and inode.

1.6.3. File Naming Conventions

A filename can contain digits, characters, dot (.), hyphen (-), or underscore symbol.

A filename can be both uppercase and lowercase.

A directory can have all characters.

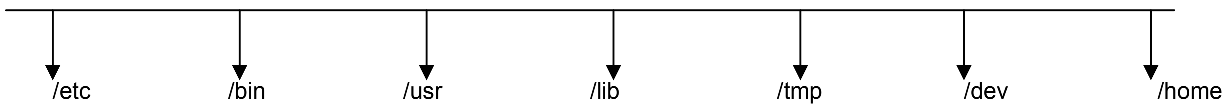
They are case sensitive. For instance, a filename myfile is different from MYFILE.

A filename should not contain white spaces.

File System Organization (Directory Hierarchy)

Directories are special files that contain names of other files and sub-directories. Typical directory structure of the UNIX system consists of the following directories:-

Figure 1.2



The top level directory is called the root directory and is denoted by a single / (forward slash). All the directories and files belongs to the root directory.

Following are the list of standard directory names in the UNIX file system.

/ Root directory. This is the parent of all the directories and files in the UNIX file system.

/etc System configuration files and executable directory. Most of the administrative, command-related files are stored here.

/bin Command-line executable directory. This directory contains all the UNIX native command executables

/usr Architecture dependent and architecture independent sharable files

/lib The library files for various programming languages such as C are stored in this directory

/tmp This directory is used for the temporary storage of files.

/dev Device directory containing special files for character- and block-oriented devices such a printers and keyboards. A file called null existing in this directory is called the bit bucket and can be used to redirect output to nowhere.

/home This directory contains data specific to an individual user.

1.7. GETTING STARTED

Several people can be using a UNIX system at the same time. In order for the system to know who you are and what resources you can use, you must identify yourself. This process of identifying yourself is know as login in. Before you can access the system, someone – usually the System Administrator – must configure the computer for your use.

You need a unique identification called the user name, which is the name through which you are identified by the system. To avoid others from using your name the system protects your account by providing a password. You are given the responsibility of keeping the password secure.

1.7.1. LOGIN

Once you have your user name and password, you can access the system. The system prompts you for your user name by printing login:

```
Login:
```

You respond to this prompt by typing a valid username (supplied to you by the system administrator) called the user_id. Press<**Enter**> key. In the next line it prompts you for a password:

```
Password:
```

Password is a sequence of letters and digits used to verify that you are allowed to use this user_id. Initially the password is supplied to you by the system administrator. Later you can change your password to some secret key, that you alone are aware. It is through the password only that the system knows that you are an authorized user.

When your respond to the password prompt, the characters you write are not written to the screen. This is to protect your password from others so that they might not use it later to tamper with your work.

After the valid username and password have been entered, the $ prompt is displayed on the screen. The $ prompt indicates that, now, UNIX system is ready to accept commands.

Example 1.1

```
Red Hat Enterprise Linux ES release 4 (Nahant)
Kernel 2.6.9-5.ELsmp on an i686
login: plokam

Password:
Last login: Tue Dec  6 15:51:42 from 192.168.5.100
[plokam@Linux1 ~]$
```

1.7.2. The Root User

There is a user id called "root" or super user. This user has special privileges. The root user has access to all parts of the UNIX operating system. There are no files that cannot be read by the super user, there are no portions of the file system inaccessible to the super user and there are no UNIX commands unavailable to the super user. The super user controlles all aspects of UNIX system usage and configuration. Incidentally, the system administrator who creates your account in the system is the super user. As super user, you are entitled to the special prompt: #

Note : Because you have no restrictions, as the root user you can seriously damage your UNIX system. A simple accident can cause you a lot of grief. So be very careful when you are logged in as the root user. This is the reason why you are not allowed to login as root user.

1.7.3. passwd

For changing the password you can use the following command.

```
[plokam@Linux1 ~]$ passwd
Changing password for user plokam.
Changing password for plokam
(current) UNIX password:
New UNIX password:
Retype new UNIX password:
passwd: all authentication tokens updated successfully.
[plokam@Linux1 ~]$
```

1.7.4. clear

When you want to clear the current screen you can clear it by using the clear.

```
$ clear
```

1.7.5. logout

When you are done using the system, you should log out. This will prevent other people from accidentally or intentionally getting access to your files. It will also make the system available for their use.

The normal way to log out is to type exit. Another way of logging out is to type the end-of-file character (typically Control + D). The Third way of doing is is by using the command called logout.

```
$ exit
or
$ ^D
or
```

```
$logout
```

1.7.6. shutdown

If you want to shutdown the machine and turn the power off you can execute the following command.

```
$ shutdown now
```

1.8. UNIX COMMANDS

Once you have successfully logged into a UNIX system, you can start using the system by issuing commands to it. Here, you will not directly be talking to the UNIX kernel but through a command interpreter or ***shell.*** The $ prompt that you see is called the command prompt or shell prompt.

A UNIX command is a series of characters that you type. These characters consist of words that are separated by whitespace. Whitespace is the result of typing one or more Space or Tab keys. The first word is the name of the command. The rest of the words are called the command's arguments. The arguments give the command information that it might need, or specify varying behavior of the command. To invoke a command, simply type the command name, followed by the arguments. The shell collects all the characters you have typed in till you press the enter key and then it interprets them.

Let us try out some simple and commonly used UNIX commands.

1.8.1. date

Displays the current date and current time of the system.

Sysntax

Date ["+<string> <options>"]

To display the current date and time

```
[plokam@Linux1 ~]$ date
Fri Dec  9 15:34:06 EST 2005
```

As can be seen above, the date command gives the day of the week, month, day, time (24 hour clock, Greenwich Meantime) and the year.

Note: Every command must be ended with a **<Enter>. <Enter>.** Informs the system that you have finished typing the command and are ready for the UNIX system to do the processing of the command.

To display the date without strings

```
[plokam@Linux1 ~]$ date +%D
12/09/05
```

To display the date with strings

```
[plokam@Linux1 ~]$ date  "+Current Date: %D"
Current Date: 12/09/05
```

The following options can be used in the date command

%D	displays the data in MM/DD/YY
%d	displays the day of the month(01-31)
%m	displays the month(01-12)
%y	displays the year

Example 1.2

```
[plokam@Linux1 ~]$  date "+ Current date [DD/MM/YY] : %d%m%y"
   Current date [DD/MM/YY] : 12/09/05
```

The following options can be used with the date command to display the abbreviated weekday and month

%a	displays abbreviated weekdays(Sun-Sat)
%A	displays abbreviated weekdays(Sunday-Saturday)
%b	displays abbreviated month(Jan-Dec)
%B	displays abbreviated month(January-December)

Example 1.3

```
$ date "+ Cnrent day: %a"
Current day: Fri
$ date "+current day: %A"
Current day: Friday
$ date "+Current month: %b"
Current month: Dec
```

Note: You can use %h to achieve the above result.

```
$ date "+Current month: $B"
```

```
Current month: September
```

The following options can be used with the date command to display the current time.

%H	displays the hour
%M	displays the minutes
%S	displays the seconds
%I	displays the IST time
%r	displays the time with meridian(AM/PM)
%n	displays the output in the newline

Example 1.4

```
$ date "+ 24 Hours Timing : %H : %M : %s %r"
24 Hours Timing :  13:35:31 PM

$ date "+IST Timings : %I : %M : %s %r"
IST Timings : 01 : 35 : 31 PM
```

To view more options use the following command

```
$ man date
```

Try the following

Display the date as abbreviated month, day and year
Eg. December 11,00

Display the day and hour only
Display the following using a single date command

Current Date [MM/DD/YY]:
Current Time[HH:MM:SS]:
Current Time with merdian

1.8.2. cal

The cal command generates a simple calendar as the output. By default, the output is the calendar for current month.

Syntax:

```
cal [ [month] year]
```

where

month Specifies the month to be displayed, represented as a decimal integer from 1 (*January*) to 12 (*December*). The default is the current month.

year Specifies the year for which the calendar is to be displayed, represented as a decimal integer from 1 to 9999. The default is the current year.

Example 1.8.2.1

Display the Calendar for the Month of November for Year 2005.

```
[plokam@Linux1 ~]$ cal 11 2005

    November 2005

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

[plokam@Linux1 ~]$
```

Note: Year must be entered as a four-digit number. For example, if you enter 99, the cal command will display the calendar for the year 99 A.D., and not the year 1999. If only one argument is given to the cal command, it is considered as the year and not the month. For example: cal 07 will display the calendar for the year 07 A.D., and not the month July.

1.8.3. Finger

The finger command displays a detailed list of the the user information. If you specify a user name, information only for that user is displayed. If no user name is given, information for all users currently logged in to the system is displayed.

NAME

Finger – user information lookup program

SYNOPSIS

finger [-lmsp]

DESCRIPTION

The finger displays information about the system users.

Options are:

-l Produces a multi-line format displaying all of the information described for the –s option as well as the user's home directory, home phone number, login shell, mail status, and the contents of the files ".plan" and ".project" and ".forward" from the user's home directory.

-s Finger displays the user's login name, real name, terminal name and write status (as a "*" after the terminal name if write permission is dinied), idle time, login time, office location and office phone number. Login time is displayed as month, day, hours and minutes, unless more than six months ago, in which case the year is displayed rather than the hours and minutes.

The finger command displays in multi-column format the following information about each logged-in user:

+	user name
+	user's full name
+	terminal name (prepended with a "*" (asterik) if write-permission is denied
+	idle time
+	login time
+	host name, if logged in remotely

Example 1.8.3.1

```
[plokam@Linux1 ~]$ finger plokam
Login: plokam                          Name: Prasad V Lokam
Directory: /home/plokam                Shell: /bin/bash
On since Fri Dec  9 15:31 (EST) on pts/2 from 192.168.5.100
No mail.
No Plan.
[plokam@Linux1 ~]$
```

1.8.4. id

This command displays your user ID, user name, group ID, group name. The system uses this user ID to identify the files you own. The group ID works the same except it is used for the group level identification.

Syntax

id [username]

if no user operand is provided, the id utility will write the user and group IDs and the corresponding user and group names of the invoking process on the terminal.

The following options can be used in id command

-g	displays the output of group id
-u	displays the output of user id

Example 1.7

```
[plokam@Linux1 ~]$ id
uid=500(plokam) gid=500(plokam) groups=500(plokam),501(mirage)
context=user_u:system_r:unconfined_t
```

1.8.5. man

The man command enables you to find information you to find information in the online manuals by specifying a keyword. The manual entry is called a man page, even though it is often more than one page long. There are common sections to man pages. Depending on the command, some or all of the sections may be present. At the start of the man page is the Name. This is usually a one-line that give the command's name along with a pharase describing what it does. Next is the Synopsis which gives the command's syntax including its arguments and options. In the Synopsis, if any argument is enclosed in square brackets ([ ]), then the argument is optional. If two elements of the syntax are separated with a vertical bar (|), then either one or the other (but not both) of the items is allowed. The other sections are Description, Files and See Also.

Example 1.8

```
[plokam@Linux1 ~]$  man pwd
```

1.8.6. Typing more than one command on a line

You can type more than one command on a line by separating the commands with a semicolon. For example, you can find out the current time and also your user ID by typing the dat and id commands on the same line.

Example 1.9

```
[plokam@Linux1 ~]$  date; id
Mon Jan 18 20 : 39 : 56 GMT 1999
uid =102 (plokam) gid =40 (plokam)
```

1.8.7. who & whoami

who' prints information about users who are currently logged on. If given no non-option arguments, 'who' prints the following information for each user currently logged on; login name, terminal line, login time, and remote hostname or X display. The who dommand enables you to find out the users on the systems.

NAME

who – show who is logged on

SYNOPSIS

Who [OPTION] [am I]

DESCRIPTION

-H --heading print line of column headings

-i --add user idle time as HOURS:MINUTES, or old

-q --counts and prints only the login names and the number of users logged on.

Options

q Print only the login names and the number of users logged on.

Print line of column headings.

who

who am i

it displays the username, terminaltype, date and login time

***Example* 1.10**

What is the command that is used to display the list of users logged into the system.

```
[plokam@Linux1 ~]$ who

root         pts/0         may 15   13:37
plokam             pts/1         may 15   16:44
ajay         pts/2         may 15   17:12
oracle       pts/3         may 15   17:12
```

How do you know who is the current user of this session / terminal?

```
[plokam@Linux1 ~]$ whoami
root         pts/0         may 15   13:37
```

```
[[plokam@Linux1 ~]$ who –q
root uc673 uc2007 uc783 uc776 uc782 uc777 uc752 uc670 mohan
proj238 uc658 uc781
uc780 sat uc778 uc 4000 uc030110
# users=18
```

```
[plokam@Linux1 ~]$ who –h

USER         LINE          LOGIN-TIME           FROM
root         tty6          jan      2  03:26
uc673        pts/0         jan      2  10 : 53
uc2007       pts/2         jan      2  09 : 45
uc783        pts/1         jan      2  10 : 47
```

```
uc776      pts/3      jan   2  10 : 45
uc782      pts/4      jan   2  10 : 46
uc777      pts/5      jan   2  10 : 47
uc752      pts/6      jan   2  10 : 53
uc670      pts/9      jan   2  11 : 32
mohan      pts/7      jan   2  10 : 56
```

```
[plokam@Linux1 ~]$ who –i

root       tty6       jan   2  03 : 26      02 : 27
uc673      pts/0      jan   2  10 : 53
uc2007     pts/2      jan   2  09 : 45      00 : 55
uc783      pts/1      jan   2  10 : 47
uc776      pts/3      jan   2  10 : 45      00 : 07
uc782      pts/4      jan   2  10 : 46      00 : 06
uc777      pts/5      jan   2  10 : 47      00 : 13
uc752      pts/6      jan   2  10 : 53
uc670      pts/9      jan   2  11 : 32      00 : 06
mohan      pts/7      jan   2  10 : 56
proj238    pts/10     jan   2  11 : 27
```

echo

Copies the written string to the screen. Exist as internal in all shells.

env

Alters the current environment and invokes a utility or shows the current environment.

export

Exports a shell variable to environment

setenv

Sets an environment variable. With no argument it shows all environment variables.

unsetenv

Removes environment variables.

1.9. System Commands

uname ***displays the operating system name***

```
[plokam@Linux1 ~]$ uname
Linux
```

Logname ***displays the login name of the current user***

```
[plokam@Linux1 ~]$ logname
plokam
```

hostname ***displays the host name of the unix operating system***

```
[plokam@Linux1 ~]$ hostname
Linux1
```

1.10. LAB EXERCISES

Simple Commands

1. Obtain the following results

   ● to print the name of operating system
   ● to print the login name
   ● to print the host name

2. Find out the users who are currently logged in and find the particular user also

3. Display the calendar for

   ● jan 2000
   ● Feb 1999
   ● 9th month of the year 7 A.D
   ● for the current month
   ● Current Date Day Abbreviation, Month Abbreviation along with year

4. Display the time in 12-Hour and 24 Hour Notations.

5. Display the Current Date and Current Time.

6. Display the message “GOOD MORNING” in enlarged characters.

7. What is the command to logout?

8. What is the output of id command?

9. What is the output of finger command?

10. What is the command to view the help of each command?

11. How will you type more than one command on command prompt?

Chapter 2

2. Unix Files and Directories

2.1. OBJECTIVE OF CHAPTER 2

The objective of this chapter is to make you conversant with UNIX directory commands and their usage. You learn how to create directories and manipulate its components. Then you also learn how to work with files. You learn various file manipulation commands and also file printing commands.

In this chapter you will learn the following

- UNIX directory commands
- Working directory and home directory concept
- Create, change, remove directories
- Displaying the directory contents
- Concept of absolute and relative paths
- UNIX meta characters
- Creating and examining files
- Copy, rename, remove files
- Compare file contents
- Print the file contents

2.2. UNIX DIRECTORIES

Directories provide a convenient means of organizing files. Since, in UNIX system, the file system has a hierarchical structure, a directory can further contain sub directories.

2.2.1. Home and Working Directories

Then you log in to the system, you are positioned to a particular directory in the directory hierarchy. This is your home directory. The administrator creates this directory when your user account is created. The system is notified about this directory and whenever you login, you are automatically positioned at your HOME directory.

You are not always stuck at your Home directory. You can freely navigate between directories for which you have access permission. At any point of time, you will be positioned at a particular directory. This directory is referred as the current working directory.

2.2.2. pwd

pwd command displays the absolute path of your present working directory. As discussed earlier, when you login, you are at your HOME directory. To verify this, use pwd command.

Syntax

```
pwd
```

The output from this command verifies that the current working directory is /user/plokam. Since this is your home directory, you are the owner of it and all the files under it. Therefore, you have all the permissions to read, write, create, change, or delete the contents of your home directory.

Example 2.1

```
$ pwd
/user / plokam
```

2.2.3. Absolute and Relative Pathname

In a UNIX file system, a file is identified by its exact location in the directory structure. A path name represents the path to be followed from root through the directory tree to locate a particular file. There are two ways by which a file can be accessed:

Absolute path name

Relative path name

Absolute pathname It is the complete path name from the root that UNIX must follow to reach a particular file. Absolute path name starts with a slash (/). For example, to access of the file **myfile** residing under sub-directory **files** in the home directory, **/usr/plokam,** we have use the notation

```
/usr/plokam/files/myfile
```

The initial slash (/) refers to the root directory. The following slashes separate the names of subdirectories. The final slash denotes the actual file name.

Relative Pathname: In some cases you find that typing absolute pathnames to be tedious as they are very long leading to type. IN such cases you can use a relative pathname.

a pathname that is shortened in relationship to your present directory position.

If your current working directory is /user/plokam, you can also access a particular file relative to your present directory. Relative path name is represented by a dot (.). For example, you wanted to access **myfile**, instead of starting the search from the root, relative referencing starts from the present directory to reach **myfile.**

```
./files/myfile
```

The dot represents the present directory. The following slashes separate the subdirectories and the final slash denotes the actual file name.

As we saw in our previous example, a dot (.) represents the present working directory. The path that follows the dot is relative to the present directory.

Like dot, you can also use double dots (..) if you want to specify the parent directory of the current directory in a relative pathname.

```
../ajay/files/myfile
```

The current working directory is /user/plokam. The above path represents the file myfile under files directory under /user/ajay.

You can use multiple double dots to move up to the parent directory's parent directory and so on. Unless you are logged into the root directory, every directory in UNIX is actually a subdirectory of another directory.

```
../ ../etc/passwd
```

The above relative path from /user/plokam represents /etc/passwd. The first double dot takes you to the parent directory of /user/plokam, which is /user. The second double dot takes you to its parent directory, which is root (/). From there, you navigate down to the etc directory and finally reach the file password.

2.2.4. Cd

Now that you know where you are with the pwd command, let's move ahead. The cd command enables you to change from your present working directory to a new directory.

Syntax

```
cd    [directory]
```

Where **directory** is the name of the directory that you want to change to.

The cd command is a handy tool to move you where you need to be in the file system. Let us assume the present directory is /user/plokam

Example 2.2

```
 $ pwd
/usr/plokam
```

To move to the root directory

```
$ cd   /
```

Here, we are using the cd command to move to another directory and the slash denotes the root directory.

To move to another directory, combine cd command with the path name

```
$ cd  /user/ajay
```

To return to HOME directory

```
$ cd
$pwd
/usr/plokam
```

You can use either absolute or relative pathnames to change directories.

Note:

cd cannot per form the requested directory change if it does not exist.

cd without any argument will always take you to your HOME directory.

cd ../ takes you to the parent (previous directory) i.e. one level up.

cd ../../ takes you to the parent's parent directory i.e. two levels up.

2.2.5. Mkdir

You use mkdir command to create new directories, thus building a hierarchy of directories to maintain your files in an orderly manner. If you have hundreds of files it is always better to organize them in directories based on their related information. For example, all files containing the personal information of the employees can be kept under one directory, say, personal. This way, it becomes easier to locate files if they are categorized in subdirectories.

Syntax

```
mkdir   [  dir-name-1, dir-name-2,…………dir-name-n  ]
```

where pathname is the path of the directory to be created

Example 2.3

To create a new directory under the HOME directory

```
$ mkdir personal
```

From HOME directory, change to personal

```
$ cd personal
$ pwd
/user/plokam/personal
```

From your current directory (personal), create another directory

```
$ mkdir salary
$ cd salary
$ pwd
/usr/plokam/personal/salary
```

In this way, you can create directories within directories. When you create a new directory, the system builds a new inode (information node) for the directory. This allows access to the data stored in the directory file and informs the system that the file is a directory type file. The directory is a preformatted file, containing the filename and its related inode number.

Note: mkdir command only creates a new directory. It does not change your current directory. To make this directory as the current directory, you have to use the cd command. More than one directory can be created at a time.

2.2.6. ls

ls command can be used to display the names of files and directories. You use this utility to know what files and subdirectories exist in your directories. Different options can be used with the ls command to list the contents in different formats.

```
$ pwd
$/usr/plokam
$  ls
example1      example4      example7      files     typescript
example2      example5      example8      login
example3      example6      example9      personal
```

Syntax

ls [options]

The following options can be used with ls command

-a List all files, including the hidden file

-C,x Multi-column output with files sorted down the in column wise

-F Put a slash (/) after each filename if the file is a directory, and asterisk (*) if the file is an executable, and an at-sign (@) if the file is a symbolic link.

-i For each file, print the i-node number in the first column of the report.

-l List in long format, giving mode, ACL indication, number of links, owner, group, size in bytes, and time of last modification for each file etc.

-R Recursively list subdirectories encountered.

-t Sort by the stamp (latest first) instead of by name. The default is the last modification time.

Example 2.4

```
$  ls  -FC
```

```
example1     example4     example7     files/       typescript
example2     example5     example8     login
example3     example6     example9     personal/
```

The complete list of files and directories is displayed by using:

```
$ ls -il
```

```
total 132
75372 -rw --r-- 1 plokam          dba       5922 May 15 15:09 example1
75404 -rw --r-- 1 plokam          plokam    6661 May 17 11:35 example2
75405 -rw --r-- 1 plokam          plokam    6661 May 17 11:35 example3
75406 -rw --r-- 1 plokam          plokam    6661 May 17 11:35 example4
75407 -rw --r-- 1 plokam          plokam    6661 May 17 11:35 example5
75408 -rw --r-- 1 plokam          plokam    6661 May 17 11:35 example6
75409 -rw --r-- 1 plokam          plokam    6661 May 17 11:35 example7
75410 -rw --r-- 1 plokam          plokam    6661 May 17 11:35 example8
75411 -rw --r-- 1 plokam          plokam    6661 May 17 11:35 example9
37732 drwxr-xr-x      2 plokam          plokam        512     May   17   16:47
files
75370 -rw --r-- 1 plokam          dba       304  May 15 14:07 login
37880 drwxr-xr-x      3 plokam          plokam        512     May   17   11:38
personal
75371 -rw --r-- 1 plokam          plokam    43   May 17 11:36 typescript
```

The information displayed is as follows:

Total **132** implies the total number of blocks occupied by the files listed implies that it is a directory file.

Column 1 refers to the inode

-rw-r--r—refers to the various access permissions (refer chmod command)

column 3 specifies the number of links a file has

column 4 gives the name of the user/owner

column 5 gives the name of the group to which the user belongs

column 6 gives the size of file in bytes

column 7&8 give the date and time of last modification

column 9 refers to the name of the file

To list the subdirectories within the directories, -R option can be used with the ls command. It will recursively list all your directories.

```
          $  ls  -R

          . :
          example1      example4      example7      files         personal
          example2      example5      example8      fruits        typescript
          example3      example6      example9      login

          . /files:
          myfile

          . /fruits:
          apples     oranges
. /fruits/apples:

          . /fruits/orange:

          .  /personal:
          salary

          . /personal/salary:

$ ls –x

Main.dt Personal.dt    a             arth.c        bin
com.m.1     comm.1 comm.2 d             dead.letter
demo.c      demol.c demo2.c dev         e
error       etc           etcshadow     f             for
g           hash          hh            ibin          if.c
initial.c initial.c l     lib           lost+found
matric.c    mbox          mnt           num           opt

$  ls –a

.
..
.NetWare
.Xauthority
.lastlogin
```

```
.mailrc
.mosaic-global-history
.mosaic-hotlist-default
.mosaic-personal-annotations
.mosaicpid
.netscape
.odtpref
.profile
.scoadmin.pref
.scohelp-global-history
```

Note: ls command lists its files in an alphabetical order

.

2.2.7. rmdir

Just as you can easily create directory-using **mkdir,** so can you easily remove one with the **rmdir** command, when it is no longer needed. The one thing required before removing a directory is that it should not contain any files i.e. it should be empty. If there are files in the directory when **rmdir** is executed, then you will not be allowed to remove the directory.

```
$ cd /user/plokam/fruits/
$ rmdir apples
```

The directory apple is removed.

Syntax

```
rmdir [ [options] dir-name ]
```

Note: Before removing a directory there should be no files in the particular directory [to be removed]. Change it to the corresponding parent directory and then only we are able to delete the files

Example 2.5

Assume the current directory is **fruits.** To remove a directory oranges from the fruits directory

```
$ ls oranges
$  ttl
$
$  rmdir oranges
rmdir: directory “oranges”:  Directory not empty
```

So first you have to remove all the files from the directory to be removed, and then carry on with the removal of that directory.

Note: The directory to be removed must be empty. The empty directory has two entries, the . (dot) representing itself and the ..(dot dot) representing the parent directory. More than one directory can be removed at a time.

You can delete the directory using rm –r command

```
rm –r  <directory name>
```

Note: Before removing the directory make sure that the present working directory should be parent directory

Example 2.6

To remove the directory example which is under the /usr/plokam $ pwd

```
/usr/plokam/example
$ cd..
$ pwd
/usr/plokam
$ rm r example
$ pwd
/usr/plokam
$ cd example
example:  does not exist
```

2.3. FILENAME EXPANSION

One powerful feature of the UNIX system is *filename expansion.* It enables you to work with files collectively. The group of filename-matching characters is represented by wildcards. Wildcards allow you to manipulate multiple files at one time, with a single command. Wildcards are a kind of shorthand that allows you to specify similar files without having to type multiple names.

There are three types of UNIX wildcards: ***, ?, [].** The shell expands these wildcards, substitutes a group of valid filenames and these filenames are then given to the respective commands.

*	matches zero or more characters
?	matches exactly one character
[]	matches any one of the characters in the given range

2.3.1. The Asterisk (*)

The * is interpreted as a set of zero or more characters. It can be used in a number of different ways. It provides an easier and quicker way to search directories and access files. For example,

Example 2.7

List all files which begin with a letter 'a'

```
$ .ls              a*
acs                ajay.txt
```

search all files ending with **.c** in the current directory and display them on the screen

```
$ find . –name '*.c' -print
./foo.c
./fool.c
```

display the contents of every file starting with **'b'** and ending with **'t'** i.e. the filename can have any number of characters inside **'b'** and **'t'.** The examples can be filenames like bt, bat, brat, bit, braaaat, and so on.

```
$ cat b*t
```

2.3.2. The Question mark (?)

The **?** is interpreted as a single character. It can be used to match only one occurrence of the character.

Example 2.8

List all files with **'exam'** as the first two characters followed by any single character.

```
$ ls exam?
```

exam1 exam2

Display the contents of files like chap, chip, chop, chup, chep chkp, and so on. The point is the **?** is replaced by exactly one character.

2.3.3. The Rang specifier []

The [ ] can be used to specify a range of characters which matches either occurrence of the character.

Example 2.9

Lists the files starting with any one of the alphabets and ending in **'t'**

```
$ ls [a-z] *t
```

Displays the files chap1, chap2 or chap3

```
$ cat chap [123]
```

2.4. WORKING WITH FILES

The UNIX system provides many tools that enable you to work easily with files. Among these tools are the commands that enable you to create new files, copy files, removes files, move files between directories, examine the contents of the files, and so on. In this session we will learn to use some of these commands.

2.4.1. cat

The **cat** command can be used for the following purposes:

To create a new file.

To display the contents of an already existing file.

Syntax

```
Cat [ filename 1, filename 2,………filename n ]
```

Where **filename** is the name of the file to be created or to be displayed.

To create a new file **newfile,** type the following on your terminal:

Example 2.10

```
$ cat > newfile
This is my new file created by cat command
^D
```

Pressing ctrl-d marks the end to file. Now that you have created a file you can use **ls** command to verify that the does exist

```
$ ls
```

```
exam1  example2      example5      example8      frruits   personal
exam2  example3      example6      example9      login     typescript
example1      example4      example7      files            newfile
```

To view the contents of the file **newfile**

```
$ cat newfile
This is my new file created by cat command
```

Note the difference between two Commands. One is used to create a file and the other to display its contents.

Redirection symbols are used along with cat command

```
>       is used to create a file
>>      is used to append without overwrite the information
is used to input the file to commands
```

cat command can be used to display the contents of more than one file by giving the filenames separated by a space. To view the contents of two files **file1** and **file 2**

```
        $ cat > file1
        This is my first file.
I am learning how to create files using
        cat command., It can also be used to view
the contents of the file.
^D

$ cat > file2
```

```
This is my second file.
Cat command can also be used to display
The contents of two or more files at a time.
^D

$ cat file1 file2
This is my first file.
I am learning how to create files using
cat command.  It can also be used to view
the contents of the file.
This is my second file.
Cat command can also be used to display
the contents of two or more files at a time.
```

The cat command without a filename takes the input from standard input device (keyboard) and writes it to the standard output device (the terminal). The effect is that each line typed is echoed back, and to end this press *ctrl-d.*

```
$ cat
cat command without a file name
cat command without a file name
It is echoed on the terminal
It is echoed on the terminal
The first line is what you type
The first line is what you type
The second line is what that gets echoed
The second line is what that gets echoed
^D
$ cat > sample
Welcome to Unix Environment
I am The Crazy Man
^D
```

To view the file

```
$ cat sample
Welcome to Unix Environment
I am The Crazy Man
```

To append the file

```
$ cat >> sample
```

```
I am The Crazy Man

$ cat sample
Welcome to Unix Environment
I am The Crazy Man
I am The Crazy Man
^D
```

Note: CtrlD marks the logical end of file. To finish writing to a file always press Ctrl d on the beginning of a new line. If you try to create a file with a filename that already exist, the contents of the existing file will be overwritten.

2.4.2. cp

You may want to make a copy of a file for backup purposes, or you may want to use an existing file as the basis for a new document. If you accidentally erase the original file, you can restore the file from the last backup copy you made. These tasks are accomplished with the cp command.

Syantax

```
cp file1 file2
```

where **file1** is the source file
file2 is the target file

```
cp file1 file2 file3 … .d1
```

where **file1, file2, file3**… are the source files
d1 is the target directory

```
cp –R s1 d1
```

where **s1** is the source directory
d1 is the target directory
-R copy all files and subdirectories

Example 2.11

List all the files in your present directory (assume it to be /user/plokam)

```
        $  ls

    exam1       example3      example7      file2        newfile
    exam2       example4      example8      files        personal
example1        example5      example9      fruits       typescript
example2        example6      file1         login
```

Copy newfile to **file3.** This will cause the file named **newfile** to be copied into the file named **file3.** The **$** sign after the cp command indicates that the command was executed successfully. To verify that a new file **(file3)** was actually created, you can issue the **ls** command at the **$** prompt. You can examine the contents of **file3** using the **cat** command.

```
$ cp newfile file3
$
$ ls

exam1           example3      example7      file2        login
exam2           example4      example8      file3        newfile
example1        example5      examople9     files        personal
example2        example6      file1         fruits       typescript

$ cat file3
This is my new file created by cat command
```

To copy files **file1,file2,file3** to the directory **personal**

```
$ cp file1 file2 file3 personal
$ ls personal
file1    file2    file3    salary
```

Note: If the target file is an ordinary file and it already exists its contents are erased and are overwritten with the contents of the source file. If the target file is a directory the source file is copied to that directory with the same name as the source. With one cp command only one file can be copied to another file but one or more files can be copied to a directory at the same time

.

2.4.3. mv

There are many instances where you may want to move a file from one directory to another: You may want to move older files to an archival directory, or you may want to move previous versions of existing files, and so on. The mv command allows for the movement of files in UNIX. **mv** command can be used for three purposes:

To rename a file with a new name.

To move one or more files to a different directory.

Rename a directory with a new directory name.

Syntax

mv file1 file2

where **file1** is the old filename

file2 is the new name

mv **file1 file2** … .d1

where **file1, file2**, … . are the filenames to be moved

d1 is the destination directory

mv d1 d2

where **d1** is the old directory name

d2 is the new directory name

Example 2.12

```
$ mv file1 myfile
$ ls

exam1       example3    example7    file3       myfile
exam2       example4    example8    files       newfile
example1    example5    examople9   fruits      personal
example2    example6    file2       login       typescript
$
```

move files **file1, file2** to directory **newpersonal**

```
$ mkdir newpersonal
$ mv file2 file3 newpersonal
```

rename directory **fruits** to directory **foods**. You can verify the changes by using ls command

```
$ mv fruits foods
```

Note: The new and the old files may have the same names provided they reside in different directories. If the new filename is an existing file its contents are lost.

2.4.4. rm

Unwanted files can clog up a hard disk, slowing it down and making your file management chores unnecessarily complicated. It is a good practise to regularly go through your subdirectories and remove the file that are not required. **rm** command lets you delete the files which are no longer needed.

Syntax

rm [option] (filename)

where **filename** *is the name of the file to be removed*

option *can be any or combination of the following:*

- f	*forcefully remove a file even if it is write-protected*
- i	*interactively asks for confirming the deletion of the files*
- r	*removes a directory even if it is not empty*

Example 2.13

Remove **file2**

```
$ rm newpersonal/file2
```

Remove the directory foods with its files. **rm –i** asks for a confirmation whether the user wants to delete the file. If the answer is 'y' (yes), the command moves ahead in performing the operation. If the answer is 'n' (no), the command cancels the deletion operation. It is a safe way of removing files.

```
$ rm –i   myfile
rm:  remove myfile (yes/no ?) y
```

Note: It is always safer to remove files using rm –i option. It is not advisable to use rm –r option for removing directories

.

2.4.5. ln

`ln` command is used to create one or more links to a file. A link is another name for the same filename, having the same physical storage and same inode number. This means that more than one file can point to the same physical storage.

Syntax

```
$ ln filename1 filename2
```

Where **filename1** *is the name of the file on which a link is being created*

filename2 *is the link created on filename1.*

Example 2.14

```
$ ln newfile file1
```

will create a link for newfile and the name of the link is file1. The number of links a file has can be seen from the long listing of files.

```
$ ls  -li
total  146
75417 –rw—r  --1  plokam            plokam        1 May 17 12:11 exam1
75418 –rw—r  --1  plokam            plokam        1 May 17 12:12 exam2
75372 –rw—r  --1  plokam            dba           5922 May 15 15:09 example1
75404 –rw—r  --1  plokam            plokam        8701 May 17 12:15 example2
75405 –rw—r  --1  plokam            plokam        6661 May 17 11:35 example3
75406 –rw—r  --1  plokam            plokam        6661 May 17 11:35 example4
75407 –rw—r  --1  plokam            plokam        6661 May 17 11:35 example5
75408 –rw—r  --1  plokam            plokam        6661 May 17 11:35 example6
75409 –rw—r  --1  plokam            plokam        6661 May 17 11:35 example7
75410 –rw—r  --1  plokam            plokam        6661 May 17 11:35 example8
75411 –rw—r  --1  plokam            plokam        6661 May 17 11:35 example9
75419 –rw—r  --2  plokam            plokam        44 May 17 12:16 file1
37732 drwxr-xr-x        2  plokam            plokam        512 May 15 16:47 files
75370 –rw—r  --1 plokam             dba           304 May 15 14:07 login
75419 –rw—r  --2 plokam             plokam        44 May 17 12:16 newfile
18961 drwxr-xr-x        2 plokam             plokam        512   May   17   12:21
newpersonal
37880 drwxr-xr-x        3 plokam             plokam        512    May   17   12:19
personal
75371 –rw—r  --1 plokam             plokam        43 May 17 12:16 typescript
```

Note, both the files are having same inode numbers.

We also saw in the previous chapter, that UNIX supports another type of link known as symbolic link, which can link files from other systems. For creating symbolic link, use the –s option to the ln command:

Example 2.15

```
$ ln  -s symbolic /tmp/syml
$ ls  -il /tmp/syml
```

8 lrwxrwxrwx 1 plokam plokam 8 May 17 12:52 /tmp/syml -> symbolic

Note: In command gives more than one name to a file but the physical storage of the file is the Same. Whereas *cp* command makes two files with different physical torage. Inode numbers of all hard linked files are same. You can create a symbolic link even when the file on which the symbolic link is created is not present.

2.5. COMPARING FILES

Normally, we create many new files during the course of updating older files. We maintain many versions of the same file with slight differences in the content. At some point of time, we find it very difficult to keep track of the files and we would like to cleanup the directories removing the unwanted older versions.

There are utilities in UNIX that can be used to compare the contents of two files to see if they are same. If they are not, the nature of the difference can be determined.

2.5.1. Cmp

Cmp command is used to compare two files. It compares two files and displays the first instance where the files differ. And if there is no difference, cmp returns no output.

Syntax

```
cmp file1 file2
```

Where file1 and file2 are the files to be compared.

Example 2.16

```
$ cat > myfile
```

This is a demonstration of cmp command

^D

```
$ cat > yourfile
This is the demonstrations of cmp command
^D
```

```
$ cmp myfile yourfile
myfile yourfile differ: char 9, line1
```

cmp reports that the two files differ in the nineth character of the document, located in line 1.

```
$ cmp myfile myfile
```

No output is reported since both the files are identical. Using cmp is the simplest and quickest way to compare two files, but you only know if the files are different.

Note: cmp indicates only the first character at which the files differ. Cmp reports the difference on a character-by-character basis.

2.5.2. diff

Using cmp you only get to know if the files compared are different. You do not know to what extent the files are different, nor do you know how they are different. The diff command compares two files for differences. It determines which lines must be changed to make the two files identical. The diff command scans two files and indicates editing changes that must be made to the first file to make it identical to the second file. Editing may include adding a line, deleting a line, changing a line, and so on.

Syntax

```
diff file1  file2
```

Where file1 and file2 are the files to be compared.

Example 2.17

* Consider two files – file1 and file2:

```
$ cat file2
diff
login
kill
mv
ln
more
pg
```

```
^D
$  cat file1
diff
id
sh
mv
ln
^D
$  diff file1 file2
2, 3c2, 3
<  id
<  sh
---
> login
>kill
5a6,7
> more
> pg
```

Lines beginning with < are found only in the first file and line beginning with > are found only in the second file. The dashed line separates the two lines that appear in the same place in the differing files. The numerals indicate exactly where and how the differences occur.

In the above output, the second and the third lines from file 1 have to be changed with second and third lines of file2 to make the two files agree. This is indicated by 2,3c2,3.

Similarly, 5a6,7 indicates that the lines 6 and 7 of file2 needs to be appended to file1 to make them equal.

Likewise sign 'd' can used to delete lines from file1 if these lines do not exist in file2.

Note: diff command makes changes in the first file to make it resemble second file. Diff command is used to compare files on line to line basis.

2.6. PRINTING FILES

In certain cases, you may required to take a hard copy of the documents that you have created and that are being stored in the hard disk. This requires the document to be printed on a sheet of paper. For this, a printer is to be attached to the system. The system should request the printer to print the required document. It is the task of the system administrator to setup the printer, which is a complex one. As an end user, it is relatively simple for you to print a document in UNIX. It just requires that you are familiar with three print commands: lp, lpstat and cancel.

2.6.1. lp

lp spools the output to the printer queue. Spooling is the sending of output to a temporary storage area for later processing by the printer.

Syntax

```
lp  filename
```

where **filename** *is the name of the document to be printed.*

The print jobs are placed in a print queue as printer requests. The request Ids consist of destination printer name and a sequence number. For example, if you send your file to the printer 'hplj' for printing and the last request Id was 15, then your request Id would be hplj-15.

Example 2.18

```
$  lp myfile
request id is hplj-15  (1 file)
```

2.6.2. cancel

Cancel command removes or cancels print requests made with lp command.

Syntax

```
cancel request_id
```

where **request_id** *is the ID number given by the system when you placed a lp request*.

Example 2.19

To cancel the print request that you had given to the printer.

```
$  cancel hplj-15
```

To cancel all queued requests that you have queued up in the printer hplj, use the following command

```
$ cancel hplj
```

2.6.3. lpstat

The print requests are spooled and they may not be performed immediately. It may so happen that the printer is down due to printer out of paper or a paper jam occurred. You may want to know the status of the printer, how many request of what size are there before you request etc. In such cases, you may use the lpstat command to display the current status of all line printers.

Syntax

```
lpstat  [options]
```

Example 2.20

```
$lpstat
hplj-14          root     1524     May  10  17:34
hplj-15          plokam   1024     May  11  11:10
```

lpstat indicates that there are two requests in the print queue. The first request is with ID hplj-14 initated by root of size 1524 blocks on May 10th 17:34 Hrs and the second request is with ID hplj-15 iniated by plokam of size 1024 block on May 11th at 11:10 Hrs.

2.7. LAB EXERCISES

Directory Commands

1. Display the name of your home directory
2. Create a directory SAMPLE under your home directory
3. Create a sub-directory by name TRIAL under SAMPLE
4. Change to SAMPLE
5. Change to your home directory
6. Using absolute and relative pathname, change from home directory to TRIAL
7. Remove directory TRAIL
8. Create a directory TEST using absolute pathname.
9. Using a single command change from current directory to home directory
10. Remove a directory using absolute pathname.
11. What is the command that is used for displayng the list of all files in a directory sorted by last modified time stamp.

Chapter 3

3. UNIX : System Administration Commands

3.1. OBJECTIVE OF CHAPTER 3

The objective of this chapter is to introduce you to some of the most important commands that are needed in order to perform some of the system administration tasks. This involves various commands like tar, gzip, gunzip and others which are used for creating archives / compress and decompress files. Latter we will try to see as to how we can start, stop and restart services. Then We will look at xinetd and inetd related information. How to install and configure ftp server and telnet server.

In this chapter you will learn the following:

- ❖ Create and Manage Archives
- ❖ Compress & Uncompress archives & Files
- ❖ Start & Stop Services
- ❖ Network configuration commands
- ❖ Install & Setup ftp and telnet servers
- ❖ Configure Inetd and xinetd deamon services
- ❖ Mount & unmount file systems
- ❖ Disk Space Administration & Monitoring
- ❖ Batch Job Scheduling

3.2. Create & Manage Archives

An operating system is an important part of a computer system. A computer system can be described as a collection of hardware and software components.

3.2.1. tar

Displays the current date and current time of the system.

Sysntax

To create a tar image use the following command

```
tar –cvf  [archiveFilename.tar] [list of files]
```

To Display the contents of a tar image use the following command

```
tar –tvf  [archiveFilename.tar]
```

To Extract the contents of a tar image use the following command

```
tar –xvf  [archiveFilename.tar] [list of files to be extracted]
```

Example 2.2.1

The Following set of commands are used to make an archive of all the files that exists in the /etc directory with the host* pattern of filenames and then creates an archive called ***hosts.tar***. Then It uses that file to display the contents of the tar file and latter extracts the contents to a local directory.

```
[plokam@Linux1 ~]$ cd ~
[plokam@Linux1 ~]$ tar -cvf hosts.tar /etc/hosts*
tar: Removing leading `/' from member names
/etc/hosts
/etc/hosts~
/etc/hosts.allow
/etc/hosts.bak
/etc/hosts.deny

[plokam@Linux1 ~]$tar  -tvf hosts.tar
-rw-r--r-- root/root       195 2005-11-29 16:35:04 etc/hosts
-rw-r--r-- root/root       197 2005-11-29 16:05:25 etc/hosts~
-rw-r--r-- root/root       191 2005-10-18 19:14:16 etc/hosts.allow
-rw-r--r-- root/root       197 2005-11-29 16:05:48 etc/hosts.bak
-rw-r--r-- root/root       347 2005-10-18 19:14:55 etc/hosts.deny

[plokam@Linux1 ~]$ tar -xvf hosts.tar
etc/hosts
etc/hosts~
```

```
etc/hosts.allow
etc/hosts.bak
etc/hosts.deny
[plokam@Linux1 ~]$
```

3.2.2. gzip

gzip (GNU zip) is a compression utility designed to be a replacement for *compress. g*zip reduces the size of the named files using Lempel-Ziv coding (LZ77). Whenever possible, each file is replaced by one with the extension .gz, while keeping the same ownership modes, access and modification times.

Syntax:

gzip –c [filename]

Example:

Compressing a file:

gzip -c test.txt > test.gz

In the above example command this would compress the test.txt file as test.gz in the current directory.

3.2.3. gunzip

gzip produces files with a `.gz` extension. *gunzip* can decompress files created by *gzip*

Syntax:

gunzip –c [filename]

Example:

Expand:

gunzip -c test.gz

In the above example command you would be able to see the contents of the test.gz file. If you wanted to extract the contents of that file into another file you would use the below command to extract the contents into the test.txt file.

gunzip -c test.gz > test.txt

3.2.4. zcat

zcat takes one or more compressed data files as input. zcat decompresses the data of all the input files, and writes the result on the standard output. zcat concatenates the data in the same way cat does. The names of compressed input files are expected to end in .Z, .gz, or .bz2

Syntax:

zcat –c [filename]

to display the content of the given compressed file

Example :

zcat –c test.gz

Displays the content of test.gz when uncompressed.

3.2.5. compress

Used for Compressing the contents of a File. compress compresses each input file using Lempel-Ziv compression techniques. If you do not specify any input files, compress reads data from the standard input and writes the compressed result to the standard output.

Syntax

```
compress -c [filename]
```

Example:

```
compress –c test.txt > test.gz
```

3.2.6. uncompress

Displays the current date and current time of the system.

Sysntax

```
uncompress -c [filename]
```

Example:

```
uncompress –c test.gz > test.txt
```

3.3. Services & Daemons

Once you have successfully logged into a UNIX system, you can start using the system by issuing commands to it.

3.3.1. inetd

The /usr/sbin/inetd daemon provides Internet service management for a network. This daemon reduces system load by invoking other daemons only when they are needed and by providing several simple Internet services internally without invoking other daemons.

Syntax

```
/usr/sbin/inetd [ -d ] [ -t secondstowait ] [ configurationfile ]
```

3.3.2. xinetd

xinetd - the extended Internet services daemon.

xinetd performs the same function as **inetd**: it starts programs that provide Internet services. Instead of having such servers started at system initialization time, and be dormant until a connection request arrives, **xinetd**s the only daemon process started and it listens on all service ports for the services listed in its configuration file. When a request comes in, **xinetd** starts the appropriate server. Because of the way it operates, **xinetd**(as well as **inetd**) is also referred to as a super-server.

❖ **xinetd Configuration Files**

The configuration files for xinetd are as follows:

/etc/xinetd.conf — The global xinetd configuration file.

/etc/xinetd.d/ directory — The directory containing all service-specific files.

❖ ***The /etc/xinetd.conf File***

The /etc/xinetd.conf contains general configuration settings which effect every service under xinetd's control. It is read once when the xinetd service is started, so in order for configuration changes to take effect, the administrator must restart the xinetd service. Below is a sample /etc/xinetd.conf file:

```
defaults
{
      instances            = 60
      log_type             = SYSLOG authpriv
      log_on_success       = HOST PID
      log_on_failure       = HOST
      cps                  = 25 30
}
includedir /etc/xinetd.d
```

These lines control various aspects of xinetd:

1. instances — Sets the maximum number of requests xinetd can handle at once.
2. log_type — Configures xinetd to use the authpriv log facility, which writes log entries to the /var/log/secure file. Adding a directive such as FILE /var/log/xinetdlog here would create a custom log file called xinetdlog in the /var/log/ directory.
3. log_on_success — Configures xinetd to log if the connection is successful. By default, the remote host's IP address and the process ID of server processing the request are recorded.
4. log_on_failure — Configures xinetd to log if there is a connection failure or if the connection is not allowed.
5. cps — Configures xinetd to allow no more than 25 connections per second to any given service. If this limit is reached, the service is retired for 30 seconds.
6. includedir /etc/xinetd.d/ — Includes options declared in the service-specific configuration files located in the /etc/xinetd.d/ directory.

Note:
Often, both the log_on_success and log_on_failure settings in /etc/xinetd.conf are further modified in the service-specific log files. For this reason, more information may appear in a given service's log than this file may indicate.

❖ ***The /etc/xinetd.d/ Directory***

The files in the /etc/xinetd.d/ directory contains the configuration files for each service managed by xinetd and the names of the files correlate to the service. As with xinetd.conf, this file is read only when the xinetd service is started. In order for any changes to take effect, the administrator must restart the xinetd service.

The format of files in the /etc/xinetd.d/ directory use the same conventions as /etc/xinetd.conf. The primary reason the configuration for each service is stored in separate file is to make customization easier and less likely to effect other services.

To get an idea of how these files are structured, consider the /etc/xinetd.d/telnet file:

```
service telnet
{
      flags           = REUSE
      socket_type     = stream
      wait            = no
      user            = root
      server          = /usr/sbin/in.telnetd
      log_on_failure  += USERID
      disable         = yes
}
```

These lines control various aspects of the telnet service:

1. service — Defines the service name, usually to match a service listed in the /etc/services file.
2. flags — Sets any of a number of attributes for the connection. REUSE instructs xinetd to reuse the socket for a Telnet connection.
3. socket_type — Sets the network socket type tostream .
4. wait — Defines whether the service is single-threaded (yes) or multi-threaded (no).
5. user — Defines what user ID the process process will run under.
6. server — Defines the binary executable to be launched.
7. log_on_failure — Defines logging parameters for log_on_failure in addition to those already defined in xinetd.conf.
8. disable — Defines whether or not the service is active.

3.3.3. service

Red Hat includes the checkconfig & service utilities to help you manage your start up scripts and save you a lot of typing. This is handy when you're adding your own services and also in managing the already existing services. chkconfig is available if you want to use it on other distributions that may not come with it - just go to freshmeat.net and look it up. /sbin/service is just a shell script that comes as part of Red Hat's initscripts package.

Service is a resource provided to network clients; often provided by more than one server (for example, remote file service).

We will have all the daemons running as services in an operating system as services.in order to start or stop a service we can us the command ***service***.

Syntax:

```
service < option > | --status-all | [ service_name [ command | --full-restart ] ]
```

Example :

we can us the following syntax to start a service vsftpd.as we all know vsftpd is the most famous FTP daemon used in UNIX. let us see how to start if was not started yet.

service vsftpd start

This would start the vsftpd daemon.

Similarly if we want to stop this daemon we can say :

service vsftpd stop

If we want to take a look at all the services running

service --statusall

3.3.4. chkconfig

chkconfig - updates and queries runlevel information for system services

Syntax:

chkconfig --list [name]

chkconfig --add <name>

chkconfig --del <name>

chkconfig [--level <levels>] <name> <on|off|reset>

Example:

Without a tool like chkconfig, symbolic links to the scripts in /etc/rc.d/init.d are typically created by hand at the appropriate run levels. This can be messy & difficult to standardize. Also, it is necessary to view the contents of each run level directory to see which services are configured to run. Here's some ways to use chkconfig: What's enabled at run level 3?

```
chkconfig --list | grep  3:on
```

```
acpid       0:off  1:off  2:off  3:on   4:on   5:on   6:off
anacron       0:off  1:off  2:on   3:on   4:on   5:on   6:off
apmd         0:off  1:off  2:on   3:on   4:on   5:on   6:off
atd         0:off  1:off  2:off  3:on   4:on   5:on   6:off
auditd       0:off  1:off  2:on   3:on   4:on   5:on   6:off
autofs       0:off  1:off  2:off  3:on   4:on   5:on   6:off
bluetooth     0:off  1:off  2:on   3:on   4:on   5:off  6:off
cpuspeed      0:off  1:on   2:on   3:on   4:on   5:on   6:off
crond        0:off  1:off  2:on   3:on   4:on   5:on   6:off
cups        0:off  1:off  2:on   3:on   4:on   5:on   6:off
```

Note: not all output is included.

3.4. Network Management

An operating system is an important part of a computer system. A computer system can be described as a collection of hardware and software components.

3.4.1. hostname

Displays the hostname of the machine.

Syntax

hostname

To display the host name of the Machine

Example 1.2

```
[plokam@Linux1 ~]$ hostname
Linux1
```

3.4.2. ifconfig

Configures network interface parameters or show the status for network interface.

Sysntax

ifconfig [device name]

Example:

```
[plokam@Linux1 ~]$ /sbin/ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:B0:D0:D0:E9:EC
          inet addr:192.168.5.6  Bcast:192.168.5.255  Mask:255.255.255.0
          inet6 addr: fe80::2b0:d0ff:fed0:e9ec/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:889742 errors:0 dropped:0 overruns:0 frame:0
          TX packets:56223 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:66278582 (63.2 MiB)  TX bytes:10896198 (10.3 MiB)
```

3.4.3. ping

Sends ICMP echo requests to specified hosts. The remote host will, if it can, respond with a ICMP echo reply.

Syntax:

Ping hostname

Output:

```
[[plokam@Linux1 ~]$  ping 172.17.15.10
PING 172.17.15.10 (172.17.15.10) 56(84) bytes of data.
64 bytes from 172.17.15.10: icmp_seq=0 ttl=64 time=0.220 ms
```

```
64 bytes from 172.17.15.10: icmp_seq=1 ttl=64 time=0.216 ms
64 bytes from 172.17.15.10: icmp_seq=2 ttl=64 time=0.210 ms
64 bytes from 172.17.15.10: icmp_seq=3 ttl=64 time=0.213 ms
64 bytes from 172.17.15.10: icmp_seq=4 ttl=64 time=0.243 ms
```

Example:

This command is used to send 4 packets to the designated host called linux1.

```
[plokam@Linux1 ~]$ ping -c4 linux1
PING localhost.localdomain (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=0 ttl=64 time=0.095 ms
64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=1 ttl=64 time=0.061 ms
64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=2 ttl=64 time=0.065 ms
64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=3 ttl=64 time=0.036 ms
--- localhost.localdomain ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3002ms
rtt min/avg/max/mdev = 0.036/0.064/0.095/0.021 ms, pipe 2
[plokam@Linux1 ~]$
```

This command is used to send infinite packets to the designated host at ip address 192.168.5.6. This command keeps on pinging until the <Control – C> Character is pressed on the keyboard and then summarized how many packets have been sent and then displays the final statistics.

```
[plokam@Linux1 ~]$ ping 192.168.5.6
PING 192.168.5.6 (192.168.5.6) 56(84) bytes of data.
64 bytes from 192.168.5.6: icmp_seq=0 ttl=64 time=0.107 ms
64 bytes from 192.168.5.6: icmp_seq=1 ttl=64 time=0.088 ms
64 bytes from 192.168.5.6: icmp_seq=2 ttl=64 time=0.067 ms
64 bytes from 192.168.5.6: icmp_seq=3 ttl=64 time=0.037 ms
64 bytes from 192.168.5.6: icmp_seq=4 ttl=64 time=0.070 ms
64 bytes from 192.168.5.6: icmp_seq=5 ttl=64 time=0.035 ms

--- 192.168.5.6 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 4998ms
rtt min/avg/max/mdev = 0.035/0.067/0.107/0.026 ms, pipe 2
[plokam@Linux1~]$
```

Note: *Most of the network commands are present in /sbin directoiry. So make sure that you path is configured properly.*

host

Finds host names for IP addresses on the Internet. The host name information comes from DNS servers.

iptrace

Traces incoming and outgoing IP packets.

netstat

Shows statistics and other network information, based on options selected.

nslookup

Contacts and sends queries to DNS servers interactively or non-interactively.

route

Manages and shows the routing tables on the host.

arp

Shows and alters the ARP table which is used to map MAC addresses to their assigned IP addresses

3.4.4. nfs & nis

The NFS (Network File System) program enables you to treat files on another computer in more or less the same way you treat files on your own computer. This is similar to the file sharing option on windows.

NIS (Network Information System) is a distributed database used to manage a network of computers. This is similar t o the Microsoft's active directory services

3.5. Batch Jobs Scheduling

An operating system is an important part of a computer system. A computer system can be described as a collection of hardware and software components.

3.5.1. cron

Actually it is called 'cron daemon'. Cron is an automatic task machine. You will use it on your Unix or Linux operating systems for doing some tasks at specific intervals with out your intervention every time. You set the clock and forget. The cron daemon runs the work for you

What is cron tab?

'Cron tab(CRON TABle)' is a text file that contains a series of cron functions.

What cron will do for you?

1. If you want to send your email cources to your subscribers at 11.30 night, you will set the cron job on your server.And your cron manager sends the one email every day at 11.30 until all the emails will be finished.
2. If you want to send them on Sundays, you can schedule it with your cron.
3. You can schedule it to delete your website members with expired accounts.
4. You can schedule it to recieve an update on your subscribers from your mailing list manager.
5. You can check your links on other websites in link exchange programms.

3.5.2. at

at executes commands at a specified time.

3.6. Remote Execution Commands:

Some of the important files that we need to be aware of in terms of unix system administration are as

rcp

Copies files between two computers. The computer can be local or remote.

rdist

Distributes files from one computer to others. Will keep the owner, group, mode and modification times.

rexec

Runs commands on a remote host.

rlogin

Runs a remote login to a remote computer.

rsh, remsh

Is a remote shell which is used to connect to a host and execute one specified command.

3.7. Disk / File Management Commands

Some of the important files that we need to be aware of in terms of unix system administration are as

chfs

Changes specified attributes of a file system.

df

Shows used and free disk space for all file system or the one specified.

du

Shows disk usage for a directory and its subdirectories.

/usr/sbin/exportfs

Translates exportfs options to share/unshare commands. Without options it shows a list of shared NFS file systems.

fstat

Identifies and shows all open files in the system.

mkdev

Adds a specified device to the system.

mknfsmnt

Mounts the specified directory from the specified host at the specified mounting point.

mount

Mounts a file system or shows a file system that is already mounted. The file system can be local or remote.

unmount

Unmounts local or remote file systems. AIX also have a unmount command with the same syntax.

limit

Set limitations on the system resources available to the current shell.

ulimit

Sets and shows the size limits used by the shell and its child processes. Shows the current limit if not given.

fold

Breaks lines in text files to the specified width.

rpl

Replaces text strings in a file.

sort

Sorts and merges lines from the specified files or from STDIN, and prints them to STDOUT.

split

Splits a file into a set of smaller files. The output files will get a double letter extension (.aa, .ab, .ac ...).

head

Shows the head part of the file specified to STDOUT. Shows the First 10 lines in a file by default.

tail

Shows the tail end of the file specified to STDOUT. Shows the last 10 entries in a file by default.

top

Shows a list of the most active CPU processes.

tr

Replaces or deletes characters while copying from STDIN to STDOUT

3.8. Important Unix Files – System Administration

Some of the important files that we need to be aware of in terms of unix system administration are as follows:

/etc/hosts

/etc/defaultrouter

Defines the systems default routers. Values must be separated with whitespace, # can be used for comments.

/etc/gateways

Contains all the routes and default gateways for the system.

/etc/hostname.interface

Contains the hostname of the system and should match the hostname defined in the /etc/hosts file. The file is named with the interface name, such as hostname.hme0 or hostname.le0

/etc/hosts

Configures names and aliases of IP-addresses. Fields should be separated with Tab or white space.

/etc/inetd.conf

Is the Internet server database ASCII file that contains a list of available servers. Is invoked by inetd when it gets an Internet request via a socket.

/etc/inittab

Is a script used by init. Controls process dispatching.

/etc/lilo.conf

Is the configuration file used by the Linux Loader while booting.

/etc/modules.conf

Loads modules specific options at startup.

/etc/mygate

Defines the systems default router or gateway.

/etc/myname

Specifies the real host name for the system.

/etc/netsvc.conf

Specifies how different name resolution services will look up names.

/etc/nodename

Specifies the real hostname for the system.

/etc/nologin

Is a text file message that is shown to the user who tries to log on during a system shutdown process. After the message appears the log on procedure ends.

/etc/printcap

Describes printers and allows dynamic addition and deletion of printers by the spooling system.

/etc/rc.conf

Is a configuration file used to configure the system daemons. It has three sections, the first turns features on or off, the second turns daemons on or off and the third sets parameters for the daemons in use.

/etc/resolv.conf

Configures DNS name servers to use for hostname lookups.

/etc/sysconfig/network

Configures the system's network. Specifies hostname and gateway.

/etc/xinetd.conf

Contains the configuration for the extended internet services started by the xinetd command

/etc/passwd

The user database, with fields giving the username, real name, home directory, encrypted password, and other information about each user. The format is documented in the passwd manual page. The encrypted passwords are much more commonly found in the /etc/shadow these days. This means that almost everything about the user except the password is stored in the passwd file. History and convention make a name change undesirable.

/etc/groups

Similar to /etc/passwd, but describes groups instead of users. See the group manual page in section 5 for more information

3.9. Installing & Uninstalling Software Packages

An operating system is an important part of a computer system. A computer system can be described as a collection of hardware and software components.

3.9.1. rpm

Used to install/uninstall or manipulate an RPM [Redhat Package Manager] package.

Syntax:

To install RPM Package:

```
rpm– i [rpmfilename]
```

To uninstall RPM Package:

```
rpm– i [rpmfilename]
```

Example:

To install a package by name mysql3.1.rpm

```
rpm– i mysql3.1.rpm
```

Chapter 4

4. I/O REDIRECTION, PIPES AND FILTERS

4.1. OBJECTIVE OF CHAPTER 4

The objective of this chapter is to discuss about the concept of standard input, output and redirecting the input and output to files. Here, you are taught how to combine simple UNIX commands to build complex commands using the concept of pipes. Also, you learn about many powerful utilities of UNIX that perform filtering of data.

In this class you will learn the following:

- Standard Input, Standard Output, Standard Error
- I/O Redirection
- Pipes
- File search using find
- Pattern matching using grep
- Sort, Cut, Paste utilities
- Head, Tail, more utilities

4.2. STANDARD INPUT

Every time you instruct the shell to bring or execute a new process, the new process receives several essential pieces of information from your current shell: you user ID, your home directory, various environment settings and also the file descriptors for three open files. Whenever programs read or white files, they do not use filenames, but file descriptors which are integer values like 0, 1, 2, etc. The files are used to read in information (Standard Input, write output (Standard output) and write error messages that are pertinent to the utility (standard error). Their respective file descriptors are: 0, 1 and 2.

Normally when you run a command, you standard input is connected to the keyboard, standard output is connected to the display and standard error is also connected to the display screen. Thus a command reads from your keyboard (standard input), and writes to the screen (standard output or error).

4.3. I/O REDIRECTION

The shell allows the redirection of the standard input, output, and error of a command. Redirection of input/out is the capability to change the source of the input and the destination of the output. This is done by using greater than (>) and less than (<) signs.

4.3.1. Output Redirection:

Under the UNIX system, the output from a command usually intended for standard output can be easily diverted to a file. This capability is known as output redirection.

Syntax

```
Command > filename
```

The above format indicates that the command is being diverted to a file filename using a > sign.

Examples 3.1

- Suppose you wanted to store the names of the logged_in_users inside a file names

```
$ who > names
```

In the above example, who command gets executed and instead of writing the output to the standard output (terminal), the output is being written to the file names. To display the contents of the names file, cat command can be used.

```
$ cat names
Root        pts/0       May17 11:27     (80.0.0.98)
Plokam      Pts/1       May17 11:27     (80.0.0.98)
Oracle      Pts/2       May17 11:27     (80.0.0.98)
Ajay        Pts/3       May17 11:27     (80.0.0.98)
```

- The echo command also writes to the standard output, so it can be redirected:

```
$ echo Hi, Good Morning > salute

$ cat salute

Hi, Good Morning
```

- If a command redirects its output to a file and the file already contains some data, then that data, then that data is lost and the new data is overwritten. Consider the following example:

```
$ echo Hi, Good Morning > salute

$ cat salute

Hi, Good Evening
```

As can be seen above, the previous contents of the file salute were lot when the second echo command was executed.

- To retain the existing contents, consider the following example:

```
$ echo Hi, Good Morning > salute

$ cat salute

Hi, Good Morning

$ echo Hi, Good Evening > salute

$ cat salute

Hi, Good Morning

                    Hi, Good Evening
```

The second echo command uses a different type of redirection symbol >> (double greater than) sign. This sign causes the standard output from the command to be appended to the specified file. Therefore, the previous contents of the file are not lost and the new output is simply added on to the end of the first.

Note: In the output direction, you can make the stdout explicit by preceding the > by the number 1, which is the file descriptor for stdout.. ($ echo Hi, Good Morning 1> salute)

4.3.2. Input Redirection:

Just as the output of a command can be redirected to a file, so can the input of a command be redirected from a file i.e., the input for a command can be taken from a file instead of taking from the standard input keyboard).

Syntax

```
Command < file name
```

In this case, less than (<) sign is used to redirect the input of command from the filename.

Example 3.2

```
$ sort < mydata
```

When the sort command opens the standard input and begins to read, it will read from the file mydata, not from keyboard.

4.3.3. Error Redirection:

As mentioned earlier, along with the standard output, the error messages, if nay, generated by executing the command is also displayed on the screen. As times it may be difficult to catch the error message as it would get mixed up with the standard output. It would be better, if the error messages are made available separately. Just as the output is redirected to a file, the error can also be redirected. In the command line, you redirect standard the error by preceding the '>' out put redirection symbol by a '2'.

Example 3.3

```
$ 1s *.txt
1s: *.txt not found

$'1s *.txt 2> /tmp/err.out

$cat /tmp/err.out
1s: *.txt not found
```

4.3.4. Input and Output Redirection:

You can simultaneously redirect the input and the output of acommand provided the command reads its input from the standard input and writes the output to its standard output.

Examples 3.4

```
$ cat names
```

```
root            pts/0           May 17   11.27     (80.0.0.98)

 Plokam             Pts/1               May 17   11:27        (80.0.0.98)
 Oracle             Pts/2               May 17   11:27        (80.0.0.98)
 Ajay               Pts/3               May 17   11:27        (80.0.0.98)

$ sort < names > sort_names
$ cat sort_names
 Ajay               Pts/3               May 17   11:27        (80.0.0.98)
 Oracle             Pts/2               May 17   11:27        (80.0.0.98)
 Root               Pts/0               May 17   11:27        (80.0.0.98)
 Plokam             Pts/1               May 17   11:27        (80.0.0.98)
```

In this example, the sort command takes its input from the file names and its output is redirected to the file sort_names.

4.4. PIPES

The UNIX system enables the user to effectively connect two or more commands together. This connection is called a *pipe.* A pipe enables you to take the output from one command and feed it directly into the input of the another command. A pipe is effected by the character / (called pipe), which is placed between two commands. All of the commands in a pipeline are executed sequentially. Unix handles the flow of data from one command to the next, producing the effect as if one command is being executed.

As an example of a pipe, suppose you wanted to count the number of files in your directory. Now we know that 1s command displays the names of all files, and the command wc-1 is used to count the number of lines in a file. So both these commands can be effectively piped to get the desired result.

Examples 3.5

```
$1s  1.wc – 1
        10
```

The output indicates that the directory contains 10 files. Firstly, the 1s command is executed to list the files, this output is then sent through the pipe as an input to the wc-1 command, which is executed to give the number of files. The final output is the output of the last command.

- Similarly, command which can be piped with sort command to get a sorted list of logged_in_users

```
$ who / sort
 Ajay               Pts/3               May 17   11:27        (80.0.0.98)
 Oracle             Pts/2               May 17   11:27        (80.0.0.98)
```

```
Root          Pts/0          May 17   11:27        (80.0.0.98)
Plokam        Pts/1          May 17   11:27        (80.0.0.98)
```

- To save the sorted list of users in a file called users you can use the output redirection on the sort command.

```
     $ who / sort_names
     $ cat sort_names
Ajay          Pts/3          May 17   11:27        (80.0.0.98)
Oracle        Pts/2          May 17   11:27        (80.0.0.98)
Root          Pts/0          May 17   11:27        (80.0.0.98)
Plokam        Pts/1          May 17   11:27        (80.0.0.98)
```

4.5. FILTERS

Filter is a UNIX utility that takes input data, processes it, and sends the result to the output. A filter selectively alters the data that passes through it. For example, the grep command (discussed below) filters out unwanted data and passes on the selected data to an output file.

4.5.1. Sort

Sort is a filter utility that is used to or order the contents of the indicted file, alphabetically and display the result of the sort at the terminal. The original contents of the file remain unchanged.

Syntax

```
Sort [options] filename
```

Options can be

-n	*sorts the input numerically*
-r	*sorts the input in reverse order*
-f	*ignore the significance of uppercase and lowercase letters*
-o	*sends the output to a file rather than the standard output*
-k	*sort the records field wise*
-t SEP	*use SEParator instead of non- to whitespace transition*

Example 3.6

There is a file students that contains the names of the students of a particular batch. In order to display the names in an alphabetical order, we use the sort command.

```
$ cat students
Rama
Anju
Anita
Sanjana
Hema
Jeevan
Bobby
Neha
Priya
```

The sort command would order the names alphabetically

```
$ sort students
Anita
Anju
Bobby
Hema
Jeevan
Neha
Priya
Rama
Sanjana
```

Consider a file num which contains numbers as its contents

```
$ cat num
21
56
34
78
45
90
03
67
82
19
20
47
```

To sort this file, use –n option with the sort command

```
$ sort –n num
03
19
20
21
34
45
47
56
67
78
82
90
20
47
```

If the file contains more than one field, you can also sort the file according to a particular field. Consider, the following file dbase

```
Suresh 23          salem
Gupta 23           trichy
Krishna 21         madhrai
David 18           madrass
George 23          kanchi
```

To sort the above file based on the second field use the command

```
$sort-k 2,2     dbase
david 18           madrass
Krishna 21      madhrai
George 23          kanchi
Gupta 23           trichy
Suresh 23          salem
```

In the above examples, the fields were separated by white space. This need not be the case always. Consider the file/etc/password. Here the fields are separated by a colon and not white space.

Suppose you want to sort this file by the third field, enter

```
$ sort +2 / etc/passwd
adm:x:4:4:Admin:var/adm;
```

```
bin:x:2.2::/usr/bin:
daemon:x:1:1::/:
guest:x:1116:100:guest User:/:/usr/bin/sh
listen:x:37:4:Network Admin:/usr/net/nls:
body:x:60001:60001:Nobody:/:
nuucp:x:9:9:uucp Admin:/var/spool/uucppublic:/usr/lib/uucp/uucico
oracle:x:102:1::/home/oracle:/sbin/sh
pppuser:x:1115:100:PPPuser:/:/usr/sbin/aspppls
root:x:3:3::/:
uucp:x:1002:101:UNIX User – Ajay Sarma:/user/ajay:/usr/bin/sh
plokam:x:1001:101:UNIX User – Plokam Taden:/user/plokam:/usr/bin/sh
lp:x:71:8:Line Printer Admin:/usr/spool/lp:
oracle7:x:101:100:Oracle7 Workgroup Server
user:/opt/oracle7:/usr/bin/sh
noaccess:x:60002:60002:No Access User:/:
smtp:x:0:0:mail Daemon User:/:
wgsuser:x:100:60001:Oracle? Web user:/opt/wgsuser:/usr/bin/sh
```

The results are not sorted by the third field because sort expects spaces for delimiter. To request sorting by the third field with colon as the separator, enter

```
$ sort –t: +3 /etc/passwd
smtp:x:0:0:Mail Daemon User:/:
guest:x:1116:100:guest User:/:/usr/bin/sh
oracle7:x:101:100:Oracle7 Workgroup Server
user:/opt/oracle7:/usr/bin/sh
pppuser:x:1115:100:PPPuser:/:/usr/sbin/aspppls
ajay:x:1002:101:UNIX User – Ajay Sarma:/user/ajay:/usr/bin/sh
plokam:x:1001:101:UNIX User – Plokam Tanden:/user/plokam:/usr/bin/sh
oracle:x:102:1::/home/oracle:/sbin/sh
root:x:0:1:Super-User:/:/sbin/sh
bin:x:2:2::/usr/bin:
sys:x:3:3::/:
adm:x:4:4:Admin:/var/adm:
listen:x37:4:Network Admin"/usr/net/nls:
uucp:x:5:5:uucp Admin:/usr/lib/uucp:
nobody:x:60001:60001:Nobody:/:
wgsuser:x:100:60001:Oracle7 web user:/opt/wgsuser:/usr/bin/sh
noaccess:x:60002:60002:No Access User:/:
```

```
lp:x:71:8:Line Printer Admin:/usr/spool/lp:
nuucp:x:9:9:uucp Admin:/var?spool/uucppublic:/usr/lib/uucp/uucico
```

4.5.2. wc

Using the UNIX utility wc, you can count all the lines, words and characters in a file.

Syntax

-l	number of lines
-w	number of words
-c	number of characters

Example 3.7

```
$ wc num
12              12              36 num
```

The four fields are:

Number of lines	12
Number of words	12
Total number of characters	36
Name of the file	num

- To count only the number of lines in a file, -1 option can be used with the wc command

```
$ wc –1  num
          12   num
```

- Similarly, to count only the words and only to count the characters in file, the options –w and –c can be used respectively.

```
$ wc –w num
          12   Num
$ wc –c      num
          36   num
```

- More than one option can be used at the same time.

```
$ wc –1cnum
```

```
12          36    num
```

4.5.3. tr

tr command can be used to translate a set of characters to another. It reads from the standard input, searches for all the special characters, and translates each into another specified character and writes to the standard output the translated form of the input. It cannot read/write from/to the files. Therefore, you must use redirection symbols or pipe the input to the tr command.

Syntax

```
tr string1 string2
```

where string1, string2 are the translation control strings. Each string represents a set of characters to be converted into any array of characters used for the translation.

```
$ tr “123” “ABC” < mydata
```

As a result of executing tr, the following translation takes place in the file mydata which is output on the terminal.

```
1 -> A
2 -> B
3 -> C
```

Example 3.8

- To convert all lowercase letters into uppercase and display it

```
$ tr “[a-z]” “[A-Z]” < students

RAME
ANJU
ANITA
SANJANA
HEMA
JEEVAN
BOBBY
NEHA
```

4.5.4. grep

The grep command searches files for a pattern and prints all lines that contain that pattern.

Syntax

```
grep [options] pattern filename
```

If no files are specified, grep assumes standard input. Normally, each lines found is copied to standard output. The file name is printed before each line found if there is more than one input file.

The file is searched line by line for the pattern. Every line that contains the pattern is displayed on the terminal. The pattern tht is searched for in the file is called a result expression.

Regular Expressions

Character	Description	Example
[class]	A character class. Matches any one character in the *class.*	"[xyz]" specified the pattern either x or y or z.
[c1-c2]	Matches any one of the character specified in the range	"[a-d]" specifies the pattern either a, b, c or d
^	Pattern following it must occur at the beginning of each line	"^my" specified the pattern "my" should appear at the beginning of each line.
[^ class]	Does NOT match any of the characters specified	"[^abc] specifies that the pattern should not contain a,b or c.
$	Matches and end of line	"ball$" specifies the pattern "ball" at the end of each line
\	Escapes the special meaning of the character	"\$900" specifies the pattern "$900", the sign $ loses its special meaning
.(dot)	Matches any single character	"[abc]." Specifies the pattern a, b, or c followed by any one character

The options can be:

-i	Ignore upper/lower case distinction during comparisons.
-n	Precede each line by its line number in the file (first line is 1).
-c	Print only a count of the lines that contain the pattern
-l	Print only the names of files with matching lines, separated by NEWLINE characters. Do not repeat the names of files when the pattern is found more than once.
-v	Print all lines except those that contain the pattern.

Examples 3.9

Consider the file stud_rec with the following data

```
$ cat stud_rec
Anjana        23    UC++        Annanagar
Divya         19    UC++        Padi
Farida        26    DBA         T.Nagar
```

```
        Prashant                22      UNIX            Admin Kilpauk
        Rahul                   21      ORVB            Shenoy Nagar
        Fardeen         27      DBA             Adayar
```

- Display the students who are taking UCC++ course

```
$ grep UC++ stud_rec
Anjana                  23      UC++            Annanagar
Divya                   19      UC++            Padi
```

- Display the lines that start with "A':

```
$grep '^A' stud_rec
Anjana                  23      UC++            Anannagar
```

- Display the records in which the students' age is in twenties

```
$grep 2[0-9] stud_rec


        Anjana          23      UC++            Annanagar
        Farida          26      DBA             T.Nagar
        Prashant        22      UNIX            Admin Kilpauk
        Rahul           21      ORVB            Shenoy Nagar
        Fardeen 27      DBA             Adayar
```

- Display the records with their line numbers

```
$grep –n 2 [0-9] stud_rec
        2:Anjana23      UC++            Annanagar
        4:Farida 26     DBA             T.Nagar
        5:Prashant      22      UNIX            Admin Kilpauk
        6:Rahul  21     ORVB            Shenoy Nagar
        7:Fardeen       27      DBA             Adayar
```

- Display the total number of records with age 27

```
$grep –c 27 stud_rec
1
```

- Display the students who do ORVB

```
$grep –i orvb stud_rec
Rahul           21      ORVB            Shenoy Nagar
```

4.5.5. Using grep as a Filter:

The grep utility can be used with other utilities as an information filter.

```
$ 1s | grep samples
```

The output of the 1s command is directly given to the grep command as input. The pattern "samples" is searched in the list of filenames, with the grep command.

Note: Be careful using the characters $, *, ^, |, (,), and \ in the pattern_list because they are also meaningful to the shell. It is safest to enclose the entire pattern_list in single quotes '…'.

4.5.6. find

find command locates the files that match specified expression. It starts searching for files in the given directory and continues its search through all subdirectories. It also provides a mechanism for performing actions on the files that meet the search criteria.

Syntax

Find path expression

Where path is the list of directories to be searched. The names are separated by space

or

tab. To search the current directory, you use the dot (.) notation

expression can be:

-name <file>	name file to be found in the specified directory. For example: find/usr/ajay –name *.c will search all files with extension .c in the /usr/ajay directory
-print	Causes the current path name to be printed.
-type c	type of the file is c, where c is b, c, d, l, p, or f for block special file, character special file, directory, symbolic link, fifo (named pipe), or plain file, respectively.
-size n[c]	to find the files containing n blocks. If n is followed by c, then n is counted in characters instead of blocks.

-exec cmd	to execute the command cmd. You can set a pair of braces { } to signify the presence of current pathname. The cmd must end with an escaped semicolon (\;).

The expressions can be used in combination also.

-ok command Like –exec, except that the generated command line is printed with a question mark first, and is executed only if the user responds by typing y.

Example 3.10

- Find all the files under the current directory and print them

```
$ pwd
/usr/plokam
$ find .
.
./salary
./file1
./file2
./file3
```

- Display the files in the /usr/anay directory having a size of more than 500 bytes

```
$ find/user/ajay –size +500 –print
/usr/ajay
```

- Remove all files in the current directory named a.out or *.out that have not been accessed for a week.

```
$ find .-name a.out –o –name '*.o' –atime +7 e –exec rm { } \;
```

- Prompt the user with the name of each plain file in the current directory. If the answer is 'y', the file is to be removed.

```
$ find . –type f –ok rm { } { \;
<rm … ./file1 >?        n
<rm … ./file2 >?        n
<rm … ./file3 >?        y
```

4.5.7. cut

cut is a filter utility that is used to cut out columns or fields from each line of a file. A field is normally separated by a tab character (default delimiter), but it can also be delimited by any other character.

Syntax

```
cut –c columnlist [ file … ]
cut –f fieldlist [-d delim ] [ -s] [file … ]
```

where list is a comma-separated or blank-character-separated list of integer field numbers (in increasing order), with options –to indicate ranges (for instance, 1,4,7; 1-3,8; -510 (short for 1-5,10);ir 3- (short for third through last fields)).

-c list The list following –c specifies character positions (for instance, -c1-72 would pass the first 72 characters of each line).

-f list The list following –f is a list of fields assumed to be separated in the file by a delimiter character.

-d delimiter The character following –d is the field delimiter (-f option only). Default is tab. Space or other characters with special meaning to the shell must be quoted, delimiter can be a multi-byte character.

Example 3.11

- Consider the file names containing the following information

```
Pat     Gaja      22      C++
Tedd    Boycott   34      ORVB
Anju    Ahuja     25      C++
Hema    Malini    27      DBA
Raju    Naidu     31      ORVB
Sonu    Misra     28      ORVB
```

- Get the second, fourth fields from the file using the command

```
$ cut –f 2, 4 names
Gaja      C++
Boycott   ORVB
Ahuja     C++
Malini    DBA
Naidu     ORVB
Misra     ORVB
```

- Extract the fields with the names of users and the times they logged in from the out of who

```
$ who | cut– c1-9, 25-29
root      May 1
root      May 1
plokam    May 1
oracle    May 1
ajay      May 1
root      May 1
```

- List the Login Id's and their corresponding names from /etc/passwd. The delimiter here is:

```
$ cat /etc/passwd
root:x:0:1:Super-User:/:/sbin/sh
daemon:x:1:1::/:
bin:x:2:2::/usr/bin:
sys:x:3:3::/:
adm:x:4:4:Admin:/var/adm:
lp:x:71:8:Line Printer Admin:/usr/spool/lp:

$ cut –d: -f1, 5 /etc/passwd
root:Super-User
daemon:
bin:
sys:
adm:Admin
lp:Line Printer Admin
smtp:Mail Daemon User
```

4.5.8. paste

The paste command joins two files horizontally (column wise). For instance, if a file contains the list of students and the other file contains their ages, the two files can easily be joined together to get two columns: one containing names and the other containing ages

Example 3.12

```
$ cat students
Rama
Anju
Anita
Sanjana
Hema
Jeevan
Bobby
Neha
Priya

$ cat stu-age
29
19
22
```

```
32
35
24
19
23
31
```

- Now, paste command can be used to merge the files in a side by side manger:

```
$ paste students stu_age
Rama        29
Anju        19
Anita       22
Sanjana     32
Hema        35
Jeevan      24
Bobby       19
Neha        23
Priya       31
```

Note: Use grep (1) to make horizontal "cuts" (by context) through a file, or paste (1) to put files together column-wise (that is, horizontally). The reorder columns in a table, use cut and paste

4.5.9. head

The head utility copies the first number of lines of each filename to the standard output. If no filename is given, head copies lines from the standard input. The default value of number is 10 lines.

Syntax

```
Head [-number] file
```

Example 3.13

- Consider the file my book. It contains 16 lines

```
$ cat my book
This is my first book.
I am trying to & write.  Here, I test my skill in writing.
I leave 2 lines blank.


You may be wondering Why?
Because, I am a writer and I have to think.
```

```
I want to write but nothing seems to be coming out.
I have severe constipation of ideas.

Anyway, I am not going to give up.
Definitely, one day I am going to be a greater writer,.
But, today I think, I need a break.
Good Bye.
```

- Display the first 10 lines of my book

```
$ head my book
This is my first book
I am trying to a write.  Here, I test my skill in writing
I leave 2 lines blank

You may be wondering Why?
Because, I am a writer and I have to think
I want to write but nothing seems to be coming out
I have severe constipation of ideas
```

When more than one file is specified, the start of each file will look like:

==> filename <==

Thus, a common way to display a set of short files, identifying each one is by giving the file names as arguments.

- Display the first 5 lines of mybook

```
$ head –5 mybook
This is my first book
I am trying to a write.  Here, I test my skill in writing
I leave 2 lines blank
```

4.5.10. tail

tail command is the reverse of the head command. It is used to display the last few lines of a file. If no file is named, the standard input is used. By default, it displays the last 10 lines.

Syntax

```
Tail [ -/+ number] file
```

- Consider the file mybook discussed in the previous section. Display the last 10 lines of mybook.

```
$ tail mybook
Because, I am a writer and I have to think.
```

```
I want to write but nothing seems to be coming out
I have severe constipation of ideas

Anyway, I am not going to give up
Definitely, one day I am going to be a greater writer
But, today I think, I need a break
Good Bye.
```

- Display the last 7 lines of mybook

```
$ tao; -7 mybook
Anyway, I am not going to give up.
Definitely, one day I am going to be agreater writer.
But, today I think, I need a break.
Good Bye, Good Bye.
```

- Display the lines of mybook from 7th line till the end

```
$ tao; -7 mybook
Because, I am a writer and I have to think
I want to write but nothing seems to be coming out
I have severe constipation of ideas

Anyway, I am not going to give up
Definitely, one day I am going to be a greater writer
But, today I think, I need a break\
Good Bye.
```

4.5.11. more

more is a filter that displays the contents of a text file on the terminal, one screenful at a time. It normally pauses after each screenful. If more is reading from a file rather than a piepe, the percentage of characters displayed so far is also shown.

Syntax

```
more filename
```

More scrolls up to display one more line in response to a RETURN character; it displays another screenful in response to a SPACE character.

4.5.12. tee

Any output from a command that gets piped into another command is not seen at the terminal. Sometimes, you might want to save the output that is produced in the middle of a pipe. The tee command enables you to do it easily.

Syntax

```
tee file
```

The tee command simply copies the data coming in on standard input to standard output, in the meantime saving a copy in the specified file.

Example 3.14

- Enter the command give below

```
$ sort students | tee stud | more
Anita
Anju
Bobby
Hema
Jeevan
Neha
Priya
Rama
Sanjana
(EOF):
```

This will sort the student file, display it page wise and save it in file stud. Now the sorted file stud can be displayed on the terminal using cat command.

```
$ cat stud
Anita
Anju
Bobby
Hema
Jeevan
Neha
Priya
Rama
Sanjana
```

4.6. LAB EXERCISES

Working with files

1. Create files employee and department under Present Working Directory
2. Display the file employee and department
3. Append more lines in the employee and department file.
4. How will you create a hiddent file?.
5. Copy employee file to emp
6. Write the command to create alias name for a file.
7. Move department file to dept.
8. Copy emp file and dept file to TRIAL directory
9. If you compare a file with itself, what will the output?
10. Compare employee file and emp file
11. Append two more lines in emp file existing in TRIAL directory
12. Compare employee file with emp file in TRIAL DIRECTORY
13. Find the difenece between the above file
14. Remove the files in the TRIAL directory
15. Using a single command can you remove a directory with files in it?
16. Is there any command available to get back a deleted file?
17. Rename TRIAL as DATA
18. Copy DATA to another directory by name TRIAL
19. Create a file called dummy in TRIAL and link it to another file by name star
20. Link the dummy file in TRIAL to another file by name power in DATA

Chapter 5

5. UNIX SECURITY

5.1. OBJECTIVE OF CHAPTER 5

The objective of this chapter is to teach you the importance of security of a system. You learn about the security provided by UNIX. Particularly to get to know about the File and System Security provided by the operating system to defend the resources from being accessed by unauthorized users.

In this class you will learn the following:

- System security and File Security
- Login Access Control
- User Classes
- File Permissions
- File ownership

5.2. SECURITY

Being a multi-user operating system, UNIX has incorporated various security measures at different levels to ensure complete safety from any unauthorized access to the system.

The levels of security are:

1. System level.
2. File level.

5.2.1. System Level Security

The UNIX system has a feature called password, which prevents any unauthorized user to login to the system. When a user gives a login name, he/she is prompted for a password. If the user enters a wrong password, the UNIX system reports an error and the user is not allowed to login. This ensures that only the right person has an access to the system. Still it is possible for some notorious users to learn your password. This problem can be overcome if a user changes the password periodically, say, twice a month. The passwd command is used to change old passwords to new ones.

```
$ passwd
Enter Login password:
New password:
Re-enter new password:
```

The passwd command asks the user for the old password to make sure that the right person is changing the password. If the password entered is wrong, it will not update (change) the password. After entering the old and new passwords, it again prompts to re-enter the new password to confirm the changes.

5.2.2. File Level Security

After you have established login restrictions, you can control access to the data on your system. To doubly ensure the security of your files and data, UNIX has provided the users with security measures at the file level also. You may want to allow some people to read some files, and give other people permission to change or delete some files. You may have some date that you do not want anyone else to see.

As a member of a team working on a particular project, you may need to share files among each other. But at the same time, you may not like others, not belonging to the group, to tamper with your date. Everyone on a UNIX system is assigned to a group when their account are set up by the system administrator, and you can belong to several groups. Keeping in mind, the various needs of the users, different types of accesses were built in the UNIX system.

5.3. USER CLASSES

For each file, there are three classes of users:

Owner The file or directory owner – usually the user who created the file. The owner of a file can decide who has the right to read it, to write to it (make changes to it), or, if it is a command, to execute it.

Group	Members of a group
Others	All others who are not the file or group owner

5.4. FILE PERMISSIONS

A Designed user is permitted to access a file in any or all of the following modes:

Mode	***Description***
Read (r)	Designated users can open and read the contents of a file
Write (w)	Designated users cam write to the file (modify its contents), add to it, or delete it.
Execute (e)	Designated users can execute the file (if it is a program or shell script)
Denied (-)	Designated users cannot read, write opr execute the file

The users who have the permission to access are:

Owner	*rwx*
Group	*rwx*
Others	*rwx*

There are certain File administration commands that allow you to manipulate the permissions and user class of a file. They are:

ls	Lists the files in a directory and information about them.
chown	Changes the ownership of a file
chgrp	Changes the group ownership of a file
chmod	Changes permissions on a file.

5.4.1. ls

The file access permissions along with the owner and group information is provided by jy command A permission is displayed by an appropriate letter. If a particular permission denied it is marked by a hyphen (-) in that place.

```
$ ls – l chapl 12
-rw—r  --      1 plokam          plokam       10 Jun   9      12:46    chap
12
```

read, write and execute permissions for the owner, plokam (rwx)

read and write permissions for group, plokam (no execute permission) (rw-)

only read permission for others (r--)

5.4.2. chmod

The file access modes can be changed using chmod command, for a particular file. Suppose you want your **report** file to be read only for the group as well as others, you can set the mode of the file accordingly using chmod command.

5.4.3. Absolute Mode

You can change the permission of a file by giving an absolute numeric value to its mode property.

Syntax

```
chmod mode file
```

Where **file** is the name of the file whose mode to to be set mode is a three digit number to denote read, write and execute permissions. Each digit can take a value from 0 to 7.

The corresponding **mode** settings are:

Value Meaning

No permission

Execute only

Write only

Write and Execute

Read only

Read and Execute

Read and Write

All – Read, Write and execute

The left most digit is the permission for the owner of the file. The middle digit is the permission settings for the group and the right most digit is the permission settings for other users.

Example 4.1

To change the permissions for the file data, giving the entire world permissions to read, write and execute the file.

```
$ chmod 777 data
$ ls – l data
-rwxrwxrwx    1 plokam          plokam        10     Jun    9      12:49
      data
```

To change the permissions to where only the only the owner has the ability to read, write and execute the file,

```
$ chmod       700     xxxx
$ ls  -  1    xxxx
-rwx ------   1 plokam          plokam     10  Jun    9      12:50    xxxxx
```

5.4.4. Symbolic Mode

The other way to set the file access mode is by representing the permissions symbolically.

Syntax

chmod [modes] [grant permission] [filemodes] file

modes

u	owner of the file
g	group to which the owner belongs
o	others
a	all (by default)

grant permissions

+ to grant

to revoke

= to assign

filemodes

r	read
w	write
x	execute

Example 4.2

To grant the group plokam with the execute permissions on file chap 12

```
$ chmod          g+x       chap 12
```

To revoke the write permissions on file chap 12 from the owner. The changed permissions can be seen in the long listing

```
chmod u–w chap 12
$ ls      -1          chap 12
-r-xrwxr--           1          plokam   plokam               10 Jun    9           12:46     chap 12
```

As can be seen from the listing, the owner now has only read and execute permissions, the group has all read, write as well as execute permissions on the file chap 12.

To assign a user with a particular permission, = sign is used. The command below will assign the execute permission for file chap 12 to others.

```
$ chmod o=x chap 12
$ ls – l chap 12
-r-xrwx—x     1     plokam  plokam  10    Jun    9     12:46          chap12
```

You can remove all permissions by using the = operator and no permission string. The command below will remove all permissions for chap 12 from all users.

```
$ chmod = chap 12
$ ls – l chap 12
---------- 1     plokam  plokam  10    Jun    9     12:46   chap12
```

Note: *Only the owner of the file or root can assign or modify file permissions.*

5.4.5. chgrp

With chgrp command, a user can change the group ownership of a file or a directory. This allows you to change the group ID so that users of another group can access the files and directories. It may be useful when you want some of your files to be shared with another group.

Syntax

```
chgrp   (group)        (file_lists)
chgrp   (group)        (directory_list)
```

Where **group** *the group you wish to permit access to your files or directories.*

File_list *one or more files whose group ID has to be changed*

Directory_list *one or more directories whose group ID has to changes*

Example 4.3

To changes the group of file chap 12 from plokam to rad so that the users of group rad can access the file chap 12.

```
        # su
        # password
        # ls – l chap 13
        -rw  --r-- 1 plokam        plokam        2       Jun     9       12:55    chap
13
        # chgrp rad cahp13
        # ls – l chap 13
        -rw  --r-- 1 plokam        rad           2       Jun     9       12:55    chap
13
```

```
#
# chgrp plokam cahp 13
# ls – l chap 13
-rw  --r-- l plokam plokam          2       Jun     9       12:55   chap13
```

> **Note:** *Only the superuser can change the group of a file or a directory.*

5.4.6. chown

The owner of the files can be changed by using the chown command. Suppose you are leaving a department or the company, you may want to give your files to other users.

Syntax

chown (owner) (file_lis)

chown (owner) (directory_list)

Where **owner** *name of the user specifying the new user*

file_list *one or more files to change the ownership to a new user ID*

directory_list *one or more directories to change the ownership to a new user*

Example 4.4

```
# ls – l chap 13
-rw  --r-- 1 plokam        plokam          2       Jun     9       12:55   chap
13
#chown ajay chap 13
# ls l chap 13
-rw  --r-- l ajay    plokam          2       Jun     9       12:55   chap 13
```

Now the file chap 12 is owned by the user ajay since the ownership of the file chap 12 has been changed from plokam to ajay.

> **Note:** *Only the superuser can change ownership of the files or directories.*

5.4.7. useradd

The owner of the files can be changed by using the chown command. Suppose you are leaving a department or the company, you may want to give your files to other users.

Syntax

useradd (owner) (file_lis)

chown (owner) (directory_list)

Where **owner** *name of the user specifying the new user*

file_list *one or more files to change the ownership to a new user ID*

directory_list *one or more directories to change the ownership to a new user*

Example 4.4

```
# ls-  l chap13
-rw-r--r-- 1 plokam plokam        2      Jun    9      12:55  chap13

# chown john chap13
# ls l chap 13
-rw-r--r-- 1 john   plokam        2      Jun    9      12:55  chap13
```

5.4.8. groupadd

The owner of the files can be changed by using the chown command. Suppose you are leaving a department or the company, you may want to give your files to other users.

Syntax

chown (owner) (file_lis)

chown (owner) (directory_list)

Where **owner** *name of the user specifying the new user*

file_list *one or more files to change the ownership to a new user ID*

directory_list *one or more directories to change the ownership to a new user*

Example 4.4

```
# ls-  l chap13
-rw-r--r-- 1 plokam plokam        2      Jun    9      12:55  chap13

#chown ajay chap13

# ls l chap13
-rw-r--r-- 1 ajay   plokam        2      Jun    9      12:55  chap13
```

5.4.9. userdel

The owner of the files can be changed by using the chown command. Suppose you are leaving a department or the company, you may want to give your files to other users.

Syntax

chown (owner) (file_lis)

chown (owner) (directory_list)

Where **owner** *name of the user specifying the new user*

file_list *one or more files to change the ownership to a new user ID*

directory_list *one or more directories to change the ownership to a new user*

Example 4.4

```
# ls – l chap 13
-rw  --r-- 1 plokam        plokam        2      Jun      9       12:55    chap
13
#chown ajay chap 13
# ls l chap 13
-rw  --r-- l ajay     plokam        2      Jun      9       12:55    chap 13
```

5.4.10. groups

The owner of the files can be changed by using the chown command. Suppose you are leaving a department or the company, you may want to give your files to other users.

Syntax

chown (owner) (file_lis)

chown (owner) (directory_list)

Where **owner** *name of the user specifying the new user*

file_list *one or more files to change the ownership to a new user ID*

directory_list *one or more directories to change the ownership to a new user*

Example 4.4

```
# ls – l chap 13
-rw  --r-- 1 plokam        plokam        2      Jun      9       12:55    chap
13
#chown ajay chap 13
# ls l chap 13
-rw  --r-- l ajay     plokam        2      Jun      9       12:55    chap 13
```

5.5. LAB EXERCISES

File Security Commands

1. Change your password
2. Create a file salary in your current directory.
3. Check the permissions of the file salary.
4. Change the access permissions of salary so that the users and the group have execute permission only.
5. Change the rights of salary according to the conditions given below in symbolic and absolute modes:

User	read only
Group	read and execute
Others	Write and execute

6. Reovke read permissions from group for the file salary.
7. Take away all the permissions from all the users file salary.
8. Using symbolic mode, assign only read write modes to all the users for the file salary.

Chapter 6

6. UNIX PROCESSES

6.1. OBJECTIVE OF CHAPTER 6

The objective of this chapter is to introduce you to an important concept UNIX called process. Here you learn about the process life cycle. You are taught the commands that help you in working with various processes that are being executed by the system.

In this chapter you will learn the following:

- Multiuser and Multitasking System
- Foreground and background process
- Suspended process
- Fg, bg, jobs commands
- Nohup, kill commands
- Bash shell – history commands

6.2. PROCESSES

As learnt earlier in the first session, Unix is a multiuser as well as multitasking system. This means that more that one user can use the system and each user can perform a number of tasks at the same time. In UNIX system, a task is referred to as a process. A process is the existence of an executing program in the computer.

Since the CPU can carry out one task at a time, multitasking is accomplished using the concept of time sharing. The UNIX kernel maintains a list of processes started by a user and each process is allocated a small time quantum, called time slice, during which it can carry our its execution. When its time has elapsed, first process is suspended and other is given a change to run. When all process in the list have had their chance to run, the kernel switches back to the first process, starting the execution from where it was suspended. In this way, process is allowed to work its way to completion, a little bit at a time.

The time slice for each process is so small (a few hundredths of a second), it gives the impression that each user is being served simultaneously, even though, they are being served one at a time.

There are various types of processes that can be started simultaneously by a user. These processes fall into the following categories:

Foreground processes- processes with which the user directly interacts.

Background processes- processes that are dissociated from the terminal.

Suspended processes processes whose execution has been suspended for a while.

6.2.1. Foreground process

A foreground process is the one in which the user interfaces with, from the keyboard. The user writes a command at the $ prompt and waits till the execution is over.

Example 5.1

```
$      date
Wed    Jun    9      17 : 56 : 22 GMT    1999
```

When the command is isused, it gets executed and the result is echoed to the screen.

6.2.2. Background Process

A background process can be started along with a foreground process. When a command is appended by an ampersand (&) sign, it gets dissociated from the terminal and carries its execution in the background.

Example 5.2

```
$ cat data > /tmp/out &
[1] 1074
$
[1} + Done (0)             cat data      1>/tmp/out
$
```

The above command sort sorts the file data and redirects the output to another file out. This whose process is appended by an &, implying that is has to be carried out in the background. When a process is sent to the background, UNIX automatically displays a unique number identifying that process, called process id (PID). In the above example, it is 1074. After displaying the PID number, the terminal is immediately available to the user (indicated by the $ sign).

It is useful to carry out those processes in background which do not require user input so that, in the meantime, another process can be executed in the foreground. For example, printing a large document in the background while editing another document in the foreground.

6.2.3. Suspended process

A user can suspend a job running in background or one running is foreground. This is useful when you are running a command and you need to perform some other function. A running job can be suspended by pressing Ctrl-Z.

Example 5.3

Display a file and directory name starting with d as the first character

```
$ ls          grep “^ d”  @
[1]           14725
$      dbfile
demo
demol
[1]    +      Done          ls            grep “^d”    @
```

To set the vi editor options in the dollar prompt

```
$ set  -o     vi
```

Now you can invoke all the commands which are working in the vi editor

```
Eg:
$      find   /      -name  -print sample &
$      find   /      -name  -print sample &
$      find   /      -name  -print sample &
$      find   /      -name  -print sample &
$      find   /      -name  -print sample &

$   jobs

[6]    +      Running            find   /      -name sample &
[5]    +      Running            find   /      -name sample &
[4]    +      Running            find   /      -name sample &
[3]    +      Running            find   /      -name sample &
[2]    +      Running            find   /      -name sample &
```

```
[1]    +      Running            find   /      -name sample &
```

6.2.4. Switching Between Process

Sometimes you may need to bring a process in its suspended stat to its running state, either in background or foreground. This can be accomplished by using fg and bg commands. To know which processes are in background or in a suspended state, you use the jobs command.

6.2.5. jobs

The jobs command lets you know about the process that are in background or in suspended state.

Example 5.4

+ sign indicates the current background process

sign indicates the previous process

Note: *Using the option –l, the process Ids of the processes can also be displayed.*

6.2.6. stop

This command is used to stop the background process.

Syntax

```
$ stop [ % job id ]
```

Example 5.5

```
$ stop %6
$ jobs
[6]    +    stopped (signal)    find    /    -name sample&
[5]    +    Running             find    /    -name sample&
[4]    +    Running             find    /    -name sample&
[3]    +    Running             find    /    -name sample&
[2]    +    Running             find    /    -name sample&
[1]    +    Running             find    /    -name sample&
```

6.2.7. fg

The ***fg*** commands lets you resume suspended jobs by placing them in foreground. The suspended jobs, when brought back in foreground, become interactive again.

Syntax

```
fg [ % job ]
```

Where	*% **job** cab be :*	
	% num	*a job number for an associated process*
	% +	*current job; the last job suspended*
	% -	*previous job, process before the current job*

Example 5.6

Checke the job status

```
$ jobs

[6]    +      stopped (signal)    find   /      -name  sample&
[5]    +      Running             find   /      -name  sample&
[4]    +      Running             find   /      -name  sample&
[3]    +      Running             find   /      -name  sample&
[2]    +      Running             find   /      -name  sample&
[1]    +      Running             find   /      -name  sample&
```

Bring the stopped job 2 to foreground

```
$  fg  %6
```

6.2.8. bg

The bg command lets you resume the suspended jobs by placing them in background. For example, let's assume you are running a long process in background. At some time later, you realize that you need to give some input to the process. You can suspend the job, give the input and restart the process in background using bg command.

Syntax

```
bg      [ % job]
```

Where	**% job** *can be*	
	% num *a job number for an associated process*	
	% +	*current job; the last job suspended*
	% -	*previous job, process before the current job*

Example 5.7

Check the jobs running

```
$  jobs
[1] + Running                        ls– Rl   /   l>/tmp/out 1
```

Bring the job to foreground ans suspend it.

```
$ fg  %  6
[1]  ls - Rl   /   1>/tmp/out 1
```

```
^Z
[1]  +   Stopped  (user)                ls – Rl   /   l>/tmp/out 1
```

To restart the suspended job type **bg % 6** at the $ prompt

```
$  bg %  6
[1]  ls - Rl   /   1>/tmp/out 1  &

$ jobs

[6]  +  Running                         ls – Rl   /   l>/tmp/out 1
```

6.2.9. ps

The ps command is used to display the status of the running processes. The ps command reports the process ID, the associated terminal type, the amount of time a process has used and the command being executed.

Syntax

ps [options]

Options:

-u <username>	*to display the process status for a particular user*
-e	*to display the information about every process*
-f	*to display the full listing of header information*
-t <terminal>	*to display the process status for a particular terminal*

The column headers are:

UID	*user's ID no. of the process's owner*
PID	*process ID of the process*
PPID	*Process ID of the parent process*
C	*the processor utilization used for scheduling purposes*
STIME	*time the command started*
TTY	*terminal from which the process was started*
TIME	*cumulative execution time for the process*
COMMAND	*name of the process started*

Example 5.8

To get the process status of the processes executed from the current terminal

```
$ ps
PID    TTY           TIME   CMD
1077         pts/2          0:11   ls
```

```
1070         pts/2        0:00   sh
1072         pts/2        0:00   jsh
```

To get the process status of all the processes of the system

```
$ ps  -ef
             UID    PID    PPID   C      STIME  TTY    TIME   CMD
       root   0      0      0      09:57:52     ?      0:00   sched
root   1      0      0      09:57:52     ?      0:01   / etc / init  -
root   2      0      0      09:57:52     ?      0:00   pageout
root   3      0      0      09:57:52     ?      0:09   fsflush
root   240    237    0      09:58:36     ?      0:00
       /usr/lib/saf/ttymon
root   106    1      0      09:58:15     ?      0:00   /usr.sbin/aspppd -d
root   209    1      0      09:58:34     ?      0:00   /usr/lib/utmpd
oracle 1083   1082   1      17:58:13     ?      0:01   oracleWG73
(DESCRIPTION = (LOCAL=YES)  (ADDRESS=  (PROTOCOL=beg)  ) )
oracle 1082   1057   38     17:57:58     pts/0  0:18   imp
plokam 1077   1072   18     17:57:38     pts/2  0:12   ls  -Rl
plokam 1068   1048   0      17:56:17     pts/1  0:00   script
plokam 1070   1069   0      17:56:17     pts/2  0:00   sh  -i
oracle 308    1      0      11:26:38     ?      0:01   ora_pmon_WG73
```

To get the process status of the processes run by a particular user

```
$ ps  - u oracle
          P            TTY          TI           CMD
    ID                              ME
    1083               ?            0:           Orac
                                    01           le
    1082               pts          0:           Imp
                       /0           21
    308                ?            0:           orac
                                    01           le
    310                ?            4:           orac
                                    49           le
    312                ?            4:           orac
                                    05           le
    314                ?            0:           orac
                                    01           le
    316                ?            0:           orac
                                    00           le
    1057               pts          0:           sh
                       /0           00
```

To get the process status of the processes run from a particular terminal

```
$ ps   -t     pts/0
    PID             TTY             TIME            CMD
    1082            Pts/0           0:23            Imp
    1057            Pts/0           0:00            Sh
    942             Pts/0           0:00            sh
```

Note: *A? mark in the TTY column represents that the process is started automatically and there is no controlling terminal.*

6.2.10. Kill

If a process is running in background and for some reason you want to terminate that process, then kill command can be used to accomplish the task. There may be some process which is not responding to nay user interrupt, that needs to be terminated.

Syntax

```
kill   - 9 pid
```

Where **pid** *is the process id number*

Example 5.9

```
$ ls– R / >   /tmp/out   &
[1]         1098
$
$
$ ps

            PI                    TT                    TI                    C
            D                     Y                     M                     M
                                                        E                     D
            10                    Pt                    0:0                   ls
            98                    s/                    0
                                  2
            10                    Pt                    0:0                   sh
            70                    s/                    0
                                  2
            10                    Pt                    0:0                   jsh
            72                    s/                    0
                                  2
$          kill           1098
$          ps

            PI                    TT                    TI                    C
            D                     Y                     M                     M
                                                        E                     D
            10                    pt                    0:0                   sh
            70                    s/                    0
                                  2
            10                    Pt                    0:0                   Js
            72                    s/                    0                     h
                                  2
[1]        +         Terminated                       ls  -R  /  1>/tmp/out
```

When a process is placed in background, its PID number is immediatedly displayed. To know the PID number of the process to be terminated, you can use the ps or jobs –1 command.

Note: *Only the owner of the process or the supersuer can kill it. Killing the shell will logout the user from the system.*

6.2.11. nohup

A background process can continue its execution ever after the user has logged out. Normally, all processes terminate at the time of logout. But if we start the background process with a nohup command, the background process can still be continued.

The nohup utility invokes the named command with the arguments supplied. When the command is invoked, nohup arranges for the Hangup signal which intimates the process to hangup, to be ignored by the process. nohup can be used when it is known that command will take a long time to run and the user wants to logout of the terminal; when a shell exits, the system sends its children hangup signals, which by default cause them to be killed.

6.2.12. bash shell – History Commands

Using the pevious and next buttons at the command terminal you can move back and forth between the various commands.

There is also another neat way of looking up the list of all the commands types during a particular session. This can be done by typing the command called history to get a list of all the commands executed in this session.

Syntax

```
history
```

This will list all the commands that were executed during the current session.

Example 5.10

```
$ nohup  ls  -Rl  /  >  /tmp/out  &
[1]  1101
```

Exit the current session. Login as plokam again and you will find that that the command you gave with nohup is still running.

6.3. LAB EXERCISES

1. Explain the diff multiuser ad multitasking environment?
2. What is background process? How will you suspend it?
3. How will you switch between process?
4. What is the use of command?
5. What is the use of kill command?
6. Explain the differences between suspend and kill?

Chapter 7

7. UNIX COMMUNICATIONS

7.1. OBJECTIVE OF CHAPTER 7

The objective of this chapter is to learn about the communication facilities provided by UNIX operating system. UNIX allows the users to communicate with each other electronically using simple utilities. We discuss some them in this chapter.

In this chapter you will learn the following:

- ❖ Write command
- ❖ Mesg command
- ❖ Wall command
- ❖ Electronic mail

7.2. UNIX SYSTEM

Communication has become unanimous with UNIX system. UNIX system lets its users communicate with their fellow users through its various utilities. Users can communicate with each other if they are:

- Working on the same UNIX system.
- Working on remote UNIX systems.

In this chapter we focus on the communication facilities supported within a UNIX system.

7.2.1. wall

When you want to send a broadcast message to all users who have logged in, wall Command is used. It is used to warn all users, typically prior to shutting down the System or any such urgent message that requires the user attention.

Syntax

```
Wall    filename]
```

Wall reads its standard input until an end-of-file, usually ^d character. It then sends this message to all currently logged-in users preceded by:

Broadcast Message from ………..

If filename is given, then the message is read in from that file.

Example 6.1

```
# wall
Please logout immediately. The system is going to be shutdown.
-SysAdmin
^d
#
Broadcast Message from root (pts/0) on plokam Thu Jun 10
10:50:00 …
Please logout immediately. The system is going to be shutdown.
-SysAdmin
```

Note: *The sender must be super-user to override any protections the users may have invoked.*

7.2.2. write

The *write* command is used to communicate with another user who is currently logged in to the system. The message you send is directly displayed on the receiver's screen. He/she has the option to send back you the reply. So, it provides with a direct conversation between two users through their terminals.

Syntax

```
Write username [terminal]
```

Where **username** is the user (login)name of the user to whom you want to send a message.

Terminal is the terminal to which the message to be sent if the user has logged in from more than one terminal.

Example 6.2

To send a message to the user ajay who has logged in

```
$write ajay
```

The command tells the UNIX system that you want to start a conversation with the user ajay who is currently logged in.

If he is not logged in, the following happens:

```
Ajay is not logged on.
```

After initiating write, the system waits for you to type a message. Each line that you type get displayed on ajay's terminal. When you have finished typing your messaged, press Ctrl-d on the next line. This will terminate the write session and display the line <EOT> on ajay's terminal to tell him that you have finished writing.

```
$ write ajay
Hello !
How are you ?
^d
```

On ajay's terminal the following message is displayed

```
Message from plokam on plokam (pts/1) [Thu Jun 10 10:54:09]
Hello !
How are you ?
<EOT>
```

If he wants to reply, he can start writing to you by typing

```
$ write plokam
Hello
^d$

Message from ajay on plokam (pts/3) [Thus Jun 10 10:54:20] …
Hello
<EOT>
```

This way two users can start a direct conversation with each other.

If a user logged into more than one terminal, then the terminal also has to be specified on the command line.

```
$ who
root        pts/0       Jun 10 09:49        (80.0.0.65)
plokam      pts/1       Jun 10   09:50      (80.0.0.65)
ajay        pts/3       Jun 10   10:46      (80.0.0.65)
ajay        pts/5       Jun  10 10:47       (80.0.0.65)

$ write ajay
ajay is logged on more than one place.
You are connected to “pts/3”.
Other locations are:
Pts/5

Test
^d

$ write ajay pts/3

This is for ajay at pts/3
-Bye
^d
```

7.2.3. mesg

The mesg utility will control whether other users are allowed to send messages via write, talk (1) or other utilities to a terminal device.

mesg n denies access and mesg y grants the users access to your terminal. This is useful, for instance, if you do not want other users to disturb you while you are executing an important file. For this you can set the message mode to n (no).

```
$ mesg n
      $
```

denies any users to access your terminal. After sometime, if you have finished doing your work and you want the messages to come in, you can set the message mode to y (yes)

```
      $ mesg y
      $
```

If the mesg command is used without any arguments, it gives you the status of the message mode you are in. By default, the messages are allowed to come in.

```
      $ mesg
      is y
```

7.3. ELECTRONIC MAIL

Electronic mail gives you the capability to send messages, memos, or any tepes of documents to other users electronically, without using any paper. The main difference between sending a message to someone through electronic mail and the write command is that the latter requires the person be logged in at the time the message is being sent. With electronic mail, the mail is automatically kept by the system until the user issues the necessary command to read his/her mail. To send or receive mails you use the mail command.

7.3.1. Reading Mail

Every time you log in, the shell automatically checks to see if you have received any new mail. If you have, then the following message will be displayed:

```
You have mail
```

After you get this message, you would like to read your mail.

The mail can be read without typing any arguments to mail. This causes the *mail* program to display any mail that has been sent to you. You can then perform different commands to manipulate the messages. If you read a message, it is saved to the file **mbox** when you exit **mail**. Your *mailbox* (/var/mail/plokam) contains all new messages that are not read.

If you have a mail, a numbered list of messages is displayed on your screen.

```
[mohan@rad mohan] $ mail
Mail version 8.1 6/6/93.      Type ? for help.
“ /var/spool/mail/mohan” : 1 messages 1 new
```

```
> N  1  sriram@localhost.loc   Thu  Jan   2   03:06   14/469   "Hello !"
&
Message 1:
From sriram Thu Jan 2 03:06:09 1997
Date: Thu, 2 Jan 1997 03:06:09 +0700
From: sriram@localhost.localdomain
To: mohan@localhost.localdomain
Subject: Hello !

Hello Mohan! How are you? Met you long ago, how are you now.
Meet you soon. Till then bye.
Sriram.

&
At EOF
& q
Saved 1 messages in mbox
[mohan@rad mohan]$
```

Each line in the list contains the status of the message (Old, New), the sender of the message, delivery time, size of message in lines/bytes, and the subject. The senders may choose to write or not to write the subject. The > sign shows that this is the current mail. After the message list is displayed, you are prompted with a '?' mark. You can type any of the commands to manipulate the list of message:-

<Enter>	ignore this mail. The mail will still be there next time you read the mail
d [n]	delete nth mail. If n is not specified, it means the current mail
s filename	save the current mail to a file; next mail is displayed
?	print a list of commands
q	quit reading the mail; any unread mail will still be there next time the mail is read
x	quit reading the mail; any deleted mails will be restored.
N	n is a number stating that nth mail has to be read.

7.3.2. Sending Mail

When you want to send a mail to other users, the mail command can be used followed by

a list of users. A header is added to the mail you are sending. The header identifies you (sender), the date of creation, and a subject line. The message is then sent to each user's mailbox in the/usr/mail directory. The mailbox is named after the user (login) name. For example, ajay's mailbox will be /usr/mail/ajay.

To send a mail, you specify the names of users (in our case, ajay) as arguments to the mail command. When you specify usernames on the command line, the mail command knows that you want to send a message to recipients. After giving the command you are prompted for a subject on the next line (only first 40 characters are displayed on the screen).

```
[mohan@rad mohan] $ mail sriram
Subject: Reply – hello
I am fine here and hope the same with you. Whats up? Anything
Nw please do mail me.  See you soon.
Cc:  [mohan@rad mohan]$
```

After entering the subject, the mail command enters a simple editor or input mode. You enter your text line by line. To end the message, press ^d at the beginning of a lline.

7.4. LAB EXERCISES

1. What is the use of write command.
2. What condition should be met to hold conversation with another user.
3. How will you broadcast message to all the users?
4. How will you read new mails?
5. What is the usage of forward command?
6. When will you use reply and respond command?

Chapter 8

8. VI EDITOR

8.1. OBJECTIVE OF CHAPTER 8

The objective of this chapter is to learn about the file editing facilities provided by UNIX operating system. UNIX supports a variety of text editors. You will learn about the most commonly used text editor, vi. It has many commendable features which makes it a powerful editing too.

In this chapter you will learn the following:

- Creating simple text file
- Character positioning commands
- Line positioning commands
- File positioning commands
- Pattern matching
- Search / Replace
- Delete / Cut / Paste commands

Creating and editing text are probably the most common tasks you'll use in your day-to-day work. You make use of an editor to prepare a letter or a memo. UNIX has several editors : vi, ed, and emacs are the most common ones.

Vi stands for visual editor. Vi is used to create and edit ASCII files, which can be used in variety of situations – creating shell scripts, mail messages, source codes, editing UNIX systems files etc. Vi was developed by The University of California, Berkeley California, Computer Science Division, Department of Electrical Engineering and Computer

Science.

First, tell your shell what type of terminal you have. (If you're not sure what your shell is, type this command to see what shell you have: echo $SHELL.) For the examples given, the terminal type is "vt100". Substitute it with whatever terminal type you have. For C shell (/bin/csh), the command is this:

```
set term=vt100
```

For Bourne Shell (/bin/sh) or Korn Shell (/bin/ksh), the commands are the following:

```
export TERM
TERM=vt100
```

Next, reset your terminal with this command:

```
tset
```

Working in vi is a matter of working in two modes: insert and command. When you start vi, you are automatically in the command mode. Here, all your key strokes are interpreted as commands. To start entering text we have to switch to insert mode. A simple way to enter insert mode is to type 'i' in the command mode.

Let us create a text file using vi called text.1

```
$vi  text.1
This is the first line of the text.  This is just the starting
and there is quite a long way to go.
Just getting used to the editor.
This is enough for the present.
~
~
~
~
```

The insert-mode options are:

Command	Meaning
i	Insert at the current character
I	Insert at the beginning of the current line

a	Append to the right of the current character
A	Append to the end of the current line
o	Insert new line immediately following current line
O	Insert new line immediately before current line.

To get back to command mode press escape (ESC). Press colon (:). The cursor moves to the last line and a colon appears. Enter wq. The file text.1 is created and saved on to the disk under current directory and also you come out of vi.

You can efficiently use the editor only if you are able to move freely to various positions within the file. Vi provides you a with a variety of ways to move to the cursor position you want. To use the cursor position commands, you should be in the command mode and not in any other mode.

Cursor positioning commands can be classified as:

8.1.1. Character positioning

These commands help you in positioning the cursor within the same line. The following list gives the character positioning commands.

Command	Meaning
space	Move to the next character from the current cursor position
l	Same as <space> command
h	Move the cursor backward to the previous character
^	Move to the first non white-space character of the line
0	Move to the beginning of line even if it is space character
$	Move cursor to the end of line
w	Move to the first character of the next word
e	Move to the last character of the next word.
b	Move to the first character of the previous word.

8.1.2. Line positioning

These commands provide means of moving within the displayed screen without moving out. The following list gives the line positioning commands.

Command	Meaning
H	Move to the top line on screen
L	Move to the last line on screen
M	Position at the middle line on screen
j	Move to next line, same column
k	Move to previous line, same column
<Enter>	Move to the next line

8.1.3. File positioning

When you want to move freely within the file and position the cursor at any point, the above commands will not help. It will be difficult to move around a long file without enough commands that will help in positioning wherever you want. The following list gives the line positioning commands.

Command	Meaning
^F	Move to the next screen
^B	Move to the previous screen
^D	Scroll down half screen
^U	Scroll up half screen
^G	Go to the last line of the file

8.1.4. Search / Replace

You can position the cursor based on a particular pattern also. These commands uses the slash / character. As soon as you press / key, the cursor moves to the last line on the screen and waits for you to enter the pattern that needs to be matched. The following are the pattern matching commands:

Command	Meaning
/pat	Moves forward looking for the next occurrence of the pattern pat in the file
?pat	Moves backwards looking for the previous occurrence of the pattern pat in the fiel.
n	Repeat last / or ? command. It simplifies retyping the command
N	Reverse last / or ? command. It simplifies retyping the command.

Suppose you have given / as the last pattern match command, pressing N will move backwards instead of forward.

After searching for a particular pattern you can replace the character using the following commands:

Commands	Meaning
rx	Replace one character with the single-byte character x
RtextESC	Replace characters present by text characters
~	Change lower case character to upper case and vice-versa

8.1.5. Delete / Cut / Paste

vi editor also provides the facility to delete one or more characters present in the file and also to copy / cut a set lines of lines and paste them at another position and also allows you to undo or redo the previous command.

Command Meaning

x	Delete the current character
dw	Delete the current word
dd	Delete the current line
D	Delete the rest of the line from the current cursor position
c	Change the current character to the newly typed character
C	Change the rest of the characters in the line.
YY	Yank a line (copy)
p	Put the line that was cut / yanked previously (paste)
.	Redo (repeat) the last command
u	Undo the last command.

8.1.6. Exiting vi

Command	Meaning
:w	Save the changes of file
:w <filename>	Save the file with current changes to the name filename
:q	Quit after file is saved. Otherwise refuses to quit
:q!	Quit without saving the file
:x	Save and Quit
:wq	Save and Quit
ZZ	Save and Quit

Executing Unix Command in vi editor

: ! <unix commands>

Example 7.1

: ! ls –x

: ! cp text1.txt   text2.txt

8.1.7. Editing another file from vi editor

: e <filename>

Example 7.2

: e text1.txt

8.1.8. Setting number using vi editor

: set number

To locate the error using vi editor

: <line no>

or

$ vi +<line no> <filename>

8.2. LAB EXERCISES

1. Which command is used to edit a file in?
2. Which key is used to move the cursor up?
3. Move the cursor left, down and up using respective cursor commands.
4. How will you scroll through a file containing 10 pages.
5. What are the file commands will you quit a file without saving the charges.

Chapter 9

9. SHELL PROGRAMMING

9.1. OBEJCTIVE OF CHAPTER 9

The objective of this chapter is to teach you shell programming. Here you get to know that the shell is not just a command interpreter but a programming language also. You are taught the features of shell programming. By using these features, you can automate many repetitive functions, which is, of course, of any computer language.

In this chapter you will learn the following:

- Types of shells
- Simple Shell program
- Shell variables
- Environment variables
- Local and global variables
- Passing arguments
- Arithmetic operations
- Branching statements
- Test command
- Looping statements
- Trap command

A part from being a command interpreter, the shell is also an interpretive programming language. It reads the input from the keyboard, interprets the command and makes a request to the kernel to take action. As a programming language, the shell lets you store commands in a file and execute the file as a program.

Features of shell as a programming language are:

Scripts: The shell lets you store shell commands in a file and execute the file as a program, known as a shell script.

Variables: Values stores in names variables can be used in the scripts.

Structured loop construct: Shell implements programming language features like looping and branching.

Command substitution: The shell executers the command and substitutes the output of the command in place of the command itself.

9.2. TYPES OF SHELLS

Power and flexibility of UNIX system lies in the fact that it is not tied to one particular command interpreter. Since, shell is also a program, several types of shells have been evoled. Each differ slightly in the features it provides and in the syntax of the command language. Currently there are five popular shells:

Bourne shell(sh)

This is the original Unix shell written by Steve Bourne of Bell Labs. It is available on all UNIX systems.

This shell does not have the interactive facilities provided by modern shells such as the C shell and Korn shell. You are advised to use another shell which has these features. The Bourne shell des provide an easy to use language with which you can write shell scripts.

C shell (csh)

This shell was written at the University of California, Berkeley. It provides a C-like language with which to write shell scripts – hence its name.

TC shell (tcsh)

This shell is available in the public domain. It provides all the features of the C shell together with emacs style editing of the command line.

Korn shell (ksh)

This shell was written by David Korn of Bell labs. It is now provided as the standard shell of Unix systems.

It provides all the features of the C and TC shells together with a shell programming language similar to that of the original Bourne shell.

It is the most efficient shell. Consider using this as your standard interactive shell.

Bourne Again Shell (bash)

This is public domain shell written by the Free Software Foundation under their GNU initiative. Ultimately it is intended to be a full implementation of the IEEE Posix Shell and Tools specification. This shell is widely used within the academic community.

bash provides all the interactive features of the C shell (csh) and the Korn shell (ksh). Its programming language is compatible with the Bourne shell (sh).

If you use the Bourne shell (sh) for shell programming consider using bash as your complete shell ev=nvironment.

9.3. Check your Environment:

Make sure that the path include the current working directory***(".")***. Use the following command to check if the Path includes the current working directory(".").

```
[plokam@Linux1 ~]$echo $PATH
```

If the path does not include the Current Working Directory then add the "." To the PATH and export the Environment Variable. Open the ~/.bash_profile and add the following two lines to the end of the file and then execute the source command.

```
[plokam@Linux1 ~]$vi ~/.bash_profile
PATH=$PATH:.
Export $PATH
[plokam@Linux1 ~]$ source ~/.bash_profile
```

9.4. Writing your first shell program

To write a simple shell program, usually called a shell script you open the vi editor along with the filename, say sample

Shell Script 1: prog1

```
[plokam@Linux1 ~]$ vi prog1
#!/bin/sh
# This is a comment!
echo Hello World    # This is a comment, too!
```

"prog1" is a simple and straightforward command.

To make the file executable, the following command can be used: -

```
[plokam@Linux1 ~]$ chmod  u+x  prog1
```

Once the execute permission is granted, it can be executed simply by typing the name of the file at the $ prompt.

```
[plokam@Linux1 ~]$ prog1
Hello World
```

Alternatively, if the execute permission is not set, a file can be executed by using the sh command:

```
[plokam@Linux1 ~]$ source prog1

Hello World
```

Note: In the beginning of the chapter we discussed about different types of shells. Each shell uses a different syntax. Hence, what works in one shell, like sh, may error out in another shell, like csh. To ensure that the script that you have written is executed in the right shell environment, there is a special magic syntax. This magic line is placed at the very top of the file, in the first line. To use the bourne shell, sh, insert the following line at the start of your shell scripts:

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

9.5. SHELL VARIABLES

The shell provides with the capability to define variables and assign them to values. A variable name is a sequence of characters beginning with a letter or an underscore.

Values can be assigned to shell variables using the following format, having no spaced on either sides of the assignment operator:

Syntax

Variable_name-value

There is no type associated with a shell variable. Every value that is assigned to a variable is treated as a string of characters by the shell.

Note: There must be no spaces around the "=" sign: VAR=value works; VAR = value doesn't work.

Example 8.1

If the value is a string of characters containing embedded spaces, it must be enclosed within a pair of double quotes. If the string does not contain white spaces, the quotes are not necessary.

```
$myname="Rahul Sharma"
$myname=Gaurav
$myage=21
```

Having assigned a value to the shell variable, how do we go about accessing that value? To access the value stored inside the shell variable, the variable is immediately preceded by a dollar sign ($).

To display the value of the variable myname, using echo command, the following sequence can be used

```
$myname=Gaurav
$echo $myname
Gaurav
```

The presence of the dollar sign ($) in front of the variable name lets the shell substitute the value of the variable at that point.

```
$ myname="Rahul Sharma"
$ echo $myname
Rahul Sharma
```

9.6. SHELL INPUT WITH READ COMMAND

When writing shell programs, you might need to enter some data from the keyboard and assign it to the shell variables. The shell statement read solves this purpose.

Syntax

read var_name

Read command waits for the user to enter a value. The shell assigns that value to the shell variable specified by **var_name.** Consider a shell program copy which copies a file.

Shell Script 2: prog2

```
[plokam@Linux1 ~]$ vi prog2
echo "Enter the source file"
read source
echo "Enter the destination file: "
read destination
cp $source $destination
```

When you execute the program, it prompts the user to enter the name for the source file and the destination file. After the information is entered, the copy program called the cp command to perform the copy.

```
[plokam@Linux1 ~]$ chmod u+x prog2
[plokam@Linux1 ~]$ ./prog2
Enter the source file
john
Enter the destination file:
mike
$ ls- l john mike
-rw-r--r--    1 plokam   plokam         5  Dec  14  10:11  john
-rw-r--r--    1 plokam   plokam         5  Dec  14  10:18  mike
```

9.7. PASSING ARGUMENT TO SHELL PROGRAMS

Arguments greatly increase the flexibility and usefulness of nay program. You can have your shell programs take the arguments same way other UNIX commands take them. When the program is executed, you simply list the arguments on the command line.

If your program name is prog1 which takes arguments, it is executed as:

```
$prog1 cat bat sat
```

where ***cat, bat, sat*** are the arguments passed to the program myscript.

9.7.1. Positional parameters

Positional parameters are numbered variables representing the position of the arguments passed to the shell script. The name of the shell script is passed as positional parameter 1, and so on. UNIX shell creates a maximum of ten positional parameters indicated by $0, $1 ……….$9. Depending on the number of arguments specified in the command, the shell assigns values to these positional parameters.

Shell Script 3: prog2

```
[plokam@Linux1 ~]$ vi prog2
echo  $1  $2  $3  $4

[plokam@Linux1 ~]$ chmod  u+x prog2
[plokam@Linux1 ~]$ ./prog2 cat  bat  sat
cat  bat  sat

prog2forms positional para meter $0, cat as $1, bat as $2, sat as $3
```

9.7.2. $# and $*

$# - No. of Arguments

$* - is set to all the arguments that were passed.

Parameters can be used to create new commands or programs which work like UNIX commands.

Shell Script 4: prog4

```
[plokam@Linux1 ~]$ vi prog4
echo   "The name of the program is $0"
echo   "The first argument is $1"
echo   "The second argument is $2"
echo   "The third argument is $3"
echo   "The number of argument is $#"
echo   "The arguments are $*"
```

Execute the program sample:

```
[plokam@Linux1 ~]$ sh prog4 cat bat sat
The name of the program is prog4
The first argument is cat
The second argument is bat
The third argument is sat
The number of argument is 3
The arguments are cat bat sat
```

9.8. QUOTING

Quoting provides a way to override special meaning of certain characters. If a character is quoted, the shell does not interpret its special meaning.

There are four types of quotes:

Backslash

Double Quotes

Single Quotes

Back Quotes

9.8.1. Backslash (\)

Each special character may be preceded by a backslash to escape its special meaning.

Example 8.5

Assign a value Anita to the variable myname and display the value of myname

```
$  myname =Anita
$  echo            $myname
Anita
```

The dollar sign $ loses its special meaning when preceded by \. $myname is displayed as one parameter

```
$ echo        \$myname
$myname
```

Even a backslash can be preceded by a backslash to remove its special \ meaning

```
$ echo \\
\
```

9.8.2. Double Quotes(“”)

Double quotes allow commands substitution but escape the meaning of all other special words and characters. All text between double quotes is considered as one parameter. The backslash has no special meaning inside double quotes

Example 8.6

Suppose your directory contains the following files:

$ pwd

/user/plokam/personal

$ ls

```
file1     file2       salary
```

Consider the output of two echo commands. In the first echo commands * to zero or more characters. In the second echo command, * loses its meaning of expanding to characters. Therefore no filename substitution takes place between the pair of double quotes.

```
$ echo
file1     file2       salary
```

```
$ echo"*"
*
```

Double quotes can be used for variable as well as command substitution also. Here the double quotes substitutes the value of the shell variable $day

```
$ day=Friday
$ echo  "The day is $day"
The day is Friday
```

Double quotes also preserve white space characters by removing their special meaning to the shell.

```
$ echo "white space characters are preserved within the double quotes."
White space characters are preserved within the double quotes.
```

9.8.3. Single Quotes (``)

All characters inside single quotes are interpreted with no special meaning, when a shell variable is enclosed within single quotes, its value is not substituted by the shell.

Example 8.7

In the first two commands, the variable has been substituted with its value. In the third command, the quotes simply ignore the meaning of $

```
$ echo   $day
Friday

$ echo "$day"
Friday

$ echo   '$day'
$day
```

No processing of characters takes places within the single quotes.

9.8.4. Back Quotes (` `)

Back quotes are also called grave accents. When a command is enclosed within a pair of back quotes, the command gets executed and its output is inserted at that precise point. This is called command substitution.

Example 8.8

When the shell processes the command, it notices the back quotes and proceeds to execute whatever command (pwd here) is there inside the quotes. The output of the pwd command is substituted at that point and then the echo command is executed.

```
$ echo My current working directory is `pwd`
My current working directory is /user/plokam/personal
```

You can even have a pipeline executed in the similar fashion. The whole pipeline must be enclosed within the back quotes.

```
$ echo There are who | wc –1` users logged in
There are 2 users logged in
```

Back quotes have the capability to assign the output from a command to a shell variable:

```
$ today=`date`
$ echo $today
Mon Jun 14      10:27:30 GMT 1999
```

The difference have the capability single quotes, doubles quotes, back quotes and backslash can be seen from the following example:

```
$ cat sample
# script to compare the various quotes
# Explained with echo command

# Expand the meta character  *
echo *
# * within single and double quotes
echo    *   "*"    '*'

# use a variable
prompt="hello"
echo $today,      "$today",'$today'

# use a command
echo "the current directory is `pwd`"

# set a variable to command output
today=`date`
echo    $today,               "$today",'$today', \$today
$

$ sh sample
```

```
file1    file2    salary sample
file1    file2    salary sample    *        *
‘ ‘ $today
the current directory is /user/plokam/personal
Mon Jun 14:08:30 GMT 1999, Mon Jun 14 14:08:30 GMT 1999
$today, $today
```

9.9. ENVIRONMENTAL VARIABLE

In addition to user-defined variables, shell also has special variables called environments variables. By changing the values of these variables, a user can customize the environment of the system. All environmental variables are written in uppercase only.

Some of the environmental variables are:

HOME	When a user logs in, he/she is taken to the corresponding home Directory. The home directory is specified by assigning a value to the variable HOME.
PATH	contains a list of full path names of directories to be searched for An executable program. These names are separated by a colon. For Example :- PATH=/bin:/usr/bin:/usr/etc
LOGNAME	this variable contains the user’s login name. This cannot be Changed.
PS1	stores the system prompt symbol (\lquote $\rquote). This Symbol can be changed by assigning a new value to this Variable. For example, $PS1=”hello>” will change the system Prompt to hello>.
SHELL	stores the name of the shell

9.9.1. Knowing your profile file

This file is not listed when you use ls command because the dot (.) preceding the filename tells that it is a hidden file. To list this file, you can use the –a option with the ls command. Some of the environment variables are automatically set when you log in, like LOGNAME and HOME. This file contains the settings of environment variables. If you want to make some changes to the settings of your environment, you use the .profile file for the purpose. Every time you log in, the shell looks for this file in your HOME directory.

A .profile file may look like this:

```
$ cat  .profile
sty erase
HOME = /user/plokam
PATH = $path: /usr/openwin/bin
Export PATH
```

```
umask 022
trap  2  3
```

9.9.2. Exporting Variables

When you create a variable, it is by definition a shell or local variable, which means it is available for use by the shell and not to your UNIX applications. If you want to use these variables inside you UNIX applications, they should be made global. This is done by explicitly telling the shell to export the variables to all the programs. The variables are exported using the command ***export.***

Syntax

Export [variable]

Where **variable** is the variable to be placed in the shell environment.

<u>Example 8.9</u>

* Assign a local variable z a value of 100. Start another process. The new process does not know about the variable z assigned in its parent shell.

```
$  z = 100
$  echo  $z
100
$   sh
$   echo  $z

$ exit
```

* Now let us make the variable global so it is available to all processes. The value of z is available to the subprocess. Let us assign another value to z within the sub process. The local variable z precedes the global variable z.

```
$  export  z
$  sh
$  echo  $z
100
$  z  = 200
$  echo  $z
200
$  exit
$  echo  $z
100
```

Local variables are available only during the lifetime of the process. They get destroyed with the process. Global variables are available throughout the shell session. When a local as well as the global variable has the same name, the local variable precedes the execution of the global variable, as seen in the above example. If you use export command without any arguments, it lists out all the exported variables, if you are in a subshell, only the variables that have been exported in the subshell are listed.

```
$  export
export   z
```

9.10. ARITHMETIC OPERATIONS WITH SHELL VARIABLES

Since the shell variables are treated as characters, a different mechanism is adopted to perform arithmetic operations on the shell variables. Consider the following:

* Assign 0 to the variable count and display it.

```
[plokam@Linux1 ~]$ count=0
[plokam@Linux1 ~]$ echo  $count
0
```

* Now let us add 1 to count

```
$  echo  $count  +  1
0  +  1
```

The output is not what we expected because a shell does not recognize 0 and 1 as numbers but as strings. Therefore we do not get the output as expected by us i.e. 1.

9.10.1. expr

To overcome the problem of handling arithmetic expressions, there is a utility called expr that lets you perform calculations with variables. Consider the following commands.

```
$  expr  $count  +  1
1

$ expr  5  *  10
expr : syntax error

$ expr 5 \* 10
50
```

*The special meaning of *is removed by preceding it with a backslash(\)*

The expr command evaluates the expression given by its arguments and write the result to standard output. By using backquotes, you can assign expr's result to a shell variable.

```
$ result=`expr 1 + 2`
$ echo $result
3
```

Note: The Quotes are Back Quotes

Using expr command we can also execute the the following expression

```
$ expr \( 1 + 2 \) \* 10
30
```

9.11. CONDITIONAL COMMANDS

The shell provides various commands to allow for structured programming. These include conditional, iteration, and grouping commands.

Conditional commands allow you to execute a list of commands based on the result of testing and expression.

9.11.1. if

Syntax

```
If [ condition ]
then
```

```
                statement
            fi
```

If the value of condition of TRUE i.e. 0, the commands between then and fi statements are executed.

Example 8.10

The following sequence displays the message Hello, Gaurav if the value of the variable myname is equal to Gaurav

```
[plokam@Linux1 ~]$ vi myfile
if [ $myname = Gaurav ]
then
      echo Hello, Gaurav
fi

[plokam@Linux1 ~]$ myname=Gaurav
[plokam@Linux1 ~]$ export myname
[plokam@Linux1 ~]$ ./myfile
Hello, Gaurav

[plokam@Linux1 ~]$ myname=gaurav
[plokam@Linux1 ~]$ export myname

[plokam@Linux1 ~]$ chmod u+x myfile
[plokam@Linux1 ~]$ ./myfile
```

Consider a small program, named search, which searches for a specified pattern in a list of files and echoes it along with the line number and the names of the files.

```
[plokam@Linux1 ~]$ vi search
echo Enter the string to search for:
read string

echo Enter the filename:
read file

if grep –n $string $file
then
      echo $string is in $file
fi

[plokam@Linux1 ~]$ sh search
Enter the string to search for:
Command file1
4 : cat command.  It can also be used to view
command is in file1
```

9.11.2. IF –ELSE –IF

Syntax

```
If [ condition_command ]

then

      command_list1
```

```
else

        command_list2

fi
```

If the condition_command of the if statement is not met, we may like to display some other message. In that case the else statement of the if construct is executed. But is always only on set of commands, either commands_list1 or commands_lists2 that is executed, never both.

Shell Script 1: prog1

Example 8.11

```
[plokam@Linux1~]$   vi myfile2
if [ $myname = Gaurav ]
then
        echo Hello, Gaurav
else
        echo, it is not Gaurav
fi

[plokam@Linux1 ~]$ myname=Raghu1
[plokam@Linux1 ~]$ sh ./myfile2
sorry, it is not Gaurav
```

In the above example, since value of variable did not match Gaurav, the commands between then and else are ignored and the commands between else and fi are executed. In the simple words, it's line having two options – if one this doesn't happen, do the other thing

Note: The search program above echoed a message if the string was found in the files. However in case of failure, no action was initiated. To display the relevant failure message, we can make good use of the "else" construct as shown below.

```
Echo Enter the string and the filename
read string file
if grep –n "$string" $file
then
        echo $string is in $ file
else
        echo $string does not exist in $file
fi
```

9.11.3. IF-ELIF-FI

Syntax

```
if command_condition

then

        command_list1

elif  command_condition

then

        command_list2

else

        command_list 3

fi
```

Shell Script 1: prog1

Example

```
$cat myfile3
if [ $myname =Plokam  ]
then
             echo Hello, Plokam
elif [ $myname = Ajay ]
then
             echo hello, Ajay
else
             echo Failure again
fi

$ myname=Gaurav
$ export myname
$ sh myfile 3
Failure again

$ myname=Plokam
$sh myfile3
Hello, Plokam

$ myname=Ajay
$ sh myfile3
Hello, Ajay
```

The if construct can be nest to my level. Consider a program name check_file, that checks for the presence of the file in the current directory, after which it determines the read permission of the file. Appropriate messages are generated for the absence of each of these conditions:

Shell Script 1: prog1

```
$ cat check
echo Enter the name of the file
read filename
if [ ! -f $filename ]
then
             echo $filename does not exit
elif [ ! -r $filename ]
       then
             echo the file does not have read permission
       elif [ -f $filename -a -r $filename ]
             then

                    echo The file is OK!
       fi

$ 1s- 1 file?
-rw-r—r--    1 plokam     plokam       132 may 17 12:19 file1
-rw-r—r--    1 plokam     plokam       110 may 17 12:19 file2

$ sh check
Enter the name of the file
file1
The file is OK!

$ sh check
Enter the name of the file
file 2
the file does not have read permission
```

```
$ sh check Enter the name of the file
file 3
file3 does not exit
$
```

Example

The following programe serves as an example to find the biggest among 3 numbers.

```
$ cat greatest
echo "enter the value for a"
read a
echo "enter the value for b "
read b
echo "enter the value for c"
read c
If [ $a -gt $b -a $a -gt $c ]
then
      echo "$ a is the greatest"
elif [ $b- gt $c ]
      then echo "b is the greatest "
else
      echo "$c is the greatest "
fi

$ sh greatest
enter the value for a
2
enter the value for b
7
enter the value for c
1
7 is the greatest
```

The following program finds whether a given number is Odd or even.

```
$ cat odd
echo "enter the number\\c"
read num
if [ ' expr $num % 2 ' -eq 0 ]
then
      echo the given num is even
else
      echo the given num is odd
fi

$ sh odd
enter the number 5
the given num is odd
```

9.11.4. TEST

The test command is used to evaluate conditional expressions. If the expression is true, test command returns a return code of 0; if the expression is false, the return code is 1. The test command is normally used with the if statement to check the exit status of the conditional expression. If the test succeeded, the body of the if statement is executed, otherwise not.

The following operators are used with the test command:

9.11.5. Operators used with numeric variables

-eq	:	equals to
-ne	:	not equal to
-gt	:	greater than
-lt	:	less than
-ge	:	greater than or equal to
-le	:	less than or equal to

Shell Script 1: prog1

```
Example 8.13

$ who
root          console      Jun 15 09:41
root          pts/0        Jun 15 09:12        (80.0.065)
plokam        pts/1        Jun 15 09:12        (80.0.0.65)

$ echo $num
3

$ num='who | wc -1'

$ if test $num- le 3
> then
>      echo " The number of logged in users is very less"
> else

>      echo " The number of logged in users is good"
> fi
  The number of logged in users is very less
```

9.11.6. Operators used with string variables

=	:	equality of strings
!=	:	not equal
-z	:	null string
-n	:	non-zero string

Shell Script 1: prog1

```
$ day1=Friday
$ day2=tuesday
$ if test "$day1" = "$day2"
> then
>      echo "Both the days are same"
> else
>      echo "The days are $day1, $day2"
> fi
The days are Friday, Tuesday
$
```

9.11.7. Operators used with files

-f	: file exists and has execute permission
-s	: file exists and is not empty
-d	: directory exists
-r	: file exists and has read permission
-w	: file exists and has write permission
-x	: file exists and has execute permission

Shell Script 1: prog1

Check if the user's home directory exists. If yes, 0 is returned

```
$ test –d "$HOME"
```

The test command does not display any output. It just checks the exit status of the expression. To display the exit status $? Can be used with the echo command. $? sets the exit status of the expression.

```
$ echo $HOME
/user/plokam
$
$ test –d "$HOME"
$ echo $?
0
```

9.11.8. Logical operators

Two or more conditions can be combined together with the help of logical operators.

-a	:	logical AND, implies that both the conditions must be met
-o	:	logical OR, implies any of the condition must be met
!	:	logical NOT, implies the condition is negated

Example 8.16

Check the read and execute permissions of the file mybook. The result is true only if both the expressions are true. The result is available in $?

```
$ test –r "mybook" –a –x "mybook"
$ echo $?
l
$
```

9.11.9. CASE

It is used to perform multiple choice branches. The case statement is best suited to compare a value against a series of values.

Syntax

case value in

```
        pattern 1 ) command_list ;;
        pattern 2 ) command_list ;;
        . . .
        pattern n ) command_list ;;
esac
```

Operation of the case statement proceeds as fallows: value is successfully compared against pattern 1, pattern 2, ……, pattern n. As soon as a match is found, the command_list after the pattern is executed, until a double semicolon (;;) is reached. At that point, the case statement is terminated. If no match is found, no action is taken and the entire case is effectively skipped.

Shell Script 1: prog1

The following example displays the menu options for inventory management

```
$ cat invent

echo "1.Add Items"
echo "2.Delete Items"
echo "3.Update Stock"
echo "Enter Your Choice (1,2 or 3)      :\c"
read mychoice
case $mychoice in
1)
echo "The Choice is Add Items";;
2)
echo "The Choice is Delete Items";;
3)
echo "The Choice is Update Items";;
*)
echo "Not a Valid Choice"
esac

$ sh inven
Add Items
Delete items
Update Stock
Enter your choice (1,2 or 3)      :  4
Not a valid choice

$
$ sh inven
Add Items
Delete items
Update Stock
Enter your choice (1,2 or 3)      :  3
The choice is Update items
$
$ sh inven
Add Items
Delete Stock
Update Stock
Enter your choice (1,2 or 3)      :  2
The choice is Delete items
$
```

Note: **) is used to give a message if no choice is made or a wrong choice has been made.*

The following program used case to add/multiply 2 numbers based on the option selected.

```
$ cat trial
echo " enter the choice"
read choice
case $choice in
1) echo "Add"
echo "enter the number"
read a b
echo '  expr $a + $b   ';;
2) echo "multiplication"
echo " enter the numbers"
read x y
echo '  expr $x \*  $y  ';;
*) echo "Not a valid choice";;

esac

$ sh trial
enter the choice
2
multiplication
enter the numbers
2  7
14
```

9.12. ITERATION COMMANDS

Iteration commands let you execute a list of commands while an expression remains true. When the expression becomes false, the loop is terminated.

FOR

The for loop executes a specific number of times based on the number of words in its word_list. Once the number of words in the word_list get exhausted, the loop automatically ends. It provides a simple way to step through lists of information.

Syntax

```
for variable in argl arg2 . . . argn

do

        command

done
```

For each interaction of the loop, the variable name, variable, supplied on the for command line assumes the value of the next word in the argument list. The commands between do and done are executed as many times as the number of arguments.

Shell Script: prog8

```
[plokam@Linux1 ~]$  cat prog8
for index in 1 2 3 4 5
do
       echo $index
done

[plokam@Linux1 ~]$  sh prog8
1
2
3
4
5
```

In the above example, index is assigned the first value in the list i.e. 1. The body of the loop, echo command is then executed and it displays 1 on the terminal. After this, the next value 2 is assigned to the index and echo command displays it on the terminal. Likewise, one by one all values are assigned to the index and the body of the loop is executed till all the values are displayed.

Another example of the for loop can be to move all files from the current directory into a new directory which you specify:

Shell Script: backup

```
$ cat backup
echo "Enter the destination path: \c"
read path
echo "Backup Started"
for file in 'echo'

do
       echo $file
       cp -r $file $path/$file
done
echo "Backup at $path Over"

$ sh backup
Enter the destination path:  . . /newpersonal
Backup Started
Add
Backup
Bheck
Delete
File1
File2
For_samp
Inven
Myfile
Myfile2
Myfile3
Salary
Sample
Search
Typescript
Update
Backup at . ./newpersonal Over
```

9.12.1. WHILE

The while loop provides more control over command processing. It can be used to perform simple number control loops, process commands if another command it true. As long as the first command list returns a return exit code the while loop wile continue to execute.

Syntax

```
While condition
Do,
        Command_list
Done
```

The condition is tested first when the while loop starts. If the condition is true, the command list between do and done is executed. Each time after the command list is executed, condition is again tested. The command list between do and done will continue to be executed until the condition proves false. At this time the loop terminates.

Shell Script: prog10

In this example, a variable count is assigned a value 1 and this value is tested against 10. If it is less than or equal to 10, the body of the loop gets executed i.e. first the echo command displays the value of the count. Secondly, the value of the count if being incremented. Now the value 2 is test against 10. If it is less or equal to 10, the body is executed. Like this, when the value of count reaches 11, the condition no longer holds true and the loop terminates.

```
$ cat prog10
count=1
while [ $count -le 10 ]
do
      echo $count
      count=`expr $count + 1`
done

$ sh prog10
1
2
3
4
5
6
7
8
9
10
```

This program takes names as input as long as the user presses ^d. The names are redirected to the file names_list. After the loop ends, the contents of the file names-list is displayed.

Shell Script: prog11

```
$ cat prog11
echo "Enter the names"
echo "When finished type ^d"
if [- f name_list ]
then
rm name_list
fi
while read name
do
        echo $name >> name_list
done
echo name_list names:
cat name_list

$ sh prog11
Please enter each person’s name
When finished, type ^d
John
Mike
Rita
Jamie
Lisa
Name_list cotains the following names:
John
Mike
Rita
Jamie
Lisa
$
```

<u>Shell Script: prog12</u>

```
$ cat prog12
# this program is used to display the numbers between 1 to 100
# that are divisible by 7
       i=1
       while test $i –le 100
       do
              if test ‘expr $i % 7` -eq 0
              then
                     echo $i
              fi
              i= ‘expr $i + 1’
       Done

$ sh prog12 7
7
14
21
28
35
42
49
56
63
70
77
84
91
98
```

Shell Script: prog13

The following program is used to the pascal's triangle.

```
$ cat prog13

i=1
while test $i -le 100
do
        if test `expr $i % 7` -eq 0
        then
                echo $i
        fi
i=`expr $i + 1`
done

$ sh prog13

1
1      2
1      2      3
1      2      3      4
1      2      3      4      5
```

9.12.2. BREAK

Break statement can be used to come out of a loop. The further execution of the loop stops and the next statement after the done statement is executed. This may be useful, for instance, if we want the while loop to execute indefinite number of time. If we do not provide with any mechanism, the execution will remain in loop forever and probably, the system may hang. To come out of such situations, we use the break statement.

Shell Script: prog14

```
$ cat prog14
while true
do
       echo "enter file name \c"
       read file
       if [ ! -f $file ]
       then
                     echo "File not found, Quiting"
                     break
              else
                     echo "File found, find the next one"
              fi
              echo "Bye"
       done
$

$  sh prog14
enter file name file1
File found, fine the next one
Bye
Enter file name file4
File not found, Quiting
```

9.12.3. CONTINUE

contine statement, line break, causes the execution to come out of the loop. But unlike break which causes the execution to follow after the loop, continue causes the execution to restart from the loop.

Shell Script: prog15

```
$ cat cont1
until false
do
       echo "Enter a number less than 1000 : \c"
       read num
       if [ $num- ge 1000 ]
       then
              continue
       else
              echo the number is $num
       fi
       done

$ sh cont1
Enter a number less than 1000 : 24
The number is 24
Enter a number less than 1000 : 1005
Enter a number less than 1000 : 100
The number is 100
Enter a number less than 1000 : 1008
Enter a number less than 1000 : ^c
```

Let's write a program that determines whether a given number is a prime number. This program illustrates the use of break and continue:

Shell Script: prog16

```
$ cat prog16
echo "Enter the number : \c"
read num
flag=0
k=2
q= 'expr $num % $k'
if [ $q –eq 0 ]
then
             echo $ num is not a prime number exit 2
else
             while [ $q –ne 0 ]
             do

                    k= 'expr $k + 1`
                    tem= 'expr $num \/ 2`
                    if [$k -1e $temp]
                    then
                           q=`expr $num % $k`
                           continue
                    else
                           flag =1
                           break
                    fi
             done
fi
```

```
if [ $flag –eg 1 ]
then
             echo $num is prime
else
             echo $num is not prime
fi

$sh prog16
Enter the number :13
13 is prime
$ sh prime
Enter the number :16
16 is not a prime number
```

9.13. TRAP COMMAND

When you are writing programs, you should keep in mind that during execution many things can happen that are not under the control of the program. The user of the program may press the interrupt key or send a kill command to the process, or the controlling terminal may become disconnected from the system. In UNIX, any of these events cn cause a signal to be sent to the process. The default action when a process receives a signal is to terminate.

Sometimes, however, you may want to take some special action when a signal is received. If a program is creating temporary data files, and it is terminated by a signal, the temporary data file remain. You can change the dafault action of your program when a signal is received using the trap command.

Syntax

trap command_string signals

Where Signals *are one or more signals whish are to be trapped*

command –string *is one more commands separated by semicolon which are to be executed when the signals are received.*

Shell Script: prog15

```
$   trap   "rm   /tmp/*$$;  exit "   1  2  15
$
```

The command informs the shell to remove the files that end with the PID at the temporary directory and exit the program when signals 1, 2 and 15 are received.

Signal	Description
1	Hangup
2	Operator Input
15	Software Termination (kill signal)

9.14. LAB EXERCISES

Shell Programming

1. Write a program to accept 2 integers and find their sum,difference and Product.
2. Write a program to find the greatest among 3 given numbers.
3. Write a program to find the greatest among ‘n’ given numbers.
4. Write a program to find whether a given number is odd ar even.
5. Write a program to find the factorial of any given number.
6. Write a program to display the multiplication table for any given number
7. Write a program to accept the rollno,name,marks in 3 different subjects for 10 Students and display those records for which the sum of the marks is above 60.
8. Write a program to check whether a given number is prime or not.
9. Write a program to display the prime number between 1 to 100.
10. Write a program display the fibonacci numbers till 200.
11. Write a program to find the following: 1!+2!+…+5!.
12. Write a program to calculate the employee net amount using switch case.

    Grade A: salary between 2000 to 3000

    Hra=30%,Pf=10%,Da=40%.

    Grade B: salary between 3000 to 5000

    Hra=40%,Pf=20%,Da=50%.

    Grade C: salary between 5000 to 10000

    Hra=50%,Pf=30%,Da=60%.

Chapter 10

10. AWK PROGRAMMING

10.1. OBJECTIVE OF CHAPTER 10

The objective of this class is to make you knowledgeable about programming in one of the powerful pattern scanning and processing languages in UNIX called awk. This language helps you in pattern matching and report writing based on the results of the matches. At the end of this class, awk language would come very handy to you to do quick data processing and report generation using data in ACII files.

In this class you will learn the following:

- Overview of awk
- Basic awk program
- Awk command files
- Readable awk programs
- Relational Operations
- Arithmetic Operations
- BEGIN and END
- Using FOR loops:
- Changing the Field Separator:
- Formatting awk Output:
- Redirecting awk Input and Output

10.2. OVERVIEW OF AWK

The awk utility is a powerful data manipulation programming language. It is used in the same manner as other UNIX tools. It's primary value is in the manipulation of structured text file, where information is stored in column form and information is separated by consistent characters such as tabs, spaces etc., known as field separators. awk allows you to select lines of input based on specified criteria (pattern searching), and then to taken certain actions on the data in those lines (processing).

As mentioned earlier, awk is treated by the command interpreter, the shell like any other UNIX utility. It can be invoked at the command prompt following the syntax given below:

Syntax

```
awk    commands   [Files]
```

awk scans each input file, present in the command line argument Files, for lines that match any of a set of patterns specified in the argument, commands. Patterns can be anything ranging from simple strings to complex arithmetic and logical expressions. With each pattern in commands there will be an associated action, which is also provided in commands, that will be performed when a line of a file matches the patter. The following sections will provide you mastery over awk programming.

10.3. BASIC AWK PROGRAM

Create a simple database file called *dbfile.*

```
$vi dbfile
Brian              22           Male         4000.00
Lisa               25           Female       7500.00
Paul               28           Male         10000.00
Charlie            19           Male         2000.00
Bill               23           Male         5500.00
Nancy              21           Female       3500.00
Tammy              25           Female       9000.00
$
```

Let us print all the records in the file using the following commands:

```
$ awk '{print}' dbfile
Brian              22           Male         4000.00
Lisa               25           Female       7500.00
Paul               28           Male         10000.00
Charlie            19           Male         2000.00
Bill               23           Male         5500.00
Nancy              21           Female       3500.00
Tammy              25           Female       9000.00
```

```
$ awk '{print $0}' dbfile
Brian              22           Male         4000.00
Lisa               25           Female       7500.00
Paul               28           Male         10000.00
Charlie            19           Male         2000.00
Bill               23           Male         5500.00
Nancy              21           Female       3500.00
Tammy              25           Female       9000.00
```

Inside the command script we use the reserved work **print** which prints the value of the variable to be printed followed by a new line. This is the maximum used work in the entire awk programming.

* To print the first field alone, use the following command:

```
$ awk '{print $1}' dbfile
Brian
Lisa
Paul
Charlie
Bill
Nancy
Tammy
```

* You can print the fields in any order. The command below prints third field followed by first field alone

```
$ awk  '{print $3,  $1}' dbfile
Male         Brian
Female       Lisa
Male         Paul
Male         Charlie
Male         Bill
Female       Nancy
Female       Tammy
```

* The primary feature of awk is pattern matching, let us learn about it. To get the list of the male members, use the following command:

```
$awk  '$3 ~ /Male/ {print $0}' dbfile
Brian              22          Male          4000.00
Paul               28          Male          10000.00
Charlie            19          Male          2000.00
Bill               23          Male          5500.00
```

Note: The names $0, $1, $2 etc. we have used so far are recognized by awk as variable names associated with specific value. These are called predefined variables. There are some more predefined variables that will be covered in the later sections.

10.4. AWK COMMAND FILES

Thus far you have entered all awk commands from the shell command line. You can also store a series of awk commands in a file and the use awk to excute a file.

* Create a command file ***awkprog.1*** containing the following instructions:

```
$ cat awkprog.1
/Female/ {print $1}
```

* To execute the command enter the following command line from the shell:

```
$ awk –f awkprog.1  dbfile
Lisa
Nancy
Tammy
```

Advantages

Keeping awk programs inside files, rather than entering them from the command line, is particularly useful as the programs become longer and the risk of mis typing increases.

Command files also make it possible to run the sam awk command over and over, and to keep record of what commands you have run.

10.5. READABLE AWK PROGRAMS

The awk utility has syntax rules that must be followed so that it will interpret your programs correctly. There rules specify to a certain exten how your awk programs are to be laid out. For example, awk syntax required that action statements should be surrounded by curly braces.

We can make the awk program more readable by following the tips discussed below:

* Instead of having all the commands in one line, they can be split and made more readable like:

```
/Female/  {
       print  $1
       }
```

in the file comm.1 instead of the statement

```
/Female/  {print $1 }
```

* Instead of using cryptic predefined awk variables like $0, $1 etc. we can define our own variables. awk programming assignment statements. This improves the program readability. Create a file comm.3 with following lines:

```
$  cat  awkprog.3
/ Female /  {
       name  =  $1
       age  =  $2
       print  name,  age
       }
```

* Include words in print statement to improve clarity and execute.

```
$ cat awkprog.3
/ Female /    {
       name$1
       age=$2
       print "Name is" name, "Age is" age
       }
$awk -f awkprog.3 dbfile
Name is Lisa Age is 25
Name is Nancy Age is 21
Name is TammyAge is        25
```

10.6. RELATION AND LOGICAL OPERATIONS

awk not only selects records based on the input file's contents but also provides another way of selecting records based on the position in the database.

* Create a new command file awkprog.2 that contains the following command

```
$ cat awkprog.2
NR  ==4  {print}
```

* Execute the following command

```
$ awk -f awkprog2 dbfile
Charlie       19           Male          2000.00
```

Examining the dbfile revals that the line selected is the fourth line, the fourth record in the file. The notation = = is the relation "equal to" operator and the whole program means: if the current record is the 4th record then perform the specified action, namely print the record.

To get the members whose age is above twenty five and less than thirty five years, use the following command:

```
$ awk '$2>25  &&  $2<35   {print  $1}'  dbfile
Paul
```

Here we use the relational operators "greater than" and "less than" to make the selection of records. In addition the logical operator && "and" is used to validate both relational operations. Similarly, we can use the relational operators and logical operators in awk programming.

Logical Operators:

a | | b	- (Logical OR) Evaluates to true if either a or b is true
a && b	- (Logical AND) Evaluates to true if both a and b are true
! a	- (Logical NOT) Evaluates to true if a is not true

Relational Operators:

a = = b	- (Equal to) Evaluates to true if a equals b
a < b	- (Logical AND) Evaluates to true if a is less than b
a > b	- (greater than) Evaluates to true if a is larger than b

Note: We have so far used some of the predefined variables like $0, $1, $2 etc. Here is another predefined variable NR, recognized by awk as the Number of the Record currently being processed. There is one more predefined variable of this sort, NF, which is the number of fields in the current record.

10.7. ARITHMETIC OPERATIONS

So far we have selected records, printed specified fields and assigned values to variables. **awk** also allows to perform arithmetic operations like addition, subtraction etc. on fields and variables.

* From the database, the salary for the members should be dispersed after deducting the PF amount of 10%. To accomplish this create a program comm.4 with the following commands.

```
$ cat awkprog.4
{
       name = $1
       age = $2
       sex = $3
       salary = $4
       percent = 0.1
       deduction = $4 * percent
       print name, age, sex, salary  - deduction
}
```

```
$
[plokam@Linux1 ~]$ awk -f awkprog.4 dbfile
Brian   22      Male    4000.00
Brian 22 Male 3600
Lisa    25      Female  7500.00
Lisa 25 Female 6750
Paul    28      Male    10000.00
Paul 28 Male 9000
Charlie 19      Male    2000.00
Charlie 19 Male 1800
Bill    23      Male    5500.00
Bill 23 Male 4950
Nancy   21      Female  3500.00
Nancy 21 Female 3150
Tammy   25      Female  9000.00
Tammy 25 Female 8100
[plokam@Linux1 ~]$
```

In the above file we used multiplication and substraction.

* In the following example let us use addition and division. Let us calculate the average age of the members. Create a file awkprog.5 with following commands and execute.

```
[plokam@Linux1 ~]$cat  awkprog.5
       {
              sum+ =$2
       }
       END  {
                     print "Sum is", sum, "Average is", sum/NR
       }

[plokam@Linux1 ~]$ awk -f awkprog.5 dbfile
Sum is 163 Average is 23.2857
[plokam@Linux1 ~]$
```

Note: The statement sum + = $2 is the same as sum = sum + $2. The operator + = means that take the value held by the variable on the left, add to it the value specified on the right and assign the result of the sum toe variable on the left.

10.8. BEGIN AND END

In the last example, you would have noticed a special pattern END was present. The awk utility beings its processing of a data file by scanning each line in sequence for a specified pattern and performing the requested action on the lines that contain the pattern. It is often desirable to perform some actions before and/or after the lines of a dba file are processed. To facilitate this the special patterns BEGIN and END provide.

* **Create the file awkprog.6 with the following:**

```
$ cat awkprog.6
BEGIN{
             print "The members of the database list"
             print ""
             print "Name                Age    Sex          Salary"
       }
{
             name = $1
             age  =  $2
             sex  =  $3
             salary  =  $4
print name,  "      ", age, "               ", sex, "                ',
salary-deduction
}
END
{
             print "There are", NR, "Members in the database"
}
```

Note: The Program wont work as the "{" should be right after END in the above example.

```
[plokam@Linux1 ~]$ awk -f awkprog.6 dbfile
The members of the database list

Name                    Age     Sex              Salary
Brian   22      Male    4000.00
Brian      22              Male              4000
Lisa    25      Female  7500.00
Lisa       25              Female              7500
Paul    28      Male    10000.00
Paul       28              Male             10000
Charlie 19      Male    2000.00
Charlie      19              Male                2000
Bill    23      Male    5500.00
Bill       23              Male             5500
Nancy   21      Female  3500.00
Nancy      21              Female              3500
Tammy   25      Female  9000.00
Tammy      25              Female              9000
There are 7 Members in the database
[plokam@Linux1 ~]$
```

The line that contains the BEGIN statement instructs awk to execute the instructions that immediately follow it before even looking at the contents of the file comm.6

Only after printing the quoted sentences, awk does the normal processing of the roceds according to the instructions. The work BEGIN must be followed by an opening curly brace on the same line.

After all lines have been processed, the END pattern is matched and its associated action, printing the total number of records, is performed.

Some more examples to Check out:

Print and sort the login names of all users:

```
BEGIN  { FS = ":" }
       { print $1 | "sort" }
```

Count lines in a file:

```
       { nlines++ }
END    { print nlines }
```

Precede each line by its number in the file:

```
{ print FNR, $0 }
```

Concatenate and line number (a variation on a theme):

```
{ print NR, $0 }
```

10.9. CHANGING FIELD SEPARATOR

So far we have been using files with fields separated by either spaces or tabs (white space), which is **awk**'s default field separator. There are many files such as /etc/passwd, and other configuration file separated by different character such as colon(:).

In such cases **awk**should be informed that the field separator is not the defau lt white space but some other character. This can be indicated by using the -F option in the command line while invoking **awk.**

```
$ awk -F: 'NR == 3 {print $1 }' /etc/passwd
bin
```

10.10. INPUT / OUTPUT REDIRECTION

As with other UNIX commands, awk can be used in pipes and its output can be directed to other files or directly to the printer.

* To redirect awk's output to a file, enter:

```
$ awk  -f  comm.1  dbfile  >  out.1
$ cat out1
Subashini
Aiswarya
Anna
```

* If no filename is specified, awk takes its input from standard input. For example, enter:

```
$ sort dbfile   |  awk  -f  comm.1
Aiswarya
Anna
$
```

The **dbfile** is sorted by name and sent to standard output, which was connected to the input of **awk.**

10.11. LAB EXERCISES

Redirection, Pipes, and Filters

1. How will you send the output of ls command to a file sample.
2. Send the output of who command to a file named week.
3. Create a file by name database, which contains names, ages, designations and salaries.
4. Sort the above file.
5. Sort the file database based on the field ‘designations’ in reverse order.
6. Create a file numeric and insert ten numbers.
7. Sort the numeric file by using input redirection and send its output to sample.
8. Sort the file database based on the field ‘names’ ignoring case sensitivity.
9. Display all those lines in file database which contain “d” or “D”.
10. Display all those lines in file database which contain ‘c’ in the beginning of every line.
11. Display all those lines which do not contain the pattern “er” in the file database.
12. Count the number of lines in the file database, which contain either “s” or “S”.
13. Display those records in the file database having age between 20 to 30.
14. Count the number of lines, words and characters in the file database.
15. Count the number of files in your current directory.
16. How can you see a large document per screen? Observe the difference between pg and more.
17. Change the characters a,b,c in the file database with A,B,C.
18. Sort the output of who command without using temporary file.
19. In the above question, can you catch the output of who before sorting using a single command.
20. List all the files in your current directory which start with the letter ‘s’ using grep command.

11. Summary of Unix Commands:

Basic:

ls	cat	more	cp
mv	rm	diff	fold
head	tail	sort	rpl
awk	grep	sed	find
tr	wc	paste	nl

disk:

df	du	mount	umount
fstat	limit	ulimit	mkdev
chfs			

permissions:

chmod	chown	chgrp	Id
groups	passwd	umask	

Process / Jobs:

ps	jobs	bg	fg
kill	cron	at	service
sleep	nice	watch	uptime

Network:

ifconfig	ping	telnet	ftp
traceroute	netstat	nslookup	hostname
uname			

Archive / Compression:

tar	gzip	gunzip	
zcat	compress	uncompress	

Miscellaneous:

who	whoami	whereis	finger
cal	date	time	clear
exit	which		

12. Important Contacts:

Training:

Raju Cherukuri

scherukuri@miraclesoft.com

Office: 1-248-233-3654

Mobile 91-944-101-3362

Ravi Ijju

rijju@miraclesoft.com

Office: 1-248-233-3651

Delivery / Project / Technical Reports:

Ramani Lokam

rlokam@miraclesoft.com

Office: 1-248-233-1131

Mobile: 1-248- 760-1920