

UNIX PROGRAMMING UNIT – 1

INTRODUCTION TO UNIX

1. Brief History

The initial contributions to the development of Unix by The Bell Laboratory of AT & T and the University of California, Berkeley (UCB) are notable.

1. Bell Laboratory's contribution

In 1965, Massachusetts Institute of Technology (MIT), General Electric and The Bell Laboratories of AT&T worked on a joint venture project called Multics (Multiplexed Information & Computing System), which intended to develop a multi-user operating system as it is not satisfactory, AT&T withdrew itself from the Multics project in early 1969.

Ken Thompson and Dennis Ritchie worked on PDP-7 machine, developed an OS called UNICS (Uniplexed Information and Computing System) during the latter period of 1969. UNICS was developed in assembly language of PDP-7 and so it was not portable. To achieve portability, Thompson worked on the implementation of the system in high level language called B. As the B language didn't yield the expected results, Ritchie developed a higher level language called C in 1973. The researchers in AT&T showed interest in the Unix project (around 1970 UNICS became Unix). During those days many text-processing utilities along with a text editor called the ed editor and a simple command interpreter called the shell were developed. The ed editor was a line editor and the then developed shell became the Bourne shell (sh), the grandfather of almost all the currently available shells.

The results of the research and developments made at the Bell laboratory were first published in the form of Unix Programmer's Manual in the late 1971. Since then, there have been a total of 10 editions of this manual. Each of these manuals correspond to a version of the Unix released by AT&T. The 3rd edition published in early 1973 included the details of C compiler. Ritchie completely rewrote the entire Unix system during the same year using C. Actually, around 95% of this Unix system was written in C and the remaining was written in the assembly language. The platform used was a PDP-11 machine. The details of the Unix implementation in C was made public through a paper published in 1974 and its authors, Thompson and Ritchie were later awarded with the prestigious ACM Turing award.

A system called Unix System V was announced in 1983. With this release AT&T assured the upward compatibility of all its future releases. System V has since then undergone many revisions and releases. The most important of the releases is System V release 4 (SVR4) in 1991. SVR4 brought all the important features of various operating systems like BSD, XENIX and SUN operating systems together that were available by then. Early days of development of Unix, AT&T was forbidden to manufacture and promote any equipment, that was not related to telephone or telegraph services. However, it made Unix system available to universities, commercial firms and defence laboratories either free of cost or at a nominal price.

UNIX PROGRAMMING UNIT – 1

2.UCB's contribution

University of California at Berkeley (UCB) was one of the early universities that was interested in the Unix operating system and its development. It was responsible for many important technical contributions and in development of useful utilities.

Ex: ex editor and Pascal compiler were developed during 1974 by Bill Joy and Chuck Haley, then graduate student of UCB. Later the ex editor, which was also a line editor, was provided with the screen-editing facilities and was called the vi editor. Another important contribution of Bill Joy was the C-shell (csh). UCB released their own version of Unix, called BSD-Unix during the spring of 1978. Since then UCB had several of BSD releases. These are referred to as 4.0BSD (1980), 4.1BSD (1981), 4.2BSD (1983), 4.3BSD (1986) and 4.4BSD (1993).

DARPA (Defence Advanced Research Projects Agency) funded Unix systems development activities, was interested in the development and integration of TCP/IP network protocol suite. The financial support of DARPA helped UCB to release its BSD versions as listed above. UCB technical contributions are Virtual Memory System (VMS), Fast File Systems (FFS), socket facility, Larger file names and a reliable signals implementation and of course, the TCP/IP. After 4.4BSD, scarcity of funds, competition from external commercial organizations, difficulties in the management of large and complex system by a small group of researchers, made it difficult by the UCB to further work on the development of Unix systems.

3.Other's contribution

During the same period, many computer vendors had developed their own Unix systems. For example, Sun Microsystems (a company that was promoted by Bill Joy) developed Sun operating system, which was revised and renamed Solaris. Solaris 7 is one of the widely used OS even today. Digital Equipment Corporation (DEC) developed a system called Ultrix, which was revised and renamed Digital Unix. Microsoft developed a system called XENIX, the first Unix variant to be run on a PC. This OS was based on both AT&T and BSD systems. XENIX was finally sold to SCO (Santa Cruz Operations). Later SCO developed its own version of these systems named SCO Unixware-7 and the SCO open server. Other important systems developed are AIX (by IBM), HP-UX (by HP) and IRIX (by Silicon Graphics).

4.Why so many variants?

From the mid 1970s there have been many variants of Unix system. One of the reasons is, AT&T being a telephone company, was not permitted to sell computer-based products. However, it could do so free of cost or for a nominal fee. BSD also giving its products free of cost, many obtained the copies of Unix and worked on them. Another reason was that these systems were developed mostly by researchers for researches and were revised constantly to suit different requirements. All of these resulted in the development of many Unix variants as well as its confinement only to the portals of universities, research organizations and American Defense Laboratories. One of the important points that worked against the popularity of any Unix variant

UNIX PROGRAMMING UNIT – 1

for a long time was its user-unfriendliness. However, the introduction of X Window system by MIT during the second half of 1990s has made it user-friendly.

5.Are there any standards?

The first attempt was made by the IEEE standards board to standardize the Unix system. This group came out with a set of rules that should be compiled with for an OS to be called standard Unix. These set of rules are widely known as POSIX (Portable Operating System Unix). Now POSIX has also undergone many revisions. The latest one is IEEE 1003.10. Infact, AT&T also has its own standard called Unix International (UI). IBM and HP and DEC also formed a consortium called Open Software Foundation for the same purpose. However, still there exist a large number of Unix variants in the market.

6.Linux

In August 1991, a system called Linux was announced by Linus Torvalds in Finland. Actually, it was based on a system called Minix (chiefly developed by Andrew S Tanenbaum) which again was based on Unix. It brought in the speed, efficiency and flexibility of Unix to a PC environment, thereby using the advantages of all the capabilities of Unix. In March 1994, Torvalds released the 1.0 kernel of the Linux. Actually, Linux is an open source program - its source code is freely available. Anyone can work on it and make enhancements to it and as a result, it is under constant development. Like other Unix variants it was also initially popular only among the researchers and programmers at universities and research environments. However, at present, Linux has become widely popular among commercial and industrial circles along with the universities and research organizations around the world. Today, Linux has many flavors and can be found on computers ranging from desktops to corporate servers. Red Hat Linux is one of the most popular flavors of Linux. All versions of Linux may be downloaded free of cost from the Web.

2.What is UNIX?

Unix is an operating system. An OS is a software that acts as an interface between the user and the computer hardware and also as resource manager.

- From a user's perspective, an OS is the means to run application programs such as word processors, electronic spreadsheets, database management systems and the like. In other words, application programs access the computer's hardware via an operating system like Unix.
- From the system point of view, the concurrently running tasks are just different processes- them belonging to the same user or to different users is immaterial.

UNIX PROGRAMMING UNIT – 1

Salient features of Unix:

- 1) UNIX is a multi-tasking operating system.
 - It has the ability to support concurrent execution of 2 or more active processes.
 - Different tasks like processes running concurrently belong to one user.
- 2) UNIX is a multi-user operating system.
 - It has the ability to support more than 1 user to login into the system simultaneously and execute programs. It presents virtual computer to every user by creating simulated processors, multiple address spaces and the like.
 - Different tasks belong to different users
- 3) Unix operating system supports multi-users. These users might be directly connected to the same machine through different terminals or may be connected to different machines that are interconnected. Though initially Unix had no interconnection networking with different computers, the development of communication protocols like TCP/IP have made this possible. Along with networking, the system has very good inter-machine communication facilities. This has enabled different users connected to the computer networks to exchange information in the form of e-mail and shared data.
- 4) Unix operating system is highly portable. Compared to other OS, it is very easy to port Unix onto different hardware platforms with minimal or no modifications at all because a larger chunk of Unix is built on the language C , which itself is highly portable.
- 5) Unix offers solid security at various levels, beginning from the system startup level to accessing files as well as saving data in an encrypted form.
- 6) Unix has become popular since the early 1990s, it was started during early 1970s. The good library of utilities and command, has made the development of newer application programs easy and quick.
- 7) Unix system treats everything, including memory and I/O devices as file and has a very well-organized file and directory system that allows users to organize and maintain these files/directories easily and efficiently. Furthermore, as Unix views and treats everything as a file it is device independent.

3.UNIX COMPONENTS

UNIX system consists of 3 major components. 1) The Kernel 2) The Shell 3) The file system

1)The Kernel

The Kernel is relatively a small piece of code that is embedded on the hardware, mostly written in C and gets automatically loaded on to the memory as soon as the system is booted.

UNIX PROGRAMMING UNIT – 1

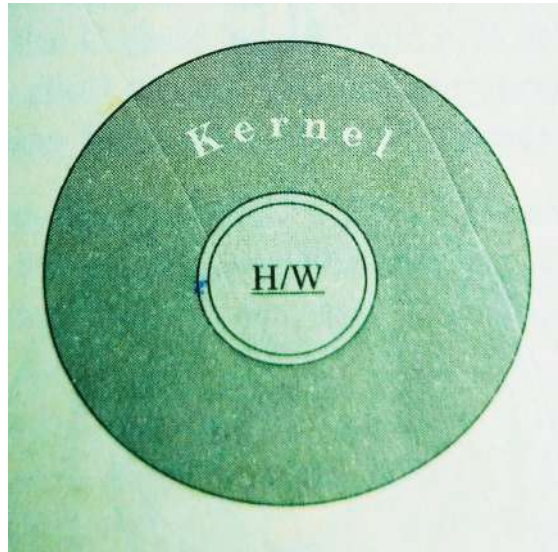


Figure 1 The Kernel

It is the only component that can directly communicate with the hardware. It manages all the system resources like memory and I/O devices, allocates time between users and processes in the case of multi-user environment, decides process priorities, manages IPC and performs many other tasks.

- **Monolithic kernels**

Earlier, all the programs that were part of a Kernel, were integrated together and moved onto the memory during booting. Such integrated kernels are referred to as monolithic kernels.

- **Microkernel**

The programs are grouped into different modules and only the necessary module is moved onto the memory during booting. This just-necessary and sufficient module consisting of a small set of Kernel programs is called a microkernel. The other modules are moved in and out of the memory depending on the requirement.

2)The Shell

A shell is a program that sits on the Kernel and acts as an agent or interface between the users and the Kernel and hence the hardware. It's similar to command.com in the MS-DOS environment

UNIX PROGRAMMING UNIT – 1

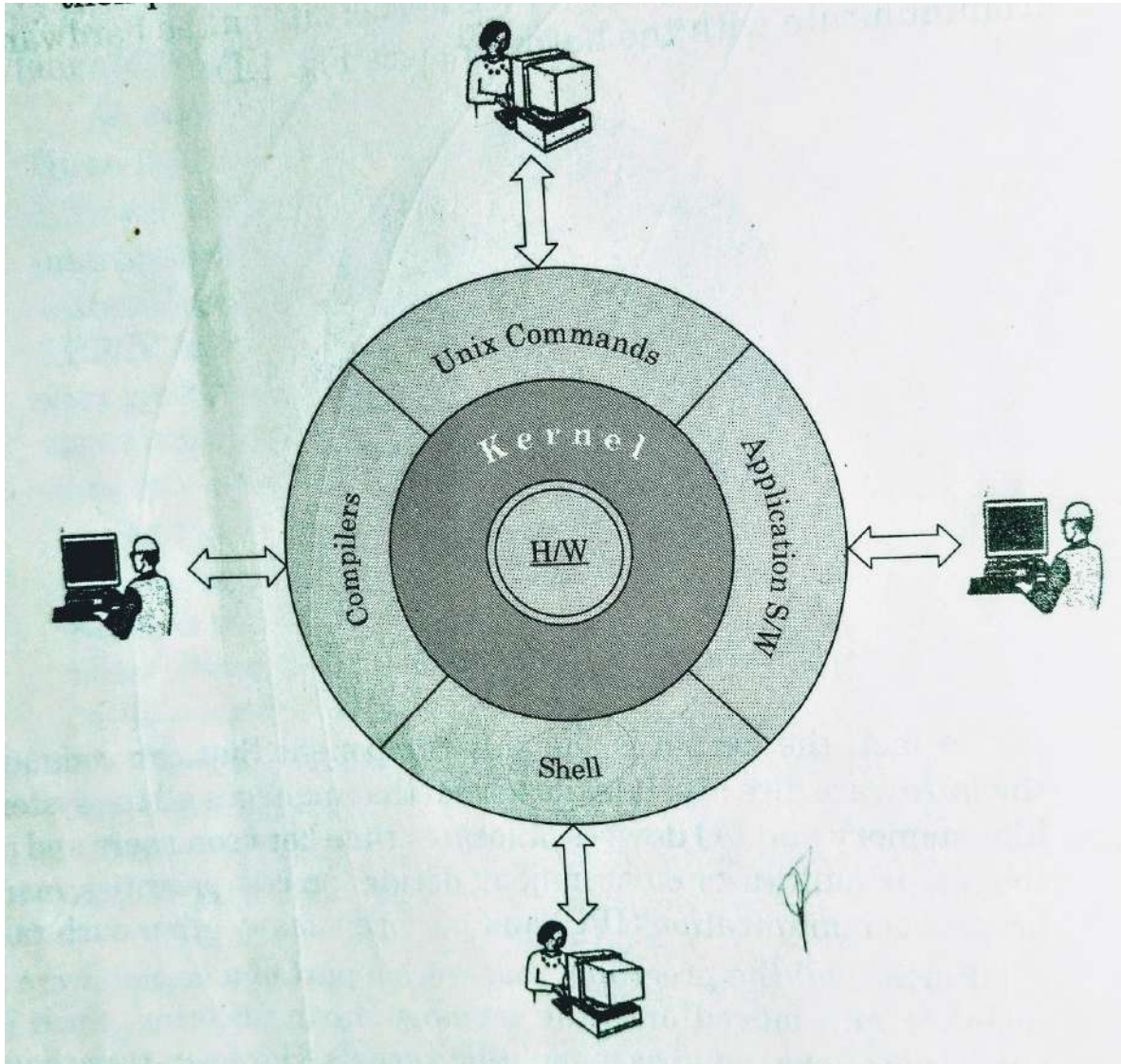


Figure 2. Unix system components

A Shell is a command interpreter or a processor. As soon as the system is booted successfully, the Shell presents a command line prompt (\$ or % symbol) at which the user can type in any UNIX command. After accepting the command, the Shell generates a readily executable simple command line by parsing it, evaluating variables, performs command substitution, interprets metacharacters like * and ? and identifies the PATH. This simple command line is then passed onto the Kernel for execution.

UNIX PROGRAMMING UNIT – 1

Shell has programming capabilities, using this we can write shell programs (Shell scripts).

a) Types of Shells

- **The Bourne shell (sh)**

It has been named after its author, Stephen Bourne at AT&T Bell Labs. This is distributed as the standard shell on almost all Unix systems.

- **The C shell (csh)**

Bill Joy developed this shell at UCB as a part of the BSD release. Its syntax is very similar to the C programming language and not compatible with Bourne shell.

Advantages

1. It has the capability to execute processes in the background.
2. A version of this Shell called tcsh is available free of cost under Linux.

- **The Korn shell (ksh)**

This Shell was developed by David Korn at AT&T Bell labs. It is built on the Bourne shell and incorporates certain features of the C shell. At present it is the widely used shells.

It can run Bourne shell scripts without any modifications.

One of its versions, public-domain Korn shell (pdksh), comes with Linux free of cost.

- **The Bourne-Again shell (bash)**

This Shell was developed by B Fox and C Ramey at Free Software Foundation. Certain Linux OS variants come with this Shell as its default shell. This is clearly a freeware shell.

b) Shell as a command processor

When interpreting a command line given at its prompt, Shell follows 1 or more or all of the following steps, depending on the contents of the command line given to it.

1. It parses the command line and identifies each and every word in it and removes additional spaces or tabs present, if any.
2. Evaluates all the variables present that might be prefixed with a \$.
3. If commands are present within back quotes, they are executed and their output is substituted into the command line. In other words, command substitution takes place.
4. It then checks for any redirection of the input and/or output and establishes the connectivity between the concerned files accordingly.
5. It then checks for the presence of wild card characters like *, ? and [,]. If any of these characters are present, file name generation and substitution take place.

It then looks out for the required commands as well as files, retrieves them and hands them to the Kernel for execution. The route or path to look for required variables are in PATH shell variable. The semicolon that allows multiple commands and logical operators are taken care by the Shell.

UNIX PROGRAMMING UNIT – 1

3) The File System

Unix treats everything- including hardware devices- as a file. All the files are organized in an inverted tree-like hierarchical structure. The structured arrangement in which all the files are stored is referred to as a file system from the user's point of view. Actually this is something more for implementers and system administrators.

A file system could be local to a system or it could be distributed.

- Local file systems store and manage their data on devices directly connected to the system.
- Distributed file systems allow a user to access files residing on remote machines.

4) USING UNIX

The process of getting into the UNIX environment is known as logging in into the system. After the system is booted a daemon called init gets started along with some other daemons. This init daemon spawns a process called getty for every terminal. Each of these gettys print the login prompt on the respective terminal.

The sequence of events in a complete login process can be listed as follows.

1. The user enters a login name at the getty's login prompt on the terminal.
 2. getty executes the login program with the login name as the argument.
 3. login requests for a password and validates it against /etc/passwd.
 4. login sets up the TERM environment variable and runs a shell.
 5. The Shell executed the appropriate startup files like .profile.
 6. The Shell then prints a prompt, usually a \$ or a % symbol and waits for further input.
- This indicates the successful entry made into a Unix environment with a proper shell.

The above sequence of events during login process is schematically shown in figure.

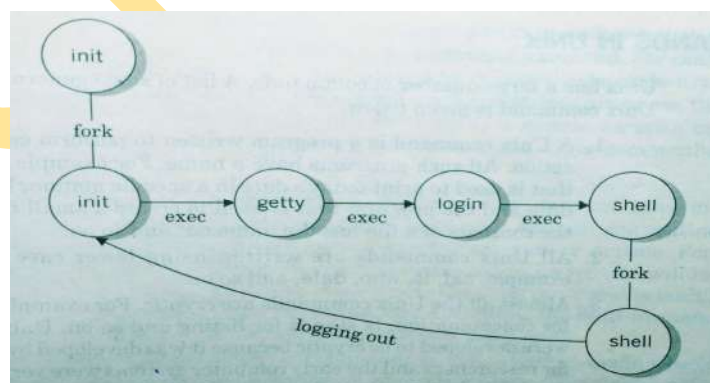


Figure 3. The log process

UNIX PROGRAMMING UNIT – 1

When a user completes the session with the system he comes out of the Unix environment. The process of coming out of the Unix environment is known as logging out. After the logout, the control returns to the init daemon, which in turn spawns a new getty on the corresponding terminal. This facilitates a new user to login to the system.

1)The Shell Prompt

Successful login into a Unix system is indicated by the appearance of a prompt called the shell prompt or system prompt on the terminal. The character that appears as a prompt depends on the shell used.

Table shows a list of the default prompts employed by different shells.

List of Default Prompts

Prompt	Shell
\$ (dollar)	Bourne and Korn shells (sh,bash and ksh)
% (percent)	C shells (csh and tesh)
# (hash)	Any shell as root

5)COMMANDS IN UNIX

General features of UNIX commands

1. It is a program written to perform certain specific action. Example: To print today's date the UNIX command is date. To create a small file or display the contents of a file cat.
2. Unix commands are written in lower case letters. For example cat, ls, who and so on.
3. The commands are cryptic. For example, cat stands for concatenation, ls stands for listing and so on.
4. Unix commands can have zero, one or more number of arguments associated with them.
5. It can have format specifiers and options. Format specifiers, whenever present are indicated by + character and options by hyphen (-).There could be many number of options associated with a command.
6. In certain situations, a Unix command with it's arguments or a series of commands may not fit in a single line (80 characters). In such cases it may overflow and is permitted. The overflow is indicated by a special prompt in the form of a > symbol in the beginning of the next line. Such a special prompt is known as the secondary prompt.
7. A current Unix command can be killed by using either <delete> or <ctrl-u> command.

UNIX PROGRAMMING UNIT – 1

8. Commands can be given to the system even when a command given earlier is being executed in the background. This is not possible with the Bourne shell, sh.

1)Types of Unix Commands External Commands

A command with an independent existence in the form of a separate file is called an external command. For example, programs for the commands such as cat, ls, exist independently in a directory called the /bin directory. When these commands are given, the Shell reaches these command files with the help of a system variable called the PATH variable and executes them. Most of the UNIX commands are external commands.

Internal Commands

A command that doesn't have an independent existence is called an internal command. Actually the routines for internal commands will be a part of another program or routine. For example, the echo command is an internal command as it's routine will be a part of the shell's routine, sh. These commands also called the built-in commands. cd and mkdir are examples of internal commands.

6)SOME BASIC COMMANDS

Unix has several hundreds of commands within it. Most of them are simple and are powerful. Some of the commands are general in nature from the user's point of view. A few of those commands are

a)The echo command

The echo command is used to display messages and useful in developing interactive Shell programs.

It takes 0,1 or more number of arguments. Arguments may be given either as a series of individual symbols or as a string within a pair of double quotes (" ").

Examples

1. \$ echo

\$

A Blank line is displayed

2. \$ echo I am studying computer science.

I am studying computer science.

\$

The extra spaces between the arguments are adjusted and output is printed in a standard form i.e., one blank between different arguments.

UNIX PROGRAMMING UNIT – 1

3. \$ echo I am studying computer science.

I am studying computer science.

\$

4. \$ echo "I am studying computer science."

I am studying computer science.

\$

When the string is given as an argument, it is printed as it is, without the adjustments of blanks.

5. \$ echo The home directory is \$HOME The home directory is /usr/mgv

\$

If an evalutable argument is given, it is first evaluated and it's value is printed along with the other arguments.

b)The tput Command

This command is used to control the movement of the cursor on the screen as well as to add certain features like blinking, bold face and underlining to the displayed messages on the screen.

Example

1. \$tput clear

This command along with clear argument clears the screen and puts the cursor at the left-top of the screen.

2. \$tput cup 10 20

This command along with the cup argument and certain co-ordinate values is used to position the cursor at any required position on the screen. Here, in this cursor will be placed at the tenth row and twentieth column on the screen.

3. \$tput lines

48

\$

\$tput cols

142

\$

UNIX PROGRAMMING UNIT – 1

The number of rows and columns on the current terminal is known by using the lines and cols as arguments to the tput command. Here, from the above examples, there are 142 columns and 48 lines on the current terminal.

c)The tty Command

In Unix, every terminal is associated with a special file, called the device file which is present in /dev directory. A user can know the name of his device file on which he is working by using the tty command.

Example

```
$tty
/dev/tty01
$
```

Here, tty01 is the device file name and will be available in the directory /dev. Under Linux, the output of this command will be shown below.

```
$tty
/dev/pts/0
$
```

d)The who Command

The user can know login details of all current users by using the who command. Generally, this command is used by the system administrator for monitoring terminals.

This provides a list of all the current users in the three-column format by default, as follows.

```
$who
root      console   Nov 19 09:35
mgv       tty01     Nov 19 09:40
dvm       tty02     Nov 19 09:41
$
```

The first column shows the name of the users, the second column shows the device names and the third column shows the login time.

Some options like -H, -u and -T can be used with this command. The -H option provides headers for the columns and the -u option provides more details like idle time, PID and comments as shown in the example below.

```
$ who -Hu
NAME      LINE      TIME          IDLE          PID          COMMENTS
root      console   Nov 19 09:35  .             32 mgv       tty01
```

UNIX PROGRAMMING UNIT – 1

```
Nov 19 09:40    0:20    33 dvm    tty02    Nov 19 09:41
0:40    34 $
```

If any terminal is idle (not active) for the last 1minute, will be indicated in IDLE column and this information will be useful to the system administrator. The PID indicates the process identification number.

The self-login details of a user can be obtained as a single line output using `am` and `i` arguments along with the `who` command as follows.

```
$ who am I
mgv      tty01      Nov 19 09:40
$
```

e)The `uname` Command

When this command is used, it gives the name of the UNIX system being used by the user. Certain options like `r`, `v`, `m` and `a` can be used with this command.

Examples

1. `$uname`
Linux
\$
2. `$uname -r`
2.4.18 -3 #release details
\$
3. `$uname -m`
i686 #machine details
\$

f) The `date` command

The user can display the current date along with the time nearest to the second.

```
$ date
Sat Jan 10 11:58:00 IST 2004
```

It allows the use of format specifiers as arguments. Format specifiers are single characters using which, one can print the date in a specific manner. Each format specifier is preceded by a `+` symbol followed by the `%` operator.

UNIX PROGRAMMING UNIT – 1

Example

1. Format specifier `m` display the month in numeric form.
`$date+%m`
09
\$
2. The name of the month can be displayed using the `h` format specifier
`$date+%h`
Sep
\$
3. More than one format specifier can be specified at a time. In such cases either double quotes (" ") or single quotes (' ') are used.
`$date+"%h %m"`
Sep 09
\$
4. `D` and `d` for the day of the month. (`D` gives the day in the format `mm/dd/yy`, where as `d` gives the day in the format `dd`).
`$date "Today's date is +%D"`
Today's date is 03/16/04
\$
5. `Y` and `y` for the year(`Y` gives all the four digits of the year, whereas `y` gives only the last two digits).
6. `H`,`M` and `S` stand for hour, minute and second respectively.
7. Many number of options like `u`,`r`,`R`,`f` can be used with this command. Example
`u` option displays the universal time (Greenwich Mean Time) where UTC is Coordinated Universal Time.
`$ date -u`
Sat Sep 25 05:58:20 UTC 2004
\$

The System Date

1. The `date` command is used by the system administrator to change or reset the system date.
2. To set the date numeric argument is given usually 8 characters long string of form `MMDDhhmm` (month,day,hour in 24-hour format and minutes) followed by an optional two-digit year.

UNIX PROGRAMMING UNIT – 1

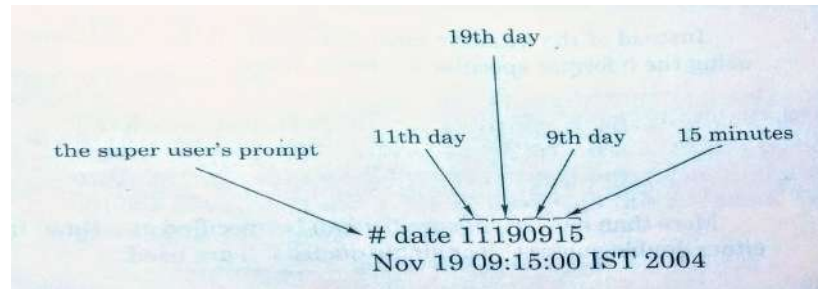


Figure 4. Setting system date

g)The cal command

1. This command is used to print the calendar of a specific month or a specific year. When used this command without any arguments, the calendar of the current month of the current year will be printed.

\$cal

\$cal						
Dec 2004						
Su	Mo	Tu	We	Th	Fr	Sa
			1	2	3	
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31
\$						

2. When two numeric arguments, are given the first argument will be considered as the month and the second argument will be considered as the year.

\$cal 09 1949

\$cal 09 1949						
September 1949						
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	
\$						

3. When given with a single numeric argument, the complete calendar for the entire year represented by the numeric argument will be printed as follows.

UNIX PROGRAMMING UNIT – 1

Year	Month	Su	Mo	Tu	We	Th	Fr	Sa
2002	Jan	6	7	8	9	10	11	12
	Tu	1	2	3	4	5	6	7
	We	13	14	15	16	17	18	19
	Th	20	21	22	23	24	25	26
	Fr	27	28	29	30	31		
	Sa							
	Feb	3	4	5	6	7	8	9
	Tu	10	11	12	13	14	15	16
	We	17	18	19	20	21	22	23
	Th	24	25	26	27	28		
	Fr							
	Sa							
Mar	31							
Tu	1	2	3	4	5	6	7	
We	8	9	10	11	12	13	14	
Th	15	16	17	18	19	20	21	
Fr	22	23	24	25	26	27	28	
Sa	29	30						
Apr	1	2	3	4	5	6	7	
Tu	8	9	10	11	12	13	14	
We	15	16	17	18	19	20	21	
Th	22	23	24	25	26	27	28	
Fr	29	30						
Sa								
May	1	2	3	4	5	6	7	
Tu	8	9	10	11	12	13	14	
We	15	16	17	18	19	20	21	
Th	22	23	24	25	26	27	28	
Fr	29	30	31					
Sa								
Jun	1	2	3	4	5	6	7	
Tu	8	9	10	11	12	13	14	
We	15	16	17	18	19	20	21	
Th	22	23	24	25	26	27	28	
Fr	29							
Sa								
Jul	1	2	3	4	5	6	7	
Tu	8	9	10	11	12	13	14	
We	15	16	17	18	19	20	21	
Th	22	23	24	25	26	27	28	
Fr	29	30	31					
Sa								
Aug	1	2	3	4	5	6	7	
Tu	8	9	10	11	12	13	14	
We	15	16	17	18	19	20	21	
Th	22	23	24	25	26	27	28	
Fr	29	30	31					
Sa								
Sep	1	2	3	4	5	6	7	
Tu	8	9	10	11	12	13	14	
We	15	16	17	18	19	20	21	
Th	22	23	24	25	26	27	28	
Fr	29							
Sa								
Oct	1	2	3	4	5	6	7	
Tu	8	9	10	11	12	13	14	
We	15	16	17	18	19	20	21	
Th	22	23	24	25	26	27	28	
Fr	29	30	31					
Sa								
Nov	1	2	3	4	5	6	7	
Tu	8	9	10	11	12	13	14	
We	15	16	17	18	19	20	21	
Th	22	23	24	25	26	27	28	
Fr	29							
Sa								
Dec	1	2	3	4	5	6	7	
Tu	8	9	10	11	12	13	14	
We	15	16	17	18	19	20	21	
Th	22	23	24	25	26	27	28	
Fr	29	30	31					
Sa								

h)The calendar Command

1. It is like an engagement diary that contains text information and offers a remainder service based on a file called the calendar.
2. This file must be in the present working directory/home directory. This file is created and managed by the user with the help of an editor on the screen.

UNIX PROGRAMMING UNIT – 1

3. This file works on today and tomorrow dates concept. The present working day's date is taken as today and the days upto and including the next working day are treated as tomorrow.

```
Contents of the file calendar { Sep 28, 2002 freshers day.  
On 30/09/02 mock G.R.E. test.  
First test begins from Oct 6, 2002.  
$date  
Sat Sep 28 10:45:50 IST 2002  
$  
$calendar #here calendar is the command  
Sep 28, 2002 freshers day  
On 30/09/02 mock G.R.E. test.  
$
```

i) The passwd Command

1. Unix is a multi-user system due to which there is always a security threat. The simplest and most widely used by all individual users is the use of passwords.
2. The system administrator permits or authorizes the new user by assigning a unique password to him or her.
3. A user can change their password using the passwd command.

```
$passwd  
Old Password: *****  
New Password: *****  
New Password: *****  
$
```

j) The Lock Command

1. The lock Command is used for locking a session for any required amount of time.
2. By default, the user can lock it for 30 minutes. This locking period can be changed by assigning a different value for the system variable DEFLOGOUT.
3. password: ***** re-enter password:***** terminal locked by mgv 0 min ago

When lock command is given, terminal asks for a password twice and it need not be the actual password that is used to log into the system. It could be a temporary password.

UNIX PROGRAMMING UNIT – 1

4. `$lock -45` #locks for 45 minutes
A numeric option may be used to lock a terminal for any period ranging between 1 and 60 minutes.
5. Many Linux distributions include a locking command called `vlock`, used to lock all individual sessions simultaneously.
6. Also a utility called `lock screen` is available, with many modern OS, using which a session on a terminal can be locked.

k)The banner Command

1. This command is available on SCO Unix. It is used to display banners or posters. 2. `$banner Larry Wall`

```
$banner Larry Wall
#
#      ##  #####  ##### #  #
#      # #  #  #  #  #  #  #
#      # #  #  #  #  #  #  #
#      #####  #####  ##### #
#      #  #  #  #  #  #  #
##### #  #  #  #  #  #  #

#      #
#      #  ##  #      #
#      # #  #  #  #  #
#      # #  #  #  #  #
#      # #  ##### #  #
#      # #  #  #  #  #
##  ##  #  #  #####  #####
$
```

There are two arguments and each argument has been printed on a separate line. Maximum of 10 characters are printed per line and if more the remaining will be truncated.

3. `$banner "Larry Wall"`

```
$banner "Larry Wall"
#
#      ##  #####  ##### #  #  #  #  #  #  #
#      # #  #  #  #  #  #  #  #  #  #  #  #
#      # #  #  #  #  #  #  #  #  #  #  #  #
#      #####  #####  #  #  #  #  #  #  #
#      #  #  #  #  #  #  #  #  #  #  #  #
##### #  #  #  #  #  #  #  #  #  #  #  #
$
```

A series of arguments may be given as a single argument in the form of a string.

UNIX PROGRAMMING UNIT – 1

l)The cat Command

1. The basic purpose of this command is to create small UNIX files.

Example

```
$cat>review
```

A> symbol following the command means that the output goes to the file name following it.

```
<ctrl-d>
```

```
$
```

- In the above example, review is file name. After \$cat>review command execution, the user can type the input from the keyboard.
 - The input operation is terminated by using <ctrl-d> on a new line.
2. The drawback of this method to create file is that it lacks editing capabilities. The cat command is seldomly used to create files of considerable size for this editors like vi and emacs are used.

m)The bc Command

1. The bc command is both a calculator and a small language for writing numerical programs.Math functions are used by invoking bc with the option -l.

<i>Function</i>	<i>Acronym</i>
Cosine	c(n)
Sine	s(n)
Tan	t(n)
Arctan	a(n)
natural log	l(n)
exponential function	e(n)
square root	sqrt(n)
exponent	^

2. The bc can be used by either entering expressions to be evaluated from the keyboard or running programs stored in files.
3. The syntax used to write numeric programs and to define user-defined functions is similar to C programming language.

1. \$bc
sqrt 55
7 quit
\$

UNIX PROGRAMMING UNIT – 1

```
2. $bc
   scale=4
   sqrt 55
   7.4161 quit
   $
```

```
3. $bc
   ibase=5
   obase=16
   2341 424
   ibase=16
   obase=5
   424 2341
   quit
   $
```

From the above examples we can understand the following

1. The default value of the function scale is 0 (Zero).
2. Precision is set to 4 or any required value using the scale function above.
3. The result is displayed immediately in the next line after the execution of every line.
4. A session with bc is terminated by using the quit command.
5. Base conversion is carried out using ibase and obase functions.
6. ibase stands for the input base and obase stands for the output base. Default values for both ibase and obase is 10.

Example that uses the control construct is shown here.

```
$bc
for(i=1;i<=4;i=i+1)i^2
1
4
9 16
quit
$
```

UNIX PROGRAMMING UNIT – 1

n)The spell and ispell Commands

The spell command is the first program that was developed to check for words that are wrongly spelt in a document. This command displays a list of misspelled words in the document used as arguments, as shown below.

```
$cat spell.ux
This is an exmple
I am testing the spel command.
Als I am testing the ispell comand.
$
```

```
$spell spell.ux
Als comand
exmple spel
$
```

The misspelled words are displayed in alphabetical order based on American usage. If we want for British usage then -b option is included. Actually, spell check compares the words in the text with the words on an in-built dictionary.

ispell command is an interactive spell-check program available in Linux. When used, this command displays a screen full of information in the sections as shown below.

```
$ ispell spell.ux
This is an example
I am testing the spell command
Als I am testing the ispell comand.
1) mine          5)examples
2) example      6)expol
3) exemplar     7)ampule
4) exampled    8)example's

i) Ignore       I)Ignore all
r) Replace      R)Replace all
a) Add         X) Exit
```

?

UNIX PROGRAMMING UNIT – 1

7)COMMAND SUBSTITUTION

1. In Unix, it is possible to run a command within a command. For example, the date command can be run within the echo command by writing a command line as follows.
2. Example
\$echo Today the date is 'date'
Today the date is Fri Oct 3 16:25:00 IST 2002
\$
In the above example, the command to be executed (that is, echo in this example) has to be written within a pair of backquotes("").
3. The Shell while parsing the parameters list of the echo command treats the words that are backquotes as a command, executes it and substitutes the result of this execution at the corresponding position in the parameters list. This process is known as Command Substitution.
4. In Korn Shell the command substitution is accomplished by using a \$ sign followed by the command within a pair of parenthesis as shown below.

```
$echo Today the date is $(date)
Today the date is Fri Oct 3 16:25:00 IST 2002
$
```

8)GIVING MULTIPLE COMMANDS

1. Normally, a single command is given to the Shell at its prompt. However, there are many situations when more than one command is given in a single command line. One of the ways of giving multiple commands is to use a semicolon (;) between successive commands as shown below.
\$echo "Giving multiple commands";date;who
2. Commands given in this way doesn't mutually interact with each other in any manner. They are executed independently one after the other, from left to right as they appear in the command line.

Advantage

- Giving multiple commands in a single command line has a definite advantage as the entire command line could be executed as a background job and something else could be done in the foreground.
- Of course, the Bourne shell (sh) doesn't permit processing of jobs in the background where as the Korn shell (ksh) does.

UNIX PROGRAMMING UNIT – 2

The File system

1)The Basics of Files

A file is a sequence of bits, bytes or lines that is stored on a storage device like a disk. A Unix file may contain a source program, an executable code, a set of instructions or programs for the computer system or database. Thus, for Unix everything that is just a storehouse of information is a file.

1)File Names:

A file name may be given using any of the ASCII characters except the NULL character and the forward slash (/). Files are constructed and are used by names.

The length of a file in UNIX can be up to 256 characters. Most of the file systems consider only the first 14 characters of a file name and other characters, if any, are neglected.

The file names in UNIX are case sensitive.

The recommended characters to construct a file name are.

- Alphanumeric characters (combination of letters and numeric digits).
- The period (.), the hyphen (-) and the underscore (_).

The metacharacters are not recommended to use as file name.

A Unix file name may or may not have an extension. Only application software's impose this restriction. Example, C compiler has .c extension, SQL scripts to have .SQL extension and so on.

A dot (.) character can be used to construct a file name. Any file name beginning with a dot character is called hidden file or a dot file generally used to store specific information like configuration or startup information.

2)Categories of Files:

Depending on the significance of the contents of the file and behavior of the permissions granted to these files, UNIX files are classified into the following three categories.

- (i) Regular Files
- (ii) Directory Files
- (iii) Device Files or Special Files

(i) Regular Files

It is a randomly addressable sequence of bytes also called as ordinary files. Most of the files, like data files, source program files, files containing unix commands or any text file are called regular files. These files are created, changed or deleted by the user whenever he or she needs.

UNIX PROGRAMMING UNIT – 2

ii) Directory Files

UNIX treats everything as a file, they need to be organized. Organizational details of files are stored in files called directories or directory files (Directories are known as folders under the windows environment). The directories point to some other directories called sub-directories.

The UNIX file system is organized as directories, where each directory can contain sub-directories and/or files.

In general a directory file contains the following two information chunks.

1. The file name
2. It's Identification number (called the inside number).

These two information are stored in the form of a table. A user can create or remove directories. It is the kernel that manages the directory files.

Directory Types

There are 4 types of directories available in UNIX.

- a) Root Directory (/)
- b) Home Directory
- c) Working Directory
- d) Parent Directory

a) Root Directory (/)

The Root directory is the highest level in the hierarchy. It is the root of the whole file structure. There is a reference point for all files, directories and sub-directories. This reference point is known as root directory.

The root directory belongs to the system administrator and can be changed by only the system administrator.

b) Home Directory

The directory into which a user enters automatically when she/he logs in is known as the home directory or login directory. Every user has a home directory. It is created by the system administrator whenever he opens an account for a user. Generally, home directories are created under the /usr directory and will have login name as it's name.

Example

/usr/mgv is the home directory where mgv is login name.

c) Working Directory

The working or current directory is the one that we are in at any point in a session. When we are not changing from our home directory, then our working directory is our home directory. Our working directory changes automatically, when we change our directory.

UNIX PROGRAMMING UNIT – 2

d) Parent Directory

It is immediately above the working directory or current directory. When we are in our home directory, its parent is one of the system directories. When we move from our home directory to a subdirectory, our home directory becomes the parent directory.

(iii) Device Files

A device file is a point of interface to one of the computer's hardware devices. Thus, acts as a communication channel between two or more co-operation programs.

Advantage

User can use a device without knowing the idiosyncrasies of the hardware.

Types of Device Files

There are 2 types of device files.

- Character Special Files
- Block Special Files

Character Special Files are related I/O and used to model serial I/O devices like terminals, printers and networks. These files process one character at a time. These files are also known as raw device files.

Block special files are used to model devices like disk drives and magnetic tapes. These files allow buffered blocks of data to be read from a device and sent to a device efficiently.

2) PATH NAMES

It specifies where a file is located in the hierarchically organized file system and is necessary to know how to use pathnames to navigate the UNIX file systems. The route that is taken to reach a file (of any type) in a file system is known as the path to that file. It is necessary to know how to use path names to navigate the UNIX file system.

Two types of path names available in UNIX.

- A) Absolute pathnames
- B) Relative pathnames

A) Absolute Pathnames

Absolute Pathnames tells how to reach a file beginning from the root and always begins with a slash(/) (from the root).

Example

/home/501/sample

UNIX PROGRAMMING UNIT – 2

B)Relative Pathnames

Relative pathnames tells how to reach a file from the directory we are currently in. A directory or a file under a present working directory can be accessed by providing this pathname. It never begins with a slash(/).

Example

501/sample

3) OPERATIONS OR COMMANDS UNIQUE TO DIRECTORIES

- A) List Directory (ls)
- B) Make Directory (mkdir)
- C) Change Directory (cd)
- D) Remove Directory (rmdir)
- E) Present Working Directory or Locate Directory (pwd)

A)List Directory (ls)

1. This command is used to list all the files in a current directory.

Syntax

\$ls

Example

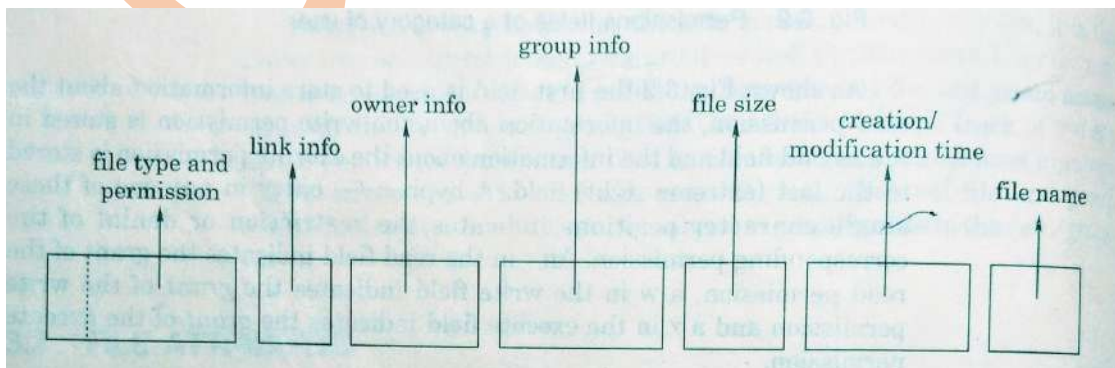
\$ls

hspmu

rthyn

n

\$



UNIX PROGRAMMING UNIT – 2

Figure 5. File attributes

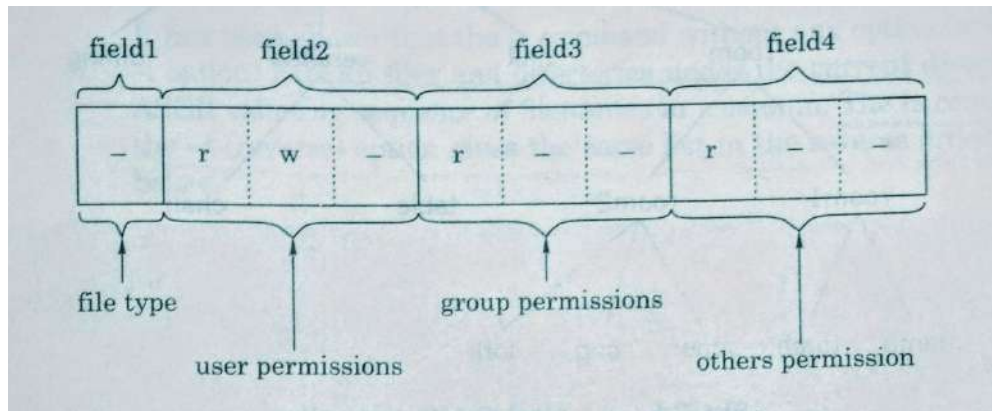


Figure 6. File type and permissions fields

2. The files can be listed row-wise by using the option -x.

```
ls -x
hspmurthy gun
$
```

3. All hidden files present in the present working directory can be listed using -a command.

```
ls -a
.
..
.
bbnhsp
murthyv
nn
$
```

4. The above listing can be obtained in the row format using the -x option along with the -a option. ls -xa

```
. .. .bbnhspmurthyvnn
```

5. The ls can be used to check if a file already exists or not by using the name of the file as an argument. \$ls my file myfile

```
$
```

UNIX PROGRAMMING UNIT – 2

B) Make Directory (mkdir)

1. The mkdir command is used to make one or more new directories. Upon execution a new directory called hmk is made under the present working directory.

Syntax

```
$mkdir directory-name
```

Example

```
$mkdir hmk
```

2. Assuming the present working directory is mane, first two sub-directories are created first using the mkdir command as follows.

```
$mkdir bin house
```

```
$
```

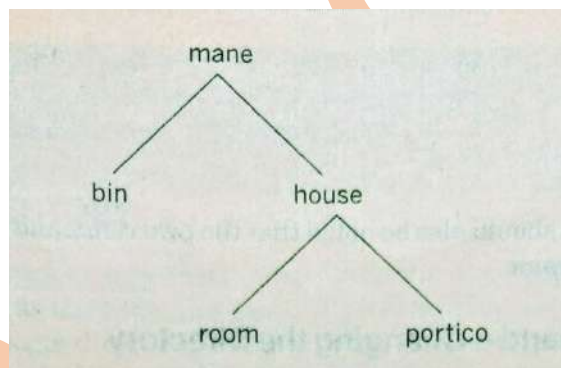


Figure 7. A typical directory

3. Next, house is made as current directory using the cd command and then the sub-directories room and portico are made using the mkdir command.

```
$cd house
```

```
$mkdir room portico
```

```
$
```

4. The entire directory is created in the single step \$mkdir bin house house /room house/portico.

C) Change Directory (cd)

The cd command is used to change the current working directory. This command uses a pathname as its argument that could be either absolute or relative pathnames.

If the cd command is used without any argument it automatically puts the user into the home directory.

UNIX PROGRAMMING UNIT – 2

Syntax

```
$cd directory-name
```

Example1

```
$cd user1
```

Example2

The present working directory is /usr/mgv, it could be changed to /usr/dvm by using the CD command.

```
$cd /usr/dvm
```

```
$
```

D)Remove Directory (rmdir)

This command is used to remove one or more directories or sub-directories. Directories can be removed using this command only when they are empty. However, if user wants to delete with remove without caring they are empty or not , it could be done by using rm command with the -r and -f options.

Syntax

```
$rmdir directory-name
```

Example

```
$rmdir user1
```

Assuming that the present working directory is /mane/house the following command line is used to remove the directory portico:

```
$ rmdir portico #current directory must be house.
```

E)Present Working Directory or Locate Directory (pwd)

The directory in which a user works at any point of time is known as the current directory or present working directory. A current directory may or may not be the user's home directory. This command is used to find out the current or present working directory. The pwd command always gives the absolute path name.

Syntax

```
$pwd
```

Example \$pwd

```
/usr/mgv
```

```
$
```

It has no options and attributes.

UNIX PROGRAMMING UNIT – 2

4) OPERATIONS OR COMMANDS UNIQUE TO FILES

- A) Create and edit file (vi)
- B) Display file (more)
- C) Print File (lpr)

A) Create and edit file (vi)

The most common tool to create a file in UNIX system is text editor such as vi. Other utilities are cat and ed. UNIX provides several utilities to edit text files.

The most common is a basic text editor such as vi.

A vi editor can be invoked in anyone of the following ways.

1. It is invoked to create a new file by giving the vi command without any argument as follows.
\$vi

A blank space with 1) the cursor on the left- top corner on the screen. 2) a message of the form new in the last line and 3) tilde characters (~) in the beginning of all the other lines. The editor will be in the command mode.

2. The second method is giving the vi command with the filename as its argument as follows.
\$vimaland

The behavior is exactly the similar to the first method except that the message on the last line will be maland [New File]

3. An existing file, say test file, can be invoked with the filename as the argument of the vi command
\$vitestfile

Other basic text editor is sed.

All of the basic edit utilities can create a file, but only some can edit one.

B) Display file (more)

This command is used to view the contents of a file page by page. It can take one or more file names as its arguments. One screen full information is displayed at a time. After each screen full of information is displayed, the more pauses with a message appearing at the bottom left corner of the screen.

UNIX PROGRAMMING UNIT – 2

Syntax

```
$more filename
```

Example1

```
$more trial.txt
```

This is a pager program and is a contribution of the Berkeley school. This command is used to view the contents of a

.....

.....

```
-- more(15%)-
```

All three files are displayed one after the other, page by page, starting from the first line. The complete display of the first file the more command pauses and a message "--more--"(Next file: sample)" appears. To continue user has to press space bar or give the f command.

Example2

```
$more sample1 sample2 sample3
```

Display options for the more command

-c (clear) tells more to display each screen top to bottom rather than scrolling

-d User can go half page forward. One can go forward by just a line using the return key.

-f one can go forward by one page

-b one can go backward by one page

-s (squeeze) displays the output with single-line spacing without affecting the original file.

=(equal to) current line number can be displayed.

The previous command can be repeated using the . (dot) command.

C)Print File (lpr) lpr means line printer. This command is used to print a file.

Syntax

```
$lpr filename
```

Example

```
$lpr story
```

5) OPERATIONS OR COMMANDS COMMON TO BOTH FILES AND DIRECTORIES

- A) Copy (cp)
- B) Move (mv)
- C) Rename (mv)
- D) Link (ln)
- E) Remove (rm)
- F) Find (find)

A) Copy (cp)

1. This command is used to create a duplicate of a file, a set of files or a directory. The cp command copies both text and binary files. This command can also be used to copy a file or group of files. First argument is source filename and second is destination filename. If the destination file exists already, it overwrites. If the destination file doesn't exist it is created and the contents of source file is copied into it.

Syntax

```
$cp source-file/directory destination-file/directory
```

Examples

```
$cp file1 file2
```

```
$cp section1 preface
```

```
$cp dir1/file1 dir2/file2
```

```
$cp dir1/file1 dir2
```

```
$cp file1 dir2/file2
```

2. Accidental overwriting can be avoided by interactive option (-i) `$cp -i section1 preface`
cp: overwrite 'preface'?

3. A file can be copied into another directory where programs is a directory under the current directory.

```
$cp section1 programs/preface
```

```
$cp section1 programs
```

```
$
```

Where section1 and preface are file names, programs is directory name.

UNIX PROGRAMMING UNIT – 2

4. Copy three files into directory.

```
$cp section1 section2 section3 chapter
```

5. Copy all files and sub-directories under a current directory into another directory can be done by recursive option.

```
$cp -r srcddesd
```

```
$
```

B) Move (mv)

1. The mv command is used to move either an individual file, a list of files or a directory. This command takes a minimum of 2 arguments. The first argument is name of a file or a directory to be moved and second argument is also a filename or directory name.

Syntax

```
$mv source-file/directory destination-file/directory
```

Example

```
$mv dir1/file1 dir2
```

```
$
```

2. A file from the current directory can be moved to another directory.

```
$mv review/usr/mgv # moves the file from the current directory to the mgv directory.
```

3. A group of files can be moved into a directory.

```
$mv section1 section2 section3 chapter1
```

```
$
```

C) Rename (mv)

1. UNIX doesn't have a specific rename command. The mv command is used to rename a file or directory. This command takes a minimum of 2 arguments. The first argument is name of a file or a directory to be renamed and second argument is also a filename or directory name.

Syntax

```
$mv old-file-name new-file-name
```

UNIX PROGRAMMING UNIT – 2

Example

```
$mv section1 section2
```

2. A directory can be renamed.

```
$mv mgvhdr #both mgv and her are directory names
```

C) Link (ln)

A file has more than one name and one of the reasons to have multiple filenames is that of security. The ln command is a standard Unix command utility used to create a hard links or a symbolic links(symlink) to an existing file.

- Hard Links

A new filename can be linked to an existing filename and inturn linked to its physical file on the disk. If trial is an existing filename and test is another name for the same file, the two are linked by the ln command.

```
$ln trial test
```

```
$
```

Advantages

- Changes or modifications made by one will be applicable to other user also.

Limitations

1. Directories cannot be linked.
2. Files across two different file systems cannot be linked.

- Symbolic Links

Symbolic links are files that hold the pathname of the original file. These are obtained by using command ln with option -s.

```
$ln -s trial inspect; $ls -li trial inspect
```

```
1372 -rw-r--r-- 1 mgvcsd 568 Nov 11 13:10 trial
```

```
8975 lrwxrwxrwx 1 mgvcsd 4 Nov 11 13:15 inspect->trial $
```

In the above, trial is filename and inspect is link name. Here, inode numbers are different, the file type of link file is l, size of link file is 4 bytes, which is sufficient to hold a path name and the link value of either of the files is not altered.

E) Remove (rm)

1. The command rm is used to remove or delete files. This command can delete more than one file with a single instruction.

Syntax

```
$rm filename
```

UNIX PROGRAMMING UNIT – 2

Example

```
$rm sample example    # removes two files
$
```

2. `$rm programs/sample` #sample file under programs directory is removed
3. Directories are usually removed using the command `rmdir` but, when the recursive option `-r` is used all files and sub-directories under the current directory are removed.

```
$ rm -r*
```

```
$
```
4. To remove write protected files use `-f` option. To avoid accidental removal of files it is advisable to remove files interactively.

F) Find (find)

In a large file environment, it is difficult to find a given file. It is like a file search option in any OS environment.

Syntax

```
$find filename
```

Example

```
$find file23
```

Examples illustrating the behavior of the find command

1. Searches for the file `bin` on the basis of the name criterion in the entire file structure.

```
#find /-namebin -print
```

```
/root/bin
```

```
/root/home/bin
```

```
/usr/li/mc/bin
```

```
/usr/kerberos/bin
```

```
/bin
```

```
#
```
2. Searches for all the `awk` files on the basis of the `-name` option in the current directory because of the dot (`.`) character in the path list.

```
#find .-name "*.awk" -print
```

```
./marks.awk
```

UNIX PROGRAMMING UNIT – 2

```
./quest.awk  
./pl.awk  
#
```

3. Searches for all the files that have been modified or created within last 2 days because of -mtime option and the -2 argument in the selection criterion.

```
#find -mtime -2 -print  
./spel.chk  
./filesz  
./salary.sh  
#
```

4. Searches for the file sample in the current directory recursively and removes it because of the -exec option and the rm command with it.

```
#find .-name sample -exec rm {}?;
```

5. Searches for the shell script example.sh in the current directory and assigns execute permissions to all categories of its users. #find .-name example.sh -exec chmod x {}";"

6. Locates all the files in the current directory that have an extension .c in their names and are larger than 40 blocks in their size and displays path of all the selected files.

```
#find .-name "*.c" -size +40 -print
```

7. Searches all awk files in the current directory that have been modified within last 15 days.

```
#find .-name "*.awk" -mtime -15 -print
```

8. Locates all the regular files in the home directory and displays their path. # find \$HOME -type f -print

9. Searches the entire file system recursively for the files owned by either joshi or paul.

```
#find /\ (-user joshi -o -user paul \) -print -exec rm {} \;
```

10. Selects all the files in the current directory that are not owned by the user jacob as the selection criterion is made up of the negation (!) operator and the -user option.

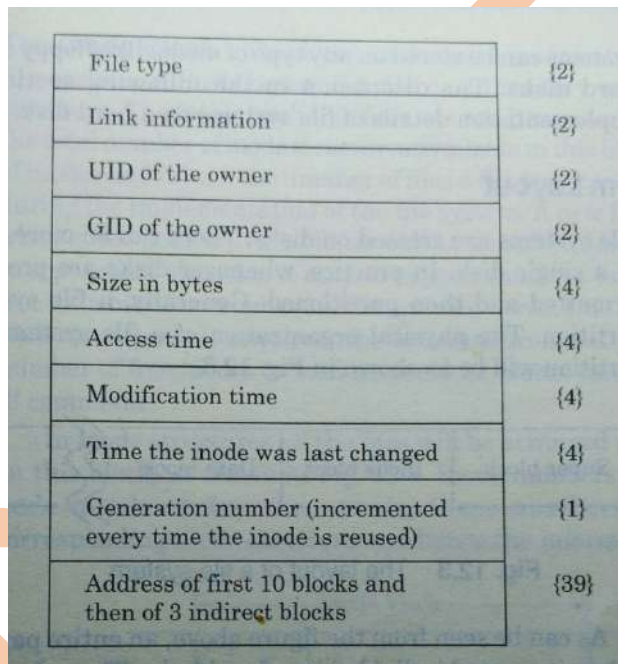
```
#find . !-user jacob -print
```

UNIX PROGRAMMING UNIT – 2

6) INODES

As soon as a file is created, the kernel allocates a unique inode number to that file. An inode number is a positive integer number, the maximum value of which depends on the maximum number of total inode numbers of the file system. The maximum value of the total inodes is decided by the person who creates the file system.

It is through these inode values that physical files on a secondary storage area are accessed. Actually inode stands for index node. These numbers are called index nodes as these numbers are used as indexes to access any required inode structure.



File type	{2}
Link information	{2}
UID of the owner	{2}
GID of the owner	{2}
Size in bytes	{4}
Access time	{4}
Modification time	{4}
Time the inode was last changed	{4}
Generation number (incremented every time the inode is reused)	{1}
Address of first 10 blocks and then of 3 indirect blocks	{39}

Figure 8. An inode structure

The inode structures will be housed in a separate block called the inode block on the secondary storage medium of 64 -byte long. As soon as an inode is allocated to a file, the corresponding inode structure gets filled up with relevant information such as file type, it's link information, size, times associated with it and so on of the file. Neither the file name nor it's inode number would be present in inode structure.

If required, the inode value of a file can be known using the ls command with the option -i, as shown in the example.

```
$ls -imyfile
1372 myfile
$
```

Usually inode number 1 will be reserved for bad blocks handling and inode number 2 will be reserved for root directory.

7)THE DIRECTORY HIERARCHY

In Unix all related files are grouped into a single group. For example, all binary files are grouped together, all temporary files are grouped together and all device files are grouped together. Each group constitutes a directory or a sub-directory and is referred by an appropriate name. All the grouped files that is directories, sub-directories are arranged in the form of an inverted tree like hierarchical structure as shown in figure. This inverted tree like organization is called the file system.

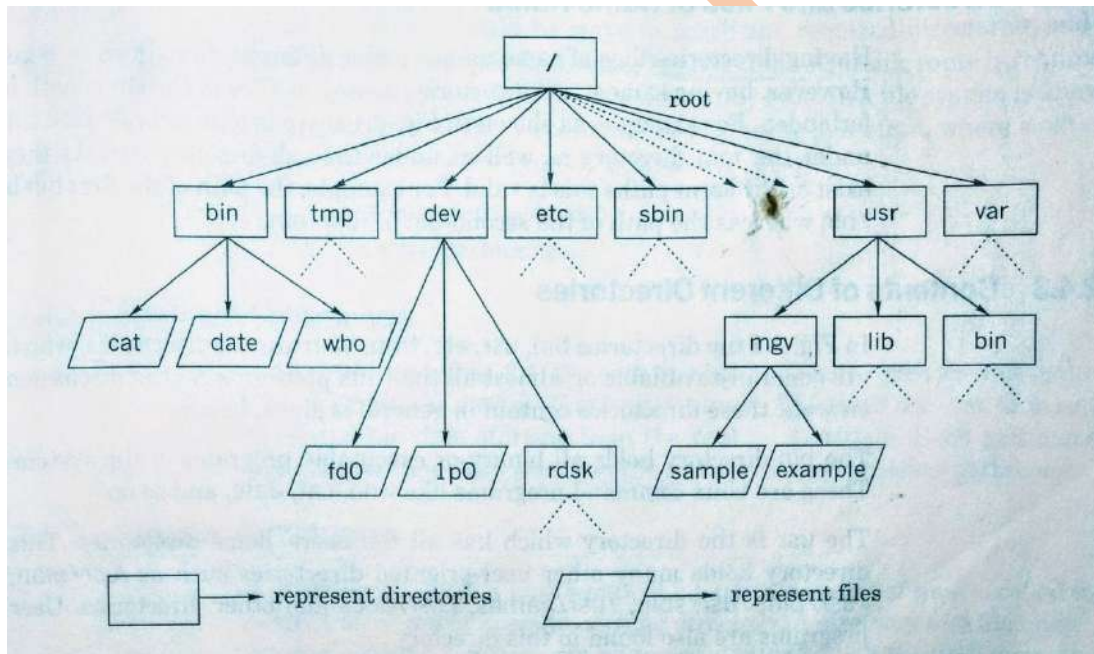


Figure 9. A typical Unix file system

There is a reference point for all files, directories and sub-directories and is known as root directory. The root directory is represented by forward slash (/).

The root will have many number of sub-directories and in turn they may have sub-directories or files within them. Thus, bin, dev, etc and usr are the sub-directories to the root directory. cat, date and who are Unix program files under the sub-directory bin and so on. Leaf nodes always represent either a regular file or a special file, that is, a device file.

1.Parent -child Relationship

There is parent-child relationship between directories, sub-directories and files. For example, fd0 is the child of the directory dev which itself is the child of the root directory. The root directory will not have any parent.

UNIX PROGRAMMING UNIT – 2

2.Directories and Files of Same Name

Directories/files of same names under different directories is valid. However, having same name directories as well as files in the same path is forbidden. The bin directory is there in the root directory as well as in sub-directory usr as they exist in different paths this is valid. The path of first bin is /bin whereas the path of the second bin is /usr/bin.

3.Contents of Different Directories

1. The bin directory holds all binary or executable programs of the system. These are Unix command programs like who, cat, date and so on.
2. The usr is the directory which has all the user's home directories. This holds user-oriented directories such as /usr/man, /usr/bin, /usr/sbin, /usr/games, /usr/docs and other directories.
3. The etc directory holds all configuration files of the system. Sometimes it can also holds some system administrative command files.
4. The/sbin directory has system files that are usually run automatically by the Unix system.
5. The dev directory holds device files under it. These are special files that represent the computer components such as keyboard, printer or disk. For example, the terminal on which one works is one of the /dev/tty files.
6. The var directory holds information that varies frequently. For example, user mailboxes that are found in the /var/mail directory.
7. The tmp directory contains the temporary files created either by the users or by the Unix. Generally, these files are deleted when the system is shut down or restarted.

Thus, in every group a directory is made or created for a specific purpose and all interrelated files are put within them.

/	root of the file system
/bin	essential programs in executable form ("binaries")
/dev	device files
/etc	system miscellany
/etc/motd	login message of the day
/etc/passwd	password file
/lib	essential libraries, etc.
/tmp	temporary files; cleaned when system is restarted
/unix	executable form of the operating system
/usr	user file system
/usr/adm	system administration: accounting info., etc.
/usr/bin	user binaries: troff, etc.
/usr/dict	dictionary (words) and support for spell(1)
/usr/games	game programs
/usr/include	header files for C programs, e.g. math.h
/usr/include/sys	system header files for C programs, e.g. inode.h
/usr/lib	libraries for C, FORTRAN, etc.
/usr/man	on-line manual
/usr/man/man1	manual pages for section 1 of manual
/usr/mdec	hardware diagnostics, bootstrap programs, etc.
/usr/news	community service messages
/usr/pub	public oddments: see ascii(7) and eqnchar(7)
/usr/src	source code for utilities and libraries
/usr/src/cmd	source for commands in /bin and /usr/bin
/usr/src/lib	source code for subroutine libraries
/usr/spool	working directories for communications programs
/usr/spool/lpd	line printer temporary directory
/usr/spool/mail	mail in-boxes
/usr/spool/uucp	working directory for the uucp programs
/usr/sys	source for the operating system kernel
/usr/tmp	alternate temporary directory (little used)
/usr/you	your login directory
/usr/you/bin	your personal programs

Table 1. Interesting Directories

8)FILE ATTRIBUTES AND PERMISSIONS

i) Ownership of a file

The person who actually creates a file will be the owner of that file. The owner of the file is also called the user. Among the three types of users i.e., owner, group and others in the domain of Unix, the owner has special privilege- the ability to modify permissions of the file of their own or group or others. The supervisor or system administrator also enjoys this power.

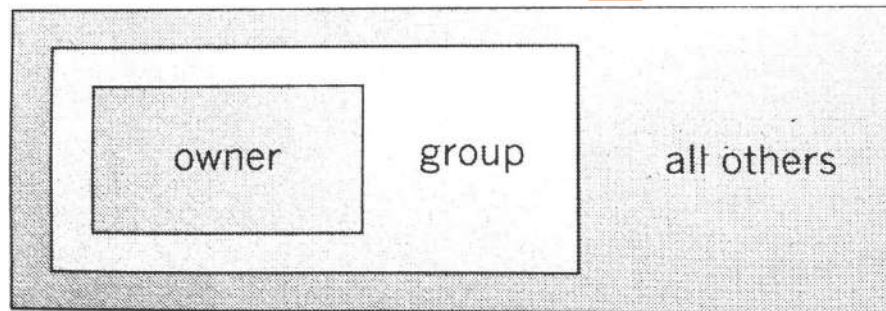


Figure 10. Types of users

User(owner) names are available in a file called the `/etc/passwd` file and the group names are available in `/etc/group` file.

UNIX actually keeps track of owners and group as numbers rather than as names. User Identification numbers(UIDs) are mapped to user names in the `/etc/passwd` file and group Identification numbers (GIDs) are mapped to group names in the `/etc/group` file.

ii) FILE ATTRIBUTES

Any type of file will have (1) a name (2) creation, modification and access times. (3) a size (4) an owner (5) group to which the owner belongs to. (6) Link Information. (7) permissions (8) inode number associated with it. All this information about a file are called it's attributes.

iii) File Permissions

Unless otherwise permitted, no one is allowed to access and use a file. A file may be accessed for one or more of the following purposes

1. Reading
2. Writing
3. Executing

UNIX PROGRAMMING UNIT – 2

These permissions may differ depending on the category of users.

User/Owner

Group

Others

By default, the owner will have only read and write permissions, and the group and others will have only read permission.

In case of regular files, read, write and execute permissions allows the users to read, write and execute the files respectively.

Example

`-rw-r- - r- -`

The file and directory permissions are depicted as,

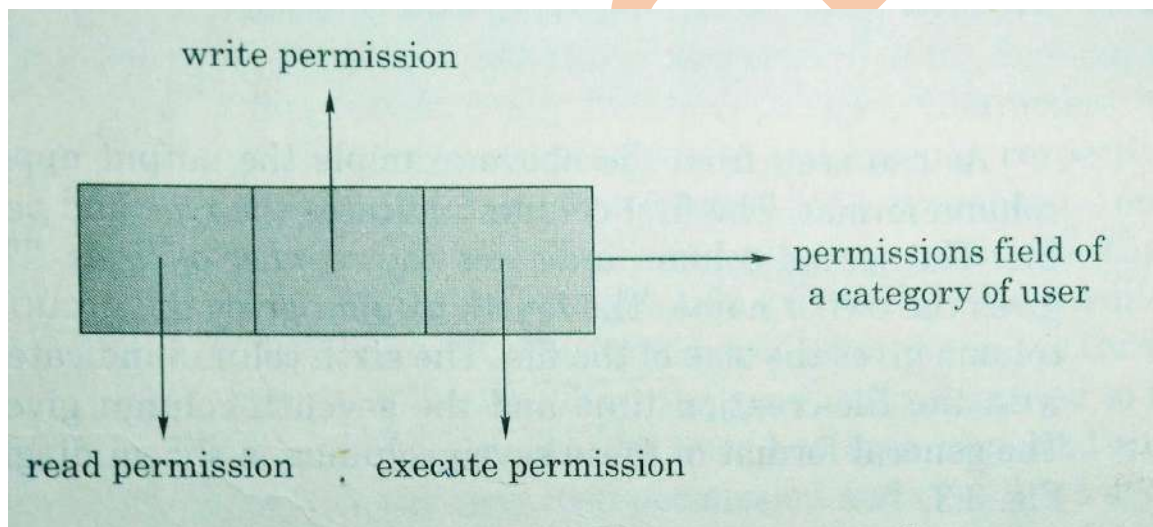


Figure 11. Permissions fields of a category of user

9) FILE PERMISSION COMMANDS

a) The file COMMAND -KNOWING THE FILE TYPE

1. The file command is used to identify the type of the files on the basis of their contents. When this command is used, it reads either the header or first few hundreds of bytes of the file and an educated guess is made on the type of the file.
2. Certain category of files such as executables are recognized by the information stored on their headers - the information stored in the first-byte. This first-byte information is known as the magic number.

UNIX PROGRAMMING UNIT – 2

3. The correlation between magic numbers and file types is contained in the file /etc/magic. For example the octal 410 is the magic number of executable files. These magic numbers can be verified by taking the octal dump of the relevant file.
4. For text files, the clues may not be available directly with the magic numbers. Rather, such clues will be available deeper in the file. Example, the clue for identifying the text files could be, the use of a new line character at the end of every line. The presence of words such as #include indicate a C source file, lines beginning with a period may indicate nroff or troff and so on.

```
Examples $file
mgvmgv:ASCII
I text
$file /bin
/bin: directory $file
mac.cmac.c: ASCII C
program text
$touch liju $file
lijuliju: empty $cd
/bin $file cshesh:
symbolic link tcsh
$
```

In all the examples shown above, filenames have been given in the form of relative pathnames. Filenames can be given in the form of absolute pathnames also. Here, it may be recalled that the listing command ls with the flag option F also gives an idea about the file types but in a limited way.

b)THE chmod COMMAND- CHANGING FILE PERMISSIONS

The chmod command is used to change the permissions of a file after its creation. Only the owner or the super user can change file permissions.

Syntax

```
$ chmodassignment_expression filename
```

The assignment expression holds the following information.

1. The information about the category of users { user -u, group -g, others -o, all -a}.
2. The information about granting or denial of the permission { the operators +, - and =}.
3. The information about the type of permission { read -r, write -w, execute -x}.

UNIX PROGRAMMING UNIT – 2

Example1

```
$chmodu+x sample
```

```
$ls -l sample
```

```
-rwxr- -r-- 1 mgvcsd 5180 Jan 07 12:06 sample
```

```
$
```

Here, sample is file name u+x is the argument expression where, u stands for user, x for execution and + for granting.

```
$ chmodugo+x sample; ls -l sample
```

```
-rwxr-xr-x 1 mgvcsd 5180 Jan 07 12:06 sample
```

```
$
```

Example2

ugo (user,group and others) can also written as a (all).

So, ugo+x can also written as a+x.

```
$chmoda+x sample; ls -l sample or
```

```
$chmod +x sample; ls -l sample
```

Example3

The chmod command can work on more than one file at a time as shown in the following example.

```
$chmodu+x sample1 sample2 sample3
```

```
$ls -l sample1 sample2 sample3
```

```
-rwxr--r-- 1 mgvcsd 5180 Jan 07 12:06 sample1
```

```
-rwxr--r-- 1 mgvcsd 6191 Jan 07 01:16 sample2
```

```
-rwxr--r-- 1 mgvcsd 7101 Jan 07 02:26 sample3
```

```
$
```

Example4

```
$chmodu-x,go+x sample
```

```
$ls -l sample
```

```
-rw-r-xr-x 1 mgvcsd 5180 Jan 12:06 sample
```

```
$
```

UNIX PROGRAMMING UNIT – 2

a)Relative and Absolute Permissions Assignment

The changes made were relative to the present settings. In other words, an expression like `u+x` sets the execute permissions to the user. It will not disturb other settings of either this or any other category. This type of permission assignment is called relative permission assignment. The use of the `=` operator in the `chmod` expression assigns or grants only specified permissions and removes all other permissions. This type of granting permissions is called absolute permission assignment.

Example

```
$chmod a=r sample; ls -l sample
-r - - r - - r - - 1 mgvcsd 5180 Jan 07 12:06 sample
$
```

From the output of the above example, one may observe that all have been given read permissions after removing the permissions associated with the file earlier.

b)Permissions with Octal Numbers

File permissions can also be assigned using octal numbers. In this representation

1. $4_8(100_2)$ assigns read permission, $2_8(010_2)$ assigns write permission and $1_8(001_2)$ assigns the execute permission and so on.
2. Permission assignments made using octal numbers are always absolute assignments. In other words, octal numbers cannot be used for relative permissions assignment.

For example, a $6_8(110_2)$ assigns both read and write permissions and denies the execute permission $5_8(101_2)$ assigns read and execute permissions and denies write permission. Because there are 3 categories of users, one has to use three octal digits in the expression field, as shown in the following example.

Example1

```
$chmod 644 sample; ls -l sample
-rw-r--r-- 1 mgvcsd 5180 Jan 07 12:06 sample
$
```

Example2

The `$chmod 761 sample` is the octal notation equivalent of the following command.

```
$chmod u=rwx, g=rw, o=x sample
$
```

UNIX PROGRAMMING UNIT – 2

- Permissions can be granted to all the files and sub-directories in a directory by using the recursive option (-R) with the chmod command. For this the argument must be the directory name. For example, the execute permission to all category of users with respect to all files and directories under the current directory can be granted using the command given below.

Example3

```
$chmod -R a+x
```

```
$
```

C)THEchown COMMAND- CHANGING THE OWNER OF A FILE

Every file has a owner. When a file is created, the creator becomes the owner of the file. Only the owner can change the major attributes of a file (ofcourse, the system administrator also can do it). Sometimes it is necessary to change the ownership of a file.

- There are two ways inwhich the ownership can be changed- by copying the file into the target user's directory, and by using the chown command.

For example, the file sample from the directory of hmk is copied to the home directory of someone else, say mgv. Then mgv becomes the new owner of the file sample. If, now, the oldfile and newfile are listed using the ls -l command, one sees that every detail will be same except the owner.

- The copying method of changing the ownership has the following disadvantages:
 - It creates an additional file and thus uses extra space.
 - the new owner should have the knowledge about the permissions of the file.
- Changing the owner of a file using the chown command is more simpler and direct method of changing the ownership. This command takes two arguments, login name of the new user and the name of the file. An example is given below.

```
$ls -l sample
```

```
-rwxr- - r-x 1 rajcsd425 May 10 20:30 sample
```

```
$chown Kumar sample; ls -l sample
```

```
-rwxr- - r-x 1 kumarsd 425 May 10 20:30 sample $
```

UNIX PROGRAMMING UNIT – 2

- Ownership once surrendered cannot be reinstated. Also moving a file doesn't change the ownership. Further this command can use the -R option-- the recursive option. When this option is used the ownership of all the files in the current directory are changed.

d) THE chgrp COMMAND - CHANGING THE group OF A FILE

- In Unix, all files not only belong to an owner but also to a group.
- One may need to change the group of a file and certain circumstances such as when new groups are set up on a system or when files are copied to a new system. This is done by using the chgrp command.
- Only the owner of a file can change the group (ofcourse, the system administrator also can do the same). Changing the group using the chgrp command is also straight forward and takes two arguments; the name of the new group and the name of the file.

Example

```
$chgrp planning sample  
$
```

As shown above, the name of the new group must appear as the first argument and the name of the file has to appear as the second argument. The recursive option-R can also be used with this command. When used with the -R option, the group of all the files under current directory is changed.

UNIX PROGRAMMING UNIT – 3

Using the Shell

A **Shell** provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output. Shell is an environment in which we can run our commands, programs, and shell scripts.

COMMAND LINE STRUCTURE

Command is a program that tells the unix system to do something.

Syntax:

Command [options] [arguments]

- Where an argument indicates on what the command isto perform its action, usually a file or a series of a file
- An option modifies the command, changing the way its performs.
- Commands are case sensitive.

Ex :

Command or commands are not the same.

Options are generally preceded by hyphen (-)

For most commands, more than one operation can be strung together.

Syntax:

Command -[option1] -[option2] -[option3]

Ex :

\$ ls -a -l -R

Options and syntax for commands are listed in the main page for the command

META CHARACTERS

Unix Shell provides various meta characters which have special meaning while using them in any Shell Script and causes termination of a word unless quoted.

The Asterisk as a Metacharacter (*)

The asterisk (*) is a universally known metacharacter. It means zero or more of any character when searching for a pattern. For example:

ls *.c

UNIX PROGRAMMING UNIT – 3

```
cselab2@cselab2-OptiPlex-7020: ~
cselab2@cselab2-OptiPlex-7020:~$ ls
a.out  Documents  examples.desktop  Pictures  selects.c  Videos
Desktop Downloads Music      Public  Templates
cselab2@cselab2-OptiPlex-7020:~$ ls*.c
ls*.c: command not found
cselab2@cselab2-OptiPlex-7020:~$ ls *.c
selects.c
cselab2@cselab2-OptiPlex-7020:~$ vi madhavi
cselab2@cselab2-OptiPlex-7020:~$ sh madhavi
naga madhavi latha kakarla
cselab2@cselab2-OptiPlex-7020:~$ ls
a.out  Documents  examples.desktop  Music  Public  Templates
Desktop Downloads madhavi      Pictures  selects.c  Videos
cselab2@cselab2-OptiPlex-7020:~$ vi latha
cselab2@cselab2-OptiPlex-7020:~$ vi latha.c
cselab2@cselab2-OptiPlex-7020:~$ ls
a.out  Documents  examples.desktop  latha.c  Music  Public  Templates
Desktop Downloads latha      madhavi  Pictures  selects.c  Videos
cselab2@cselab2-OptiPlex-7020:~$ vi ronith
cselab2@cselab2-OptiPlex-7020:~$ ls *.c
latha.c selects.c
cselab2@cselab2-OptiPlex-7020:~$
```

The Carat as a Metacharacter (^)

The carat (^) is used to denote the start of a line or a string. So how is it used?

The ls command lists the files in a folder, as follows:

ls -a

If you want to list the files in a folder that begin with a certain string, for example, gnome, the carat can be used to specify that string. For example:

ls -a | grep ^.dash

This lists the files that start with .dash. If you want files that have .dash anywhere in the filename, use the asterisk.

```
cselab2@cselab2-OptiPlex-7020: ~
cselab2@cselab2-OptiPlex-7020:~$ sh madhavi
naga madhavi latha kakarla
cselab2@cselab2-OptiPlex-7020:~$ ls -a
.          .config      .gconf       Pictures    Videos
.          .dbus        .ICEauthority .pki       .Xauthority
a.out     Desktop      latha        .profile   .xsession-errors
bash_history .dmrc       latha.c     Public     .xsession-errors.old
bash_logout Documents    local        ronith
bashrc    Downloads   madhavi     selects.c
cache     examples.desktop .mozilla    .ssh
compiz    .ex.swp     Music       Templates
cselab2@cselab2-OptiPlex-7020:~$ ls -a | grep ^.bash
.bash_history
.bash_logout
.bashrc
cselab2@cselab2-OptiPlex-7020:~$ ls -a | grep ^m
madhavi
cselab2@cselab2-OptiPlex-7020:~$
```


UNIX PROGRAMMING UNIT – 3

The Full Stop as a Meta character (.)

The Full Stop (.) indicates the current position when running commands such as `cd`, `find`, or `sh`. In applications such as `awk`, `grep`, and `sed`, it's a wildcard that denotes a specific number of any character.

Now look at this command:

```
ls | grep r..ith
```

The pipe (|) metacharacter sends that list to the `grep` command, which searches for any line in the list that contains `r..ith`, where the periods refers to two of any character.

The Dollar Symbol as a Metacharacter (\$)

The dollar symbol (\$) has multiple meanings as a metacharacter in Linux. When used to match patterns, it means the opposite of carat and denotes any pattern that ends with a particular string. For example:

```
ls | grep th$
```

This lists all files that end with `th`.

The dollar symbol is also used to access environment variables within the bash shell. For example:

```
#!/bin/bash
export name=MADHAVI
echo $name
```

The line `export name=MADHAVI` creates an environment variable called `name` and sets its value `MADHAVI`. To access the environment variable, use the \$ symbol. With the \$ symbol, the `echo $name` statement displays `MADHAVI`. Without it, the `echo name` statement displays the word `name`.

Escaping Metacharacters (\)

Sometimes you don't want the metacharacter to have a special meaning. For example, if a file is called `l.tha` and another file is called `latha`.

Now look at the following command:

```
ls | grep l.tha
```

What do you think is returned? Both `l.tha` and `latha` are returned because both match the pattern. To only return `l.tha`, escape the full stop to actually mean a full stop, as follows:

```
ls | grep l\\.tha
```

UNIX PROGRAMMING UNIT – 3

```
cselab2@cselab2-OptiPlex-7020: ~  
cselab2@cselab2-OptiPlex-7020:~$ ls | grep ro...  
rohan  
rohith  
ronith  
cselab2@cselab2-OptiPlex-7020:~$ ls | grep ro.  
rohan  
rohith  
ronith  
cselab2@cselab2-OptiPlex-7020:~$ ls | grep ro....  
rohith  
ronith  
cselab2@cselab2-OptiPlex-7020:~$ ls | grep th$  
rohith  
ronith  
cselab2@cselab2-OptiPlex-7020:~$ export name=MADHAVI  
cselab2@cselab2-OptiPlex-7020:~$ echo $name  
MADHAVI  
cselab2@cselab2-OptiPlex-7020:~$ ls | grep l.tha  
latha  
latha.c  
l.tha  
cselab2@cselab2-OptiPlex-7020:~$ ls | grep l\\.tha  
l.tha  
cselab2@cselab2-OptiPlex-7020:~$ █
```

Brackets as a Metacharacter ([])



We can use brackets ([]) when searching for patterns. Brackets specify specific letters to match anywhere in the pattern. For example:

```
ls | grep [mrl]
```

This lists all files that contain the letters m, r, or l.

```
cselab2@cselab2-OptiPlex-7020:~$ ls | grep [mrl]  
Documents  
Downloads  
examples.desktop  
latha  
latha.c  
l.tha  
madhavi  
Pictures  
Public  
rohan  
rohith  
ronith  
selects.c  
Templates  
cselab2@cselab2-OptiPlex-7020:~$ ls | grep [a-l]  
a.out  
Desktop  
Documents  
Downloads  
examples.desktop  
latha  
latha.c  
l.tha  
madhavi  
Music  
Pictures  
Public  
rohan  
rohith  
ronith  
selects.c  
Templates  
Videos  
cselab2@cselab2-OptiPlex-7020:~$ █
```

The Accent Grave Metacharacter (`)

In the examples above, the pipe metacharacter sends the results of one command (like the ls command) to another command (like the grep command).

UNIX PROGRAMMING UNIT – 3

An alternative way to do this is to use the back quote, also known as the accent grave (`), to insert the results of one command into another command. To do this, store the result of one command in a variable. For example:

```
command=`ls -lah`  
echo $command
```

```
cselab2@cselab2-OptiPlex-7020:~$ command=`ls -lah`  
cselab2@cselab2-OptiPlex-7020:~$ echo $command  
total 172K drwxr-xr-x 19 cselab2 cselab2 4.0K Jun 16 09:14 . drwxr-xr-x  
3 root root 4.0K Jun 26 2019 .. -rwxrwxr-x 1 cselab2 cselab2 12K Dec 9 2  
019 a.out -rw----- 1 cselab2 cselab2 2.4K Jun 13 12:28 .bash_history -  
rw-r--r-- 1 cselab2 cselab2 220 Jun 26 2019 .bash_logout -rw-r--r-- 1 cs  
elab2 cselab2 3.6K Jun 26 2019 .bashrc drwx----- 20 cselab2 cselab2 4.0  
K Dec 30 16:31 .cache drwx----- 3 cselab2 cselab2 4.0K Jun 26 2019 .com  
piz drwx----- 19 cselab2 cselab2 4.0K Jun 13 10:11 .config drwx----- 3  
cselab2 cselab2 4.0K Sep 30 2019 .dbus drwxr-xr-x 3 cselab2 cselab2 4.0  
K Jun 16 09:22 Desktop -rw-r--r-- 1 cselab2 cselab2 25 Jun 26 2019 .dmrc  
drwxr-xr-x 2 cselab2 cselab2 4.0K Jun 26 2019 Documents drwxr-xr-x 2 cs  
elab2 cselab2 4.0K Dec 16 2019 Downloads -rw-r--r-- 1 cselab2 cselab2 8.  
8K Jun 26 2019 examples.desktop -rw----- 1 cselab2 cselab2 12K Jun 13  
10:10 .ex.swp drwx----- 3 cselab2 cselab2 4.0K Jun 16 08:48 .gconf -rw-  
----- 1 cselab2 cselab2 13K Jun 16 08:48 .ICEauthority -rw-rw-r-- 1 cse  
lab2 cselab2 0 Jun 13 10:49 latha -rw-rw-r-- 1 cselab2 cselab2 0 Jun 13  
10:49 latha.c drwxr-xr-x 3 cselab2 cselab2 4.0K Jun 26 2019 .local -rw-r  
w-r-- 1 cselab2 cselab2 0 Jun 13 12:25 l.tha -rw-rw-r-- 1 cselab2 cselab  
2 36 Jun 13 10:44 madhavi drwx----- 4 cselab2 cselab2 4.0K Aug 21 2019  
.mozilla drwxr-xr-x 2 cselab2 cselab2 4.0K Jun 26 2019 Music drwxr-xr-x  
2 cselab2 cselab2 4.0K Jun 26 2019 Pictures drwx----- 3 cselab2 cselab2  
4.0K Nov 22 2019 .pki -rw-r--r-- 1 cselab2 cselab2 675 Jun 26 2019 .pro  
file drwxr-xr-x 2 cselab2 cselab2 4.0K Jun 26 2019 Public -rw-rw-r-- 1 c  
selab2 cselab2 0 Jun 16 09:14 rohan -rw-rw-r-- 1 cselab2 cselab2 0 Jun 1  
6 09:10 rohith -rw-rw-r-- 1 cselab2 cselab2 0 Jun 13 10:49 ronith -rw-rw  
-r-- 1 cselab2 cselab2 4.2K Dec 9 2019 selects.c drwx----- 2 cselab2 cs  
elab2 4.0K Aug 6 2019 .ssh drwxr-xr-x 2 cselab2 cselab2 4.0K Jun 26 2019  
Templates drwxr-xr-x 2 cselab2 cselab2 4.0K Jun 26 2019 Videos -rw-----  
-- 1 cselab2 cselab2 198 Jun 16 08:48 .xauthority -rw----- 1 cselab2 c  
selab2 347 Jun 16 08:48 .xsession-errors -rw----- 1 cselab2 cselab2 1.  
1K Jun 13 12:29 .xsession-errors.old  
cselab2@cselab2-OptiPlex-7020:~$
```

The Shell Metacharacters are listed here for reference.

Symbol	Meaning
>	Output redirection, (see File Redirection)
>>	Output redirection (append)
<	Input redirection
*	File substitution wildcard; zero or more characters
?	File substitution wildcard; one character
[]	File substitution wildcard; any character between brackets
`cmd`	Command Substitution
\$(cmd)	Command Substitution
	The Pipe ()
;	Command sequence, Sequences of Commands
	OR conditional execution
&&	AND conditional execution
()	Group commands, Sequences of Commands
&	Run command in the background, Background Processes
#	Comment

UNIX PROGRAMMING UNIT – 3

Symbol	Meaning
\$	Expand the value of a variable
\	Prevent or escape interpretation of the next character
<<	Input redirection (see Here Documents)

CREATING NEW COMMANDS:

When set of commands are required to be repeated for specific no of times, then it would be better to make them in to a new command.

- They can be assigned with user defined names.
- Users can use them like regular commands.

Example:1

```
$pwd
/home/cselab2
$echo 'pwd' > KNML           # Redirecting pwd command to file Called KNML
$cat KNML
pwd
$sh < KNML                  #Executing KNML file like pwd command using sh command
/home/cselab2
```

Example:2

```
$pwd
/home/cselab2
$echo 'pwd' > KNML           # Redirecting pwd command to file Called KNML
$cat KNML
pwd
$mkdir bin
$echo $PATH
/home/cselab2/bin
$mv KNML bin
$cd bin
```

UNIX PROGRAMMING UNIT – 3

```
cselab2@cselab2-OptiPlex-7020: ~/abc
cselab2@cselab2-OptiPlex-7020:~$ pwd
/home/cselab2
cselab2@cselab2-OptiPlex-7020:~$ echo 'pwd' > KNML
cselab2@cselab2-OptiPlex-7020:~$ cat KNML
pwd
cselab2@cselab2-OptiPlex-7020:~$ mkdir madhavi
mkdir: cannot create directory 'madhavi': File exists
cselab2@cselab2-OptiPlex-7020:~$ pwd
/home/cselab2
cselab2@cselab2-OptiPlex-7020:~$ echo 'pwd' > KNML
cselab2@cselab2-OptiPlex-7020:~$ cat KNML
pwd
cselab2@cselab2-OptiPlex-7020:~$ mkdir abc
cselab2@cselab2-OptiPlex-7020:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
cselab2@cselab2-OptiPlex-7020:~$ mv KNML abc
cselab2@cselab2-OptiPlex-7020:~$ cd abc
cselab2@cselab2-OptiPlex-7020:~/abc$ chmod +x KNML
cselab2@cselab2-OptiPlex-7020:~/abc$ sh KNML
/home/cselab2/abc
cselab2@cselab2-OptiPlex-7020:~/abc$
```

Example:3(As an Internal command)

Example:4(As an External command)

```
cselab2@cselab2-OptiPlex-7020:~$ cd kartheek
cselab2@cselab2-OptiPlex-7020:~/kartheek$ sh praneeth
/home/cselab2/kartheek
cselab2@cselab2-OptiPlex-7020:~/kartheek$ cd
cselab2@cselab2-OptiPlex-7020:~$ who | wc -l
2
cselab2@cselab2-OptiPlex-7020:~$ echo 'who | wc -l' > sai
cselab2@cselab2-OptiPlex-7020:~$ who
cselab2  :0                2020-06-16 08:48 (:0)
cselab2 pts/0            2020-06-16 10:30 (:0)
cselab2@cselab2-OptiPlex-7020:~$ cat sai
who | wc -l
cselab2@cselab2-OptiPlex-7020:~$ sh < sai
2
cselab2@cselab2-OptiPlex-7020:~$ mkdir msp
cselab2@cselab2-OptiPlex-7020:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
cselab2@cselab2-OptiPlex-7020:~$ mv sai msp
cselab2@cselab2-OptiPlex-7020:~$ cd msp
cselab2@cselab2-OptiPlex-7020:~/msp$ chmod +x sai
cselab2@cselab2-OptiPlex-7020:~/msp$ sh sai
2
```

COMMAND ARGUMENTS AND PARAMETERS

The shell programs will interpret the arguments such as filenames and options while running the program.

Example:

\$ cx sample

This command is shorthand for

\$ chmod +x sample

The contents of cx are

chmod, +x and sample

UNIX PROGRAMMING UNIT – 3

When shell executes file containing commands, every occurrence of \$1 will be replaced by first argument and \$2 will be replaced by second argument and so on.

Let cx contain

```
$chmod +x $1
```

If the below command is run

```
$cx sample
```

Here sub shell will be replaced \$1 with first argument "sample"

Shell can handle even multiple arguments

```
$ chmod +x $1$2 $3 $4 $5
```

A shorthand notation for this would be

```
$ chmod +x $*
```

The argument \$0 represents the command to be executed.

PROGRAM OUTPUT AS ARGUMENTS

The shell allows the standard output of one command to be used as an argument of another command.

The shell executes the **command** enclosed within single quotes and replaces the command with standard output. This is called command substitution.

Syntax:

```
'command'
```

Ex:1 (command substitution)

```
cselab2@cselab2-OptiPlex-7020:~$ echo current date is `date`
current date is date
cselab2@cselab2-OptiPlex-7020:~$ echo current date is `date`
current date is Tue Jun 16 11:35:05 IST 2020
```

Ex:2 (command substitution to generate useful messages)

```
cselab2@cselab2-OptiPlex-7020:~$ who
cselab2  :0                2020-06-16 08:48 (:0)
cselab2  pts/0            2020-06-16 11:34 (:0)
cselab2@cselab2-OptiPlex-7020:~$ echo "current users working on the systems are `who | wc -l`"
current users working on the systems are 2
```

Ex:3 (command substitution in shell scripts)

```
cselab2@cselab2-OptiPlex-7020:~$ cat > sample.sh
echo Number of users logged on to the system are `who | wc -l`
echo The present Working Directory is `pwd`
cselab2@cselab2-OptiPlex-7020:~$ sh sample.sh
Number of users logged on to the system are 2
The present Working Directory is /home/cselab2
cselab2@cselab2-OptiPlex-7020:~$
```

UNIX PROGRAMMING UNIT – 3

SHELL VARIABLES:

- A Variable is a data name used to store data value.
- Variables are defined and used with a shell.
- Shell Variables are three types
 - i. System Variables
 - ii. Local Variables
 - iii. Read-only Variables

i. SYSTEM Variables:

These variables are also called as Environment Variables.

- These variables are set either during boot sequence or immediately after logging in.
- The working environment, under which a user works, depends entirely upon the values of these variables. These are similar to global variables.
- Represented in Uppercase letters
- The different system variables are
 - a. PATH
 - b. HOME
 - c. IFS
 - d. MAIL
 - e. SHELL
 - f. TERM

a) The **PATH** variable:

The PATH environment variable has a special format. Let's see what it looks like:

```
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/sbin:/usr/sbin:
```

It's essentially a :-separated list of directories. When you execute a command, the shell **searches through each of these directories, one by one**, until it finds a directory where the executable exists.

b) The **HOME** variable:

It indicates the home directory of the current user: the default argument for the cd **built-in** command.

Ex:

```
$ echo $home
/home/501
```

c) The **IFS** variable:

It indicates the **Internal Field Separator** that is used by the parser for word splitting after expansion.

The default tokens are the three white space tokens

- Space
- Tab
- New line

K Varada Rajkumar

Assistant Professor

Department of CSE

Sir C R Reddy College Of Engineering

UNIX PROGRAMMING UNIT – 3

Because all these are non-printable characters, they can be seen or verified using the `od` command.

Example:

```
$echo "$IFS" | od -bc
0000000 040 011 012 012
          \t \n \n
```

0000004

The option `-b` displays octal value of each character.

The option `-c` displays the character itself.

d) The **MAIL** variable:

- This variable holds the absolute pathname of the file where user's mail is kept
- Usually the name of this file is the user's login name

Example:

```
$echo $MAIL
/var/spool/mail/5
```

e) The **SHELL** variable:

- This variable contains the name of the users shell program in the form of absolute pathname
- System administrator sets the default shell
- If required, user can change it
- The value of the variable `SHELL` may be known by `echo` command.

Example:

```
$echo $SHELL
/bin/bash
```

f) The **TERM** variable:

- This variable holds the information regarding the type of the terminal being used.
- If `TERM` is not set properly, utilities like `vi` editor will not work.

Example:

```
$echo $term
xterm
$
```

OTHER SYSTEM VARIABLES:

(a) The **LOGNAME** variable:

The variable `LOGNAME` holds the user name.

```
$ echo $LOGNAME
501
```

(b) The **TZ** variable:

TZ refers to Time zone. It can take values like `GMT`, `AST`, etc.

```
$echo $TZ
```


UNIX PROGRAMMING UNIT – 3

(c) The PS1 variable:

It holds the primary prompt value(\$)
\$echo \$PS1

(d) The PS2 variable:

It holds the secondary prompt value(>, right chevron)
\$echo \$PS2

ii) LOCAL variables:

- These variables are also called as User-defined Variables.
- These variables are defined and used by specific users.
- These variables are local to the user's shell environment.
- Rules for constructing variable Names:
 - Shell variable names are constructed using only alphanumeric(alphabets and digits) characters and the Underscore (_)
 - It starts with a letter.
 - The variable names are case-sensitive.

Example:

SUM, sum, Sum, suM, sUm are different.
Spaces not allowed.

Defining a Shell V variable:

Shell variables are evaluated by prefixing the variable name with a \$
Syntax:

\$variable=value

Example:

- a) \$x=20
- b) \$y=5.37
- c) \$z=sachin
- d) Sw ="india is my country"

iii) READ-ONLY variables:

- These variables uses readonly() function.
- The values of variables which can only be read but not to be manipulated are called read-only variables.

Example:

```
$cat example
echo Enter value for x
read x
echo value of x is $x
readonly x
x='expr $x+1'
echo the value of x now is $x
```

UNIX PROGRAMMING UNIT – 3

Execution:

\$sh example

Enter value for x

Value of x is 4

example: line 5 : x : readonly variable

MORE ON I/O REDIRECTION

It is possible to change the source from where the input is taken by a program as well as the destination to where the output is sent by a program. This mechanism of changing the input source and /or destination is called Redirection.

Symbol	Name	Redirection
<	Less than	Standard input redirection
>	Greter than	Standard output redirection
>>	Double Greater than	Standard output redirection with appending

Output Redirection

The output from a command normally intended for standard output can be easily diverted to a file instead. This capability is known as output redirection.

If the notation > file is appended to any command that normally writes its output to standard output, the output of that command will be written to file instead of your terminal.

Check the following **who** command which redirects the complete output of the command in the users file.

```
$ who > users
```

Notice that no output appears at the terminal. This is because the output has been redirected from the default standard output device (the terminal) into the specified file. You can check the users file for the complete content –

```
$ cat users
```

```
oko    tty01  Sep 12 07:30
```

```
ai     tty15  Sep 12 13:32
```

```
ruth   tty21  Sep 12 10:10
```

```
pat    tty24  Sep 12 13:07
```

```
steve  tty25  Sep 12 13:03
```

```
$
```

If a command has its output redirected to a file and the file already contains some data, that data will be lost. Consider the following example –

```
$ echo line 1 > users
```

```
$ cat users
```

```
line 1
```

```
$
```

You can use >> operator to append the output in an existing file as follows –

```
$ echo line 2 >> users
```

UNIX PROGRAMMING UNIT – 3

```
$ cat users  
line 1  
line 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. As the **greater-than character** > is used for output redirection, the **less-than character** < is used to redirect the input of a command.

The commands that normally take their input from the standard input can have their input redirected from a file in this manner. For example, to count the number of lines in the file *users* generated above, you can execute the command as follows –

```
$ wc -l users  
2 users  
$
```

Upon execution, you will receive the following output. You can count the number of lines in the file by redirecting the standard input of the **wc** command from the file *users* –

```
$ wc -l < users  
2  
$
```

Note that there is a difference in the output produced by the two forms of the **wc** command. In the first case, the name of the file *users* is listed with the line count; in the second case, it is not. In the first case, **wc** knows that it is reading its input from the file *users*. In the second case, it only knows that it is reading its input from standard input so it does not display file name.

LOOPING IN SHELL PROGRAMS

Looping Statements in Shell Scripting: There are total 3 looping statements which can be used in bash programming.

1. while statement
2. for statement
3. until statement

To alter the flow of loop statements, two commands are used they are,

- A. break
- B. continue

Their descriptions and syntax are as follows:

while statement:

Here command is evaluated and based on the result loop will executed, if command raise to false then loop will be terminated

Syntax

while command

K Varada Rajkumar

Assistant Professor

Department of CSE

Sir C R Reddy College Of Engineering

UNIX PROGRAMMING UNIT – 3

```
do
  Statement to be executed
done
```

for statement:

The for loop operate on lists of items. It repeats a set of commands for every item in a list. Here var is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

Syntax

```
for var in word1 word2 ...wordn
do
  Statement to be executed
done
```

until statement:

The until loop is executed as many as times the condition/command evaluates to false. The loop terminates when the condition/command becomes true.

Syntax

```
until command
do
  Statement to be executed until command is true
done
```

Example Programs

Example 1:

Implementing for loop with break statement

```
#Start of for loop
for a in 1 2 3 4 5 6 7 8 9 10
do
  # if a is equal to 5 break the loop
  if [ $a == 5 ]
  then
    break
  fi
  # Print the value
  echo "Iteration no $a"
done
```

UNIX PROGRAMMING UNIT – 3

Output

```
$bash -f main.sh
Iteration no 1
Iteration no 2
Iteration no 3
Iteration no 4
```

Example 2:

Implementing for loop with continue statement

```
for a in 1 2 3 4 5 6 7 8 9 10
```

```
do
```

```
    # if a = 5 then continue the loop and
```

```
    # don't move to line 8
```

```
    if [ $a == 5 ]
```

```
    then
```

```
        continue
```

```
    fi
```

```
    echo "Iteration no $a"
```

```
done
```

Output

```
$bash -f main.sh
Iteration no 1
Iteration no 2
Iteration no 3
Iteration no 4
Iteration no 6
Iteration no 7
Iteration no 8
Iteration no 9
Iteration no 10
```

Example 3:

Implementing while loop

```
a=0
```

```
# -lt is less than operator
```

```
#Iterate the loop until a less than 10
```

```
while [ $a -lt 10 ]
```

```
do
```

```
    # Print the values
```

```
    echo $a
```

```
    # increment the value
```

```
    a=`expr $a + 1`
```

```
done
```

UNIX PROGRAMMING UNIT – 3

Output:

```
$bash -f main.sh
0
1
2
3
4
5
6
7
8
9
```

Example 4:

Implementing until loop

```
a=0
```

```
# -gt is greater than operator
```

```
#Iterate the loop until a is greater than 10
```

```
until [ $a -gt 10 ]
```

```
do
```

```
    # Print the values
```

```
    echo $a
```

```
    # increment the value
```

```
    a=`expr $a + 1`
```

```
done
```

Output:

```
$bash -f main.sh
0
1
2
3
4
5
6
7
8
9
10
```

FILTERS

When a program takes its input from another program, it performs some operation on that input, and writes the result to the standard output. It is referred to as a *filter*. Filters are the methods to find what we required as output.

THE GREP FAMILY

The grep family consists of three commands

- a) The grep command
- b) The egrep command
- c) The fgrep command

These are called as Searching Filters Family.

The grep Command

The grep command searches a file or files for lines that have a certain pattern. The syntax is –
\$grep pattern file(s)

The name "**grep**" comes from the ed (a Unix line editor) command **g/re/p** which means “globally search for a regular expression and print all lines containing it”.

A regular expression is either some plain text (a word, for example) and/or special characters used for pattern matching.

The simplest use of grep is to look for a pattern consisting of a single word. It can be used in a pipe so that only those lines of the input files containing a given string are sent to the standard output. If you don't give grep a filename to read, it reads its standard input; that's the way all filter programs work –

```
$ls -l | grep "Aug"  
-rw-rw-rw- 1 john doc 11008 Aug 6 14:10 ch02  
-rw-rw-rw- 1 john doc 8515 Aug 6 15:30 ch07  
-rw-rw-r-- 1 john doc 2488 Aug 15 10:51 intro  
-rw-rw-r-- 1 carol doc 1605 Aug 23 07:35 macros  
$
```

There are various options which you can use along with the **grep** command –

Sr.No.	Option & Description
1	-v (Prints all lines that do not match pattern.)
2	-n (Prints the matched line and its line number.)
3	-l (Prints only the names of files with matching lines (letter "l"))

UNIX PROGRAMMING UNIT – 4

4	-c (Prints only the count of matching lines.)
5	-I (Matches either upper or lowercase.)

Egrep(Extended grep):

Extended grep (egrep) is the most powerful of the three grep utilities. While it doesn't have the save option, it does allow more complex patterns. Consider the case where we want to extract all lines that start with a capital letter and end in an exclamation point (!). Our first attempt at this command is shown as follows:

```
$ egrep -n '^ [A-Z].*!$' filename
```

The first expression starts at the beginning of the line (^) and looks at the first character only. It uses a set that consists of only uppercase letters ([A-Z]). If the first character does not match the set, the line is skipped and the next line is examined.

If the first character is a match, the second expression (.*) matches the rest of the line until the last character, which must be an exclamation mark; the third expression examines the character at the end of the line (\$). It must be an exclamation point (a bang). The complete expression therefore matches any line starting with an uppercase letter, that is followed by zero or more characters, and that ends in a bang.

Finally note that we have coded the entire expression in a set of single quotes even though this expression does not require it.

Fast grep:

If your search criteria require only sequence expressions, fast grep (**fgrep**) is the best utility. Because its expressions consist of only sequence operators, it is also easiest to use if you are searching for text characters that are the same as regular expression operators such as the escape, parentheses, or quotes. For example, to extract all lines of the file that contain an apostrophe, we could use **fgrep** as

```
$ fgrep -n " ' " file name
```

OTHER FILTERS

The different types of filters are

- **comm**
- **diff**
- **head**
- **tail**
- **nl**
- **cut**
- **paste**
- **sort**

UNIX PROGRAMMING UNIT – 4

- **tr**
- **tee**

The comm Command:

The comm. command compare two sorted files line by line and write to standard output; the lines that are common and the lines that is unique.

Suppose you have two lists of people and you are asked to find out the names available in one and not in the other or even those common to both. **comm** is the command that will help you to achieve this. It requires two sorted files which it compares line by line. Before discussing anything further first let's check out the syntax of **comm** command:
Syntax :

\$comm [OPTION]... FILE1 FILE2

- As using comm, we are trying to compare two files therefore the syntax of comm command needs two filenames as arguments.
- With no OPTION used, comm produces three-column output where first column contains lines unique to FILE1 , second column contains lines unique to FILE2 and third and last column contains lines common to both the files.
- comm command only works right if you are comparing two files which are **already sorted**.

Example: Let us suppose there are two sorted files file1.txt and file2.txt and now we will use comm command to compare these two.

```
// displaying contents of file1 //
```

```
$cat file1.txt
```

```
Apaar  
Ayush Rajput  
Deepak  
Hemant
```

```
// displaying contents of file2 //
```

```
$cat file2.txt
```

```
Apaar  
Hemant  
Lucky  
Pranjal Thakral
```

Now, run comm command as:

```
// using comm command for  
comparing two files //
```

```
$comm file1.txt file2.txt
```

```
Apaar  
Ayush Rajput  
Deepak  
Hemant  
Lucky  
Pranjal Thakral
```

UNIX PROGRAMMING UNIT – 4

The above output contains of three columns where **first column** is separated by zero tab and contains names only present in file1.txt , **second column** contains names only present in file2.txt and separated by one tab and the **third column** contains names common to both the files and is separated by two tabs from the beginning of the line. This is the default pattern of the output produced by comm command when no option is used .

The diff command:

The diff stands for **difference**. This command is used to display the differences in the files by comparing the files line by line. Unlike its fellow members, cmp and comm, it tells us which lines in one file have is to be changed to make the two files identical.

The important thing to remember is that **diff** uses certain **special symbols** and **instructions** that are required to make two files identical. It tells you the instructions on how to change the first file to make it match the second file.

Special symbols are:

a : add

c : change

d : delete

Syntax :

diff [options] File1 File2

Lets say we have two files with names **a.txt** and **b.txt** containing 5 Indian states.

```
$ ls
```

```
a.txt b.txt
```

```
$ cat a.txt
```

```
Gujarat
```

```
Uttar Pradesh
```

```
Kolkata
```

```
Bihar
```

```
Jammu and Kashmir
```

```
$ cat b.txt
```

```
Tamil Nadu
```

```
Gujarat
```

```
Andhra Pradesh
```

```
Bihar
```

```
Uttar pradesh
```

Now, applying **diff** command without any option we get the following output:

```
$ diff a.txt b.txt
```

```
0a1
```

```
> Tamil Nadu
```

```
2,3c3
```

UNIX PROGRAMMING UNIT – 4

```
< Uttar Pradesh
Andhra Pradesh
5c5
Uttar pradesh
```

Let's take a look at what this output means. The first line of the **diff** output will contain:

- Line numbers corresponding to the first file,
- A special symbol and
- Line numbers corresponding to the second file.
- Like in our case, **0a1** which means **after** lines 0(at the very beginning of file) you have to add **Tamil Nadu** to match the second file line number 1. It then tells us what those lines are in each file preceded by the symbol:
- Lines preceded by a < are lines from the first file.
- Lines preceded by > are lines from the second file.
- Next line contains **2,3c3** which means from line 2 to line 3 in the first file needs to be changed to match line number 3 in the second file. It then tells us those lines with the above symbols.
- The three dashes (“—”) merely separate the lines of file 1 and file 2.

As a summary to make both the files identical, first add *Tamil Nadu* in the first file at very beginning to match line 1 of second file after that change line 2 and 3 of first file i.e. *Uttar Pradesh* and *Kolkata* with line 3 of second file i.e. *Andhra Pradesh*. After that change line 5 of first file i.e. *Jammu and Kashmir* with line 5 of second file i.e. *Uttar pradesh*.

The head command:

The head command, as the name implies, print the top N number of data of the given input. By default, it prints the first 10 lines of the specified files. If more than one file name is provided then data from each file is preceded by its file name.

Syntax:

```
head [OPTION]... [FILE]...
```

Let us consider two files having name **state.txt** and **capital.txt** contains all the names of the Indian states and capitals respectively.

```
$ cat state.txt
Andhra Pradesh
Arunachal Pradesh
Assam
Bihar
Chhattisgarh
Goa
Gujarat
Haryana
Himachal Pradesh
Jammu and Kashmir
Jharkhand
Karnataka
Kerala
Madhya Pradesh
```

UNIX PROGRAMMING UNIT – 4

Maharashtra
Manipur
Meghalaya
Mizoram
Nagaland
Odisha
Punjab
Rajasthan
Sikkim
Tamil Nadu
Telangana
Tripura
Uttar Pradesh
Uttarakhand
West Bengal

Without any option, it displays only the first 10 lines of the file specified.

Example:

\$ head state.txt

Andhra Pradesh
Arunachal Pradesh
Assam
Bihar
Chhattisgarh
Goa
Gujarat
Haryana
Himachal Pradesh
Jammu and Kashmir

Options

Short options	Long options
-n	-Line
-c	-Bytes
-q	-Quiet
-v	-Verbose

The tail command:

It is the complementary of head command. The tail command, as the name implies, print the last N number of data of the given input. By default it prints the last 10 lines of the specified files. If more than one file name is provided then data from each file is precedes by its file name.

Syntax:

tail [OPTION]... [FILE]...

UNIX PROGRAMMING UNIT – 4

Let us consider two files having name **state.txt** and **capital.txt** contains all the names of the Indian states and capitals respectively.

```
$ cat state.txt
Andhra Pradesh
Arunachal Pradesh
Assam
Bihar
Chhattisgarh
Goa
Gujarat
Haryana
Himachal Pradesh
Jammu and Kashmir
Jharkhand
Karnataka
Kerala
Madhya Pradesh
Maharashtra
Manipur
Meghalaya
Mizoram
Nagaland
Odisha
Punjab
Rajasthan
Sikkim
Tamil Nadu
Telangana
Tripura
Uttar Pradesh
Uttarakhand
West Bengal
```

Without any option it display only the last 10 lines of the file specified.

Example:

```
$ tail state.txt
Odisha
Punjab
Rajasthan
Sikkim
Tamil Nadu
Telangana
Tripura
Uttar Pradesh
Uttarakhand
West Bengal
```

UNIX PROGRAMMING UNIT – 4

Options:

Short options	Long options
-n	-Line
-c	-Bytes
-q	-Quiet
-v	-Verbose
-f	-Follow

The nl command:

- It is a Numbered Filter.
- It is used to give number to the lines of files

Syntax:

nl [options] [lines]

Example:

```
$cat fruits
```

```
Apple
```

```
Banana
```

```
Orange
```

```
$
```

```
$nl fruits
```

```
1 Apple
```

```
2 Banana
```

```
3 Orange
```

The cut command:

- Using this command, required fields or columns can be extracted from a file
- It is a Split Filter.

Syntax:

cut[options] [files] Example:

```
$cat emp
```

```
EmpID EmpName Sal
```

```
111 aaa 10000
```

```
222 bbb 12340
```

```
333 ccc 15400
```

```
$cut -f3 emp
```

```
Sal
```

```
10000
```

```
12340
```

```
15400
```

UNIX PROGRAMMING UNIT – 4

The paste command:

This command is used to merge lines of files. It is a Merging filter. This command is used to create new tables or files by using together fields or columns from two or more files.

Syntax:

```
paste [options] [files]
```

Example:

```
$cat file1
Hi Hello
$cat file2
Welcome
$paste file1 file2
Hi Hello
Welcome
```

The sort command:

The sort command is used to sort lines of text files. This command is one of the powerful and a general purpose tool that is used for sorting information stored in file. It is sorting filter. In addition in sorting, this command can be used for merging sorted files.

Syntax:

```
sort [options] [files] Example:
```

```
$cat Names
RONITH
PRANEETH
KARTHEEK
SAI
$
$sort Names
KARTHEEK
PRANEETH
RONITH
SAI
```

The tr command:

It is used to translate or delete characters. It is Translation filter. Translation includes both substitution as well as deletion of characters or strings.

Syntax:

```
tr [options] [sets]
```

```
$ cat Names1
aaab
bbba
$
```

UNIX PROGRAMMING UNIT – 4

```
$ tr 'a' 'A' < Names1
AAAB
bbbA
$
```

The tee command:

It is a bidirectional filter. It is used to read from standard input and write to standard output and files. Bidirectional means one copy is displayed on the output screen and other copy is on a separate file.

Syntax:

```
Tee [options] [files]
```

Example:

```
$ cat file1
aaa
bbb
ccc
$
$cat file1 |wc-1 |file2
3
$
$cat file2
3
$
```

THE STREAM EDITOR SED

SED is a stream editor for filtering and transforming text. Sed is one of the most powerful filters. It stands for stream editor. Despite its name editing, it does not modify the original file. It reads the standard input (a keyboard or a file), processes it using a separate called sed script and writes the result to the standard output. A Sed script is a file containing a list of instructions to be applied to each line in the input file. If the script contains only one line it can be provided as in -line at the command line.

Options in sed command;

There are three useful options available with Sed utility.

OPTION	FUNCTION
-c	It is a default option. It indicates that the script is on the command line
-f	It indicates that the script is in a file which immediately follows this option
-n	It suppresses the automatic output. That is, it will not display the contents of the pattern space

UNIX PROGRAMMING UNIT – 4

Operation of sed:

Sed gives a line number to each line in the input file to address lines in the text. It does the following operations for each input line. It copies an input line to a buffer called the pattern space. A pattern space can hold one or more lines at the same time. To each line in the pattern space that matches the specified addresses in the `Sed` instruction; it applies all the instructions in the sed script one by one.

- If `—n` option is not specified on the command line, then it copies the pattern space to the standard output (a monitor or a file).
- It repeats this process for each input line starting with `step1`.
- The sed utility also uses a temporary buffer called hold space to hold one or more lines as directed by the sed instructions.

Example:

```
$cat names2
1 PYTHON
2 JAVA
3 C
4 JS
5 DOTNET
$
$ sed 2q names2
1 PYTHON
2 JAVA
$
$ sed -n 4p name2
4 JS
$
```

THE AWK PATTERN SCANNING AND PROCESSING LANGUAGE

The awk utility which takes its name from the initial of its authors (Alfred V. Aho, Peter J. Weinberger and Brian W. Kernighan), is a powerful programming language disguised as a utility. Its behavior is to some extent like sed. It reads the input file, line by line, and performs an action on a part of or on the entire line. Unlike sed, however, it does not print the line unless specifically told to print it.

Features of awk:

- **Field oriented file processing**
- **Regular Expressions**
- **Predefined Variables**
- **Numeric Operations**
- **Comparison Operators**
- **Arrays**
- **Control Statements**
- **Report Generation**

UNIX PROGRAMMING UNIT – 4

Syntax of an awk program statement:

`$awk [options] 'program' filelist`

Where the Use of options is optional

filelist will have zero or more input filenames

Program will have one or more statements having the following syntax

Syntax:

`pattern {action}`

The pattern component of a program statement indicates the basis for a line or record selection and manipulation.

Fields and Records in awk:

- A file is viewed as a collection of Fields and records by the awk utility.
- A field is a data unit that gives some data.
- Each line in a file is a record.
- That is a record is a collection of several fields.
- However, the record contains related data.
- A file that is organized into records is called a data file.

Example:

Consider the output of `ls -l` (long list) command in Unix.

It lists the information about all files in a directory.

```
$ls -l
```

```
- r w - r - - r - -   own I  file1
- r w - r - - r - -   own I  File2
- r w - r - - r - -   own I  File3
```

Here, it contains three fields and three records

The fields are file permissions, owner and filenames

The records give information about file1, file2, file3

OUTPUT STATEMENTS:

In awk, the output can be displayed using three statements,

print statement:

```
$awk {print}, info.dat
```

printf statement:

```
$awk {printf("%S%4d%5.2f\n", $1,$2,$3)}, info.dat
```

sprint statement:

```
value=sprint("%S%4d%5.2f\n", $1,$2,$3)
```

UNIX PROGRAMMING UNIT – 4

Different patterns available in awk:

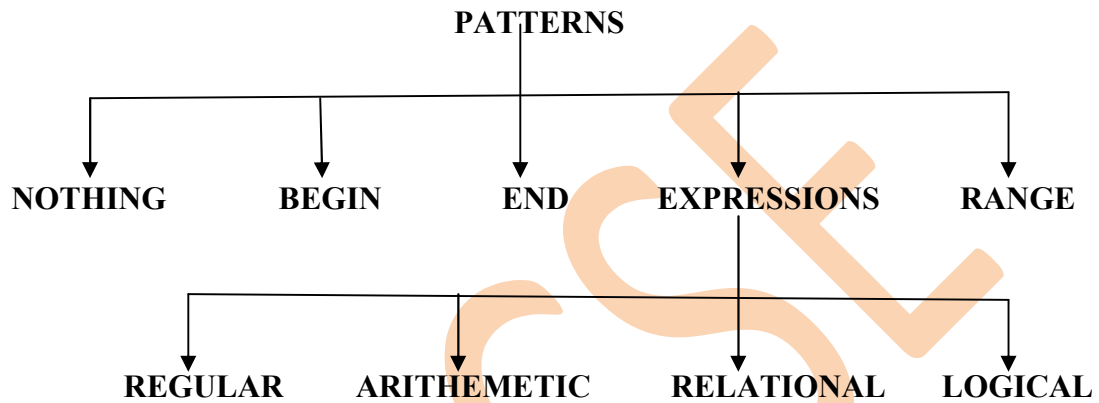
The pattern identifies on which records in a file, the action is to be taken.

If the pattern matches the record, its associated action is taken.

If the pattern does not match the record, its associated action is skipped.

If an action is specified without a pattern then it is always taken.

The awk supports different types of patterns



VARIABLES IN AWK:

Like all programming languages, awk also permits the use of variables.

In awk, two types of variables are available

- User defined variables
- Built-in variables

The Built-in variables of awk are,

<i>VARIABLE</i>	<i>MEANING</i>
FILENAME	Name of the current input file
IS	Input field Separator(default : blank and tab)
NF	Number of fields in input record
NC	Number of current record
OFS	Output field Separator
ORS	Output record separator
RS	Input record separator
ARGC	Number of command line arguments
ARGV	Command line arguments array

UNIX PROGRAMMING UNIT – 4

OPERATORS IN AWK:

<i>TYPE OF THE OPERATOR</i>	<i>SYMBOLS</i>
ARITHMETIC	+ - * / %
RELATIONAL	< <= > >= == !=
LOGICAL	&& !
ASSIGNMENT	= += -= *= /= %=
INCREMENT	++
DECREMENT	--
CONDITIONAL	? :
MATCH	~ !~

Expressions in awk;

- Arithmetic Expressions
- Relational Expressions
- Logical Expressions
- Regular Expressions

Awk decision making control structures:

- if-else structure
- nested if structure
- case structure

Awk loop statements:

- while loop
- do while loop
- for loop

Awk unconditional control statements:

- break
- continue

Associative arrays in Awk:

AWK has associative arrays and one of the best thing about it is – the indexes need not to be continuous set of number; you can use either string or number as an array index. Also, there is no need to declare the size of an array in advance – arrays can expand/shrink at runtime.

Its syntax is as follows –

Syntax

array_name[index] = value

UNIX PROGRAMMING UNIT – 4

Where **array_name** is the name of array, **index** is the array index, and **value** is any value assigning to the element of the array.

Example:

```
Employee[$2]
```

Initialization of array is given as

```
Employee[$2]=$4
```

Delete an array element is given as
delete Employee[\$2]

STRING FUNCTION IN AWK:

The string functions in awk are

- length()
- index()
- substr()
- sub()
- split()

length():

The length function counts the number of characters in a string and returns the count value.

Syntax:

```
length(str)
```

index():

The index function determines the first position of a sub string with in a string. If the string not found it returns 0.

Syntax:

```
index(str, substr)
```

substr():

The substr() returns the substring from a given string.

Syntax:

```
substr(str, starting_index)
```

sub():

The sub function substitutes one string for another string that matches a regular expression. It returns 1(true) if the string was substituted. It returns 0(false) if the matching string was not found and the substitution failed.

Syntax:

```
Sub(regular_expr, withstring, inputstring)
```

UNIX PROGRAMMING UNIT – 4

Split():

The split function divides a string in to two substrings using a field separator.

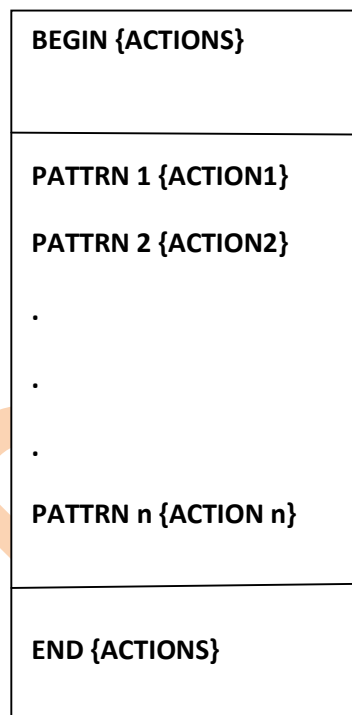
syntax:

```
split(str, array)
```

AWK SCRIPT:

An awk script is divided into three parts. They are,

- Initialization
- Body
- End of job



Initialization:

The initialization part defines the instructions for initializing variables, creating report headings, set system variables etc. This is identified with the token BEGIN and all instructions are enclosed with curly braces. This part is processed only once before awk reads the first line from the input line.

UNIX PROGRAMMING UNIT – 4

Body:

The body consists of one or more instructions for processing the data in a file. Each instruction consists of a pattern and an action that will be taken when the pattern is matched. The awk applies each instruction in the body of each record in the file one after another.

End of a job:

The end part is also executed only once after the last line from the input file has been read. It includes the instructions to analyze, print or perform other activities at the end of data processing.

GOOD FILES AND GOOD FILTERS

- The *awk* programs of one or two lines length can perform filtering contributing into a larger pipeline. This is one of the applications of *awk*.
- A single filter can solve certain problems sometimes, but the filters which are joined together as pipeline can solve the problem divided into sub-problems.
- Such usage of tool is assumed as heart of UNIX programming.
- The files that are filterable contain decorative headers, trailers or blank lines. Every line is an object of interest such as filename, word, running process description.
- Therefore the *wc* and *grep* programs can search the items by name and count them.
- Even though the file contains more data about the objects, it will be in the *line_by_line* and *column* format. Such files can be easily processed and rearranged by *awk*.

Shell Programming

A **Shell** provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output. Shell is an environment in which we can run our commands, programs, and shell scripts.

Shell Variables:

A shell variable is simply set by equating it to a value. A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.

Variable Names:

The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).

Defining Variables: Variables are defined as follows

Syntax:

```
variable_name=variable_value
```

Example:

```
NAME="CRR CSE"
```

Accessing Values: To access the value stored in a variable, prefix its name with the dollar sign (\$) – For example, the following script will access the value of defined variable NAME and print it on STDOUT.

Example:

```
echo $NAME  
CRR CSE
```

Read-only Variables:

Shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed. For example, the following script generates an error while trying to change the value of NAME.

Example:

```
NAME="CRR CSE"  
readonly NAME  
NAME="CRR IT"
```

The above script will generate the following result –

```
/bin/sh: NAME: This variable is read only.
```


UNIX PROGRAMMING UNIT – 5

Unsetting Variables:

Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you cannot access the stored value in the variable. Following is the syntax to unset a defined variable using the **unset** command.

Syntax:

```
unset variable_name
```

The above command unsets the value of a defined variable. Here is a simple example that demonstrates how the command works.

Example:

```
NAME="CRR CSE"  
unset NAME  
echo $NAME
```

The above example does not print anything.

Variable Types:

When a shell is running, three main types of variables are present.

Local Variables – A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at the command prompt.

Environment Variables – An environment variable is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually, a shell script defines only those environment variables that are needed by the programs that it runs.

Shell Variables – A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

The Export Command:

The export command is fairly simple to use as it has straightforward syntax with only three available command options. In general, the export command marks an environment variable to be exported with any newly forked child processes and thus it allows a child process to inherit all marked variables.

Options:

Tag	Description
-p	List of all names that are exported in the current shell
-n	Remove names from export list
-f	Names are exported as functions

UNIX PROGRAMMING UNIT – 5

Example-1:

To view all the exported variables.

```
$ export
```

Example-2:

user can also use -p option to view all exported variables on current shell.

```
$ export -p
```

The Profile File a Script Run During Starting:

The file **/etc/profile** is maintained by the system administrator of your Unix machine and contains shell initialization information required by all users on a system. The file **.profile** is under your control. You can add as much shell customization information as you want to this file. The minimum set of information that you need to configure includes -

- The type of terminal you are using.
- A list of directories in which to locate the commands.
- A list of variables affecting the look and feel of your terminal.

You can check your **.profile** available in your home directory. Open it using the vi editor and check all the variables set for your environment.

Example:

```
$ cat .profile
```

The First Shell Script:

A Shell script or shell program is a set of commands that are executed together as a single unit. A shell script also includes commands for selective execution, commands for I/O operations, commands for repeated execution and shell variables.

Steps to write and execute a script

- Open the terminal. Go to the directory where you want to create your script.
- Create a file with **.sh** extension.
- Write the script in the file using an editor.
- Make the script executable with command **chmod +x <fileName>**.
- Run the script using **./<fileName>**.

First Script:

Here we'll write a simple program for Hello World.

First of all, create a simple script in any editor

UNIX PROGRAMMING UNIT – 5

The set command (Assigning values to positional parameters):

Positional parameters assigned using the set command

Example:

```
guest-EwJMaM@cselab2-OptiPlex-7020: ~
guest-EwJMaM@cselab2-OptiPlex-7020::~$ set SIR CR REDDY COLLEGE OF ENGINEERING
guest-EwJMaM@cselab2-OptiPlex-7020::~$ echo "$0"
/bin/bash
guest-EwJMaM@cselab2-OptiPlex-7020::~$ echo "$1"
SIR
guest-EwJMaM@cselab2-OptiPlex-7020::~$ echo "$2"
CR
guest-EwJMaM@cselab2-OptiPlex-7020::~$ echo "$3"
REDDY
guest-EwJMaM@cselab2-OptiPlex-7020::~$ echo "$4"
COLLEGE
guest-EwJMaM@cselab2-OptiPlex-7020::~$ echo "$5"
OF
guest-EwJMaM@cselab2-OptiPlex-7020::~$ echo "$6"
ENGINEERING
guest-EwJMaM@cselab2-OptiPlex-7020::~$ echo "$#"
6
```

The shift command(Handling Excess Command Line Arguments):

The shift command in UNIX is used to move the command line arguments to one position left. The first argument is lost when you use the shift command. Shifting command line arguments is useful when you perform a similar action to all arguments one-by-one, without changing the variable name. The shift command throws away the left-most variable (argument number 1) and reassigns values to the remaining variables. This command is used to shift the position of the positional parameters.

Example:

```
guest-EwJMaM@cselab2-OptiPlex-7020: ~
guest-EwJMaM@cselab2-OptiPlex-7020::~$ set SIR CR REDDY COLLEGE OF ENGINEERING
guest-EwJMaM@cselab2-OptiPlex-7020::~$ shift 3
guest-EwJMaM@cselab2-OptiPlex-7020::~$ echo $1 $2 $3
COLLEGE OF ENGINEERING
```

The \$? Variable knowing the exit Status:

When ever a command is successfully executed the program returned zero(0), otherwise it returns a non zero value.

UNIX PROGRAMMING UNIT – 5

Example:

```
guest-EwJMaM@cselab2-OptiPlex-7020: ~
guest-EwJMaM@cselab2-OptiPlex-7020:~$ ls
Desktop      examples.desktop  input2.sh  positional.sh  Templates
Documents    first.sh          Music      positional.sh  Videos
Downloads    input1.sh         Pictures   Public
guest-EwJMaM@cselab2-OptiPlex-7020:~$ cat input1.sh
echo "Enter a number"
read num
echo "number is $num"
echo "Enter a name"
read name
echo "name is $name"
guest-EwJMaM@cselab2-OptiPlex-7020:~$ echo $?
0
guest-EwJMaM@cselab2-OptiPlex-7020:~$ cat input
cat: input: No such file or directory
guest-EwJMaM@cselab2-OptiPlex-7020:~$ echo $?
1
```

In the above program the input1.sh file exists so the program returns 0, in next example the input file is not existed so it returns 1.

More about the Set Command:

The set command without arguments:

When this command used without arguments display the contents, the system variables that are either local or exported.

Example:

```
$ set
CDPATH=:/users/gan:/usr/spool
EDITOR=/bin/vi
HOME=/user/gan
.....
$
```

The set command with options:

Many options such as `-x`, `-v`, `--` and others are allowed to be used with this command. The options `-x` and `-v` are used to debug shell scripts.

UNIX PROGRAMMING UNIT – 5

The set command and the – (double hyphen) option:

Under certain circumstances arguments to the set command are passed on through command substitution. Sometimes this method is error-prone. Such a situation is handled by using the special option –(double hyphen)

The Exit Command:

exit command in linux is used to exit the shell where it is currently running. It takes one more parameter as *[N]* and exits the shell with a return of status *N*. If *n* is not provided, then it simply returns the status of last command that is executed.

Syntax:

`exit [n]`

Options for exit command –

exit: Exit Without Parameter

```
File Edit View Search Terminal Help
naman@root:~$ exit
```

After pressing enter, the terminal will simply close.

exit [n] : Exit With Parameter

```
File Edit View Search Terminal Help
naman@root:~$ exit 110
```

After pressing enter, the terminal window will close and return a status of 110. Return status is important because sometimes they can be mapped to tell error, warnings and notifications.

For example generally, return status –

“0” means the program has executed successfully.

“1” means the program has minor errors.

Operators in Unix:

There are various operators supported by each shell. We will discuss in detail about Bourne shell (default shell) in this chapter.

We will now discuss the following operators –

- Arithmetic Operators
- Relational Operators
- Boolean Operators
- String Operators
- File Test Operators

Arithmetic Operators : These operators are used to perform normal arithmetic or mathematical operations. There are 7 arithmetic operators:

Addition (+): Binary operation used to add two operands.

Subtraction (-): Binary operation used to subtract two operands.

Multiplication (*): Binary operation used to multiply two operands.

Division (/): Binary operation used to divide two operands.

Modulus (%): Binary operation used to find remainder of two operands.

Increment Operator (++) : Uniary operator used to increase the value of operand by one.

Decrement Operator (--) : Uniary operator used to decrease the value of a operand by one.

Example:

```
cselab2@cselab2-OptiPlex-7020: ~
echo "Enter first number:"
read a
echo "Enter second number:"
read b
add=$((a+b))
echo "Addition= $add"
sub=$((a-b))
echo "Subtraction= $sub"
mul=$((a*b))
echo "Multiplication= $mul"
div=$((a/b))
echo "Division= $div"
mod=$((a%b))
echo "Reminder= $mod"
~
~
~
~
~
~
~
~
~
~
"arithmetic.sh" 14 lines, 253 characters
```

UNIX PROGRAMMING UNIT – 5

```
cselab2@cselab2-OptiPlex-7020: ~
cselab2@cselab2-OptiPlex-7020:~$ sh arithmetic.sh
Enter first number:
20
Enter second number:
10
Addition= 30
Subtraction= 10
Multiplication= 200
Division= 2
Reminder= 0
```

Relational Operators : Relational operators are those operators which defines the relation between two operands. They give either true or false depending upon the relation. They are of 6 types.

'-eq' Operator : Double equal to operator compares the two operands. Its returns true is they are equal otherwise returns false.

'-neq' Operator : Not Equal to operator return true if the two operands are not equal otherwise it returns false.

'-lt' Operator : Less than operator returns true if first operand is lees than second operand otherwise returns false.

'-le' Operator : Less than or equal to operator returns true if first operand is less than or equal to second operand otherwise returns false

'-gt' Operator : Greater than operator return true if the first operand is greater than the second operand otherwise return false.

'-ge' Operator : Greater than or equal to operator returns true if first operand is greater than or equal to second operand otherwise returns false

Logical Operators : They are also known as boolean operators. These are used to perform logical operations. They are of 3 types:

Logical AND (&&) : This is a binary operator, which returns true if both the operands are true otherwise returns false.

Logical OR (||) : This is a binary operator, which returns true is either of the operand is true or both the operands are true and returns false if none of then is false.

Not Equal to (!) : This is a unary operator which returns true if the operand is false and returns false if the operand is true.

Bitwise Operators : A bitwise operator is an operator used to perform bitwise operations on bit patterns. They are of 6 types:

Bitwise And (&) : Bitwise & operator performs binary AND operation bit by bit on the operands.

Bitwise OR (|) : Bitwise | operator performs binary OR operation bit by bit on the operands.

UNIX PROGRAMMING UNIT – 5

Bitwise XOR (^) : Bitwise ^ operator performs binary XOR operation bit by bit on the operands.

Bitwise compliment (~) : Bitwise ~ operator performs binary NOT operation bit by bit on the operand.

Left Shift (<<) : This operator shifts the bits of the left operand to left by number of times specified by right operand.

Right Shift (>>) : This operator shifts the bits of the left operand to right by number of times specified by right operand.

File Test Operator : These operators are used to test a particular property of a file.

-b operator : This operator check weather a file is a block special file or not. It returns true, if the file is a block special file otherwise false.

-c operator : This operator checks weather a file is a character special file or not. It returns true if it is a character special file otherwise false.

-d operator : This operator checks if the given directory exists or not. If it exists then operators returns true otherwise false.

-e operator : This operator checks weather the given file exists or not. If it exists this operator returns true otherwise false.

-r operator : This operator checks weather the given file has read access or not. If it has read access then it returns true otherwise false.

-w operator : This operator check weather the given file has write access or not. If it has write then it returns true otherwise false.

-x operator : This operator check weather the given file has execute access or not. If it has execute access then it returns true otherwise false.

-s operator : This operator checks the size of the given file. If the size of given file is greater than 0 then it returns true otherwise it false.

Branching Control Structures :

While writing a shell script, there may be a situation when you need to adopt one path out of the given two paths. So you need to make use of conditional statements that allow your program to make correct decisions and perform the right actions.

Unix Shell supports conditional statements which are used to perform different actions based on different conditions. We will now understand two decision-making statements here

–

- The **if...else** statement
- The **case...esac** statement

If statement:

This block will process if specified condition is true.

Syntax:

```
if [ expression ]
then
    Statement
fi
```


UNIX PROGRAMMING UNIT – 5

Example:

```
cselab2@cselab2-OptiPlex-7020: ~
day=5
case "$day" in
  1) echo "sunday";;
  2) echo "Monday";;
  3) echo "Tuesday";;
  4) echo "Wednesday";;
  5) echo "Thursday";;
  6) echo "Friday";;
  7) echo "Saturday";;
  *) echo "Invalid day number";;
esac

"case.sh" 12 lines, 206 characters
```

```
cselab2@cselab2-OptiPlex-7020: ~
cselab2@cselab2-OptiPlex-7020:~$ sh case.sh
Thursday
cselab2@cselab2-OptiPlex-7020:~$
```

Loop Control Structures:

Looping Statements in Shell Scripting: There are total 3 looping statements which can be used in bash programming

1. while statement
2. for statement
3. until statement

while statement

Here command is evaluated and based on the result loop will be executed, if command raises to false then loop will be terminated

Syntax:

```
while command
do
    statement to be executed
done
```


UNIX PROGRAMMING UNIT – 5

Example:

```
a=0
while [ $a -lt 10 ]
do
    echo $a
    a=`expr $a + 1`
done
```

```
0
1
2
3
4
5
6
7
8
9
```

for statement

The for loop operate on lists of items. It repeats a set of commands for every item in a list. Here var is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

Syntax:

```
for var in word1 word2 .... Wordn
do
    Statement to be executed
done
```

Example:

```
for a in 1 2 3 4 5 6 7 8 9 10
do
    echo "a value is $a"
done
```

UNIX PROGRAMMING UNIT – 5

```
a value is 1
a value is 2
a value is 3
a value is 4
a value is 5
a value is 6
a value is 7
a value is 8
a value is 9
a value is 10
```

until statement

The until loop is executed as many as times the condition/command evaluates to false. The loop terminates when the condition/command becomes true.

Syntax

```
until command
do
    Statement to be executed until command is true
Done
```

Example:

```
a=0
until [ $a -gt 10 ]
do
    echo $a
    a=`expr $a + 1`
done
```

```
0
1
2
3
4
5
6
7
8
9
10
```

The Continue and Break Statement

Continue Statement:

continue is a command which is used to skip the current iteration in for, while and until loop. It takes one more parameter $[N]$, if N is mentioned then it continues from the n th enclosing loop.

Syntax:
continue

Example:

```
for a in 1 2 3 4 5 6 7 8 9 10
do
    if [ $a == 7 ]
    then
        continue
    fi
    echo "a value is $a"
done
```

```
a value is 1
a value is 2
a value is 3
a value is 4
a value is 5
a value is 6
a value is 8
a value is 9
a value is 10
```

Break Statement:

break command is used to terminate the execution of for loop, while loop and until loop. It can also take one parameter i.e. $[N]$. Here n is the number of nested loops to break. The default number is 1.

Syntax:
break

Example:

```
for a in 1 2 3 4 5 6 7 8 9 10
do
    if [ $a == 7 ]
    then
        break
    fi
    echo "a value is $a"
done
```

UNIX PROGRAMMING UNIT – 5

```
a value is 1
a value is 2
a value is 3
a value is 4
a value is 5
a value is 6
```

The Expr Command:

The `expr` command in Unix evaluates a given expression and displays its corresponding output. It is used for:

- Basic operations like addition, subtraction, multiplication, division, and modulus on integers.
- Evaluating regular expressions, string operations like substring, length of strings etc.

Syntax:

`$ expr expression`

Example:

```
echo "Enter two numbers"
read num1
read num2
s=`expr $num1 + $num2`
echo "Sum = $s"
```

```
Enter two numbers
10
20
Sum = 30
```

Performing Integer Arithmetic-Real Arithmetic in Shell Programs.

Performing Integer Arithmetic:

The both operands in arithmetic operation are integers then that operation is called integer arithmetic.

Example:

```
echo "Enter two numbers"
read num1
read num2
s=`expr $num1 + $num2`
echo "Sum = $s"
```

```
Enter two numbers
10
20
Sum = 30
```

UNIX PROGRAMMING UNIT – 5

Performing Real Arithmetic:

The both operands in arithmetic operation are real then that operation is called real arithmetic, the expr command works only on integers. Real arithmetic can be managed bc (basic calculator) command along with the scale function and the echo command. The output of the arithmetic expression is piped to the bc command.

Example:

```
echo "Enter radius of circle"
read r
area=`echo "scale =3; 3.14*$r*$r" | bc`
echo "Area of circle=$area"
~
~
```

```
Enter radius of circle
2.35
Area of circle=17.340
```

The here Document(<<):

In unix it is possible to include the document on which the system has to operate along with the command itself. A here document is a way of getting text input into a script without having to feed it in from a separate file, this type of redirection tells the shell to read input from the current source (HERE) until a line containing only word (HERE) is seen.

Syntax:

```
Command << HERE
Text
Text
.....
```

Example 1:

```
~$ cat <<stop
> sir c r reddy college of engineering
> stop
sir c r reddy college of engineering
```

Example 2:

```
~$ wc -w << end
> sir c r reddy college of engineering
> end
7
```

The Sleep Command:

Using this command the user can make the system to sleep, that is, pause for some fixed period of time. The sleep command suspends the calling process for at least the specified number *f* seconds (by default), minutes, hours or days.

Syntax:

\$sleep number

Example :

```
echo "Welcome"
sleep 10
echo "Welcome"
sleep 10s
echo "Welcome"
sleep 1m
echo "Welcome"
~
~
~
```

```
guest-x1iN9Y@cselab2-OptiPlex-7020: ~
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ sh time.sh
Welcome
Welcome
Welcome
Welcome
```

Debugging Scripts:

When a script does not work properly, we need to determine the location of the problem. The UNIX shells provide a debugging mode. Run the entire script in debug mode or just a portion of the script. Debug statements options are,

Option	Meaning
set -x	Prints the statements after interpreting meta characters and variables
set +x	Stops the printing of statements
set -v	Prints the statements before interpreting meta characters and variables
set -f	Disables file name generation (using meta characters)

UNIX PROGRAMMING UNIT – 5

Example:

```
echo "Enter a Number:"
read n
for i in `seq 1 $n`
do
    echo $i
done
~
~
~
```

```
guest-x1iN9Y@cselab2-OptiPlex-7020: ~
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ sh debug1.sh
Enter a Number:
8
1
2
3
4
5
6
7
8
```

```
guest-x1iN9Y@cselab2-OptiPlex-7020: ~
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ sh -xv debug1.sh
echo "Enter a Number:"
+ echo Enter a Number:
Enter a Number:
read n
+ read n
8
for i in `seq 1 $n`
do
    echo $i
done
+ seq 1 8
+ echo 1
1
+ echo 2
2
+ echo 3
3
+ echo 4
4
+ echo 5
5
+ echo 6
6
+ echo 7
7
+ echo 8
8
```

UNIX PROGRAMMING UNIT – 5

```
guest-x1iN9Y@cselab2-OptiPlex-7020: ~
set -xv
echo "Enter a Number:"
read n
for i in `seq 1 $n`
do
    echo $i
done
~
~
~
```

```
guest-x1iN9Y@cselab2-OptiPlex-7020: ~
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ sh debug1.sh
echo "Enter a Number:"
+ echo Enter a Number:
Enter a Number:
read n
+ read n
8
for i in `seq 1 $n`
do
    echo $i
done
+ seq 1 8
+ echo 1
1
+ echo 2
2
+ echo 3
3
+ echo 4
4
+ echo 5
5
+ echo 6
6
+ echo 7
7
+ echo 8
8
```

The Script Command:

script command in Linux is used to make typescript or record all the terminal activities. After executing the *script* command it starts recording everything printed on the screen including the inputs and outputs until exit.

Syntax:

```
script [options] [file]
```


UNIX PROGRAMMING UNIT – 5

Example:

```
guest-x1iN9Y@cselab2-OptiPlex-7020: ~
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ script crrcse.txt
Script started, file is crrcse.txt
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ echo "SIRCRRCOE"
SIRCRRCOE
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ date
Wed Jun 17 11:29:15 IST 2020
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ time

real    0m0.000s
user    0m0.000s
sys     0m0.000s
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ cal

   June 2020
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

guest-x1iN9Y@cselab2-OptiPlex-7020:~$ exit
exit
Script done, file is crrcse.txt
```

```
guest-x1iN9Y@cselab2-OptiPlex-7020: ~
Script started on Wednesday 17 June 2020 11:28:57 AM IST
^[]0;guest-x1iN9Y@cselab2-OptiPlex-7020: ~^Cguest-x1iN9Y@cselab2-OptiPlex-7020:~$ echo "SIRCRRCOE"^M
SIRCRRCOE^M
^[]0;guest-x1iN9Y@cselab2-OptiPlex-7020: ~^Cguest-x1iN9Y@cselab2-OptiPlex-7020:~$ DATE^H^H[[K^H^H[[K^H^H[[K^H^H[[Kdate^M
Wed Jun 17 11:29:15 IST 2020^M
^[]0;guest-x1iN9Y@cselab2-OptiPlex-7020: ~^Cguest-x1iN9Y@cselab2-OptiPlex-7020:~$ time^M
^M
real    0m0.000s^M
user    0m0.000s^M
sys     0m0.000s^M
^[]0;guest-x1iN9Y@cselab2-OptiPlex-7020: ~^Cguest-x1iN9Y@cselab2-OptiPlex-7020:~$ cal^M
   June 2020 ^M
Su Mo Tu We Th Fr Sa ^M
      1  2  3  4  5  6 ^M
  7  8  9 10 11 12 13 ^M
14 15 16 ^[[7m17^[[27m 18 19 20 ^M
21 22 23 24 25 26 27 ^M
28 29 30 ^M
^M
^[]0;guest-x1iN9Y@cselab2-OptiPlex-7020: ~^Cguest-x1iN9Y@cselab2-OptiPlex-7020:~$ exit^M
exit^M
Script done on Wednesday 17 June 2020 11:29:28 AM IST
~
~
```

The Eval Command:

eval is a built-in Linux command which is used to execute arguments as a shell command. It combines arguments into a single string and uses it as an input to the shell and execute the commands.

Syntax:

Eval [arg ...]

UNIX PROGRAMMING UNIT – 5

Example:

```
guest-x1iN9Y@cselab2-OptiPlex-7020: ~
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ d="date"
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ c="cal"
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ eval $d
Wed Jun 17 11:36:53 IST 2020
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ eval $c
  June 2020
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
```

The Exec Command:

exec command in Linux is used to execute a command from the bash itself. This command does not create a new process it just replaces the bash with the command to be executed. If the exec command is successful, it does not return to the calling process.

Syntax:

```
exec [ -cl ] [ -a name ] [ command [arguments ] ] [ redirection ... ]
```

Example:

```
guest-x1iN9Y@cselab2-OptiPlex-7020: ~
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ bash
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ exec ls
bc1.sh      Desktop  examples.desktop  Music    sircrr.txt  tmp
crrcse.txt Documents in.txt           Pictures Templates typescript
debug1.sh  Downloads loop1.sh          Public   time.sh     Videos
guest-x1iN9Y@cselab2-OptiPlex-7020:~$
```

```
guest-x1iN9Y@cselab2-OptiPlex-7020: ~
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ bash
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ echo 'Sir C R Reddy College of Engineering' > in.txt
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ exec wc -w < in.txt
7
```

UNIX PROGRAMMING UNIT – 5

```
guest-x1iN9Y@cselab2-OptiPlex-7020: ~
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ bash
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ echo 'Sir C R Reddy College of Engineering' > in.txt
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ exec wc -w < in.txt
7
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ clear
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ bash
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ exec > tmp
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ ls
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ echo 'Sir C R Reddy College of Engineering'
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ exit
exit
guest-x1iN9Y@cselab2-OptiPlex-7020:~$ cat tmp
bc1.sh
crrcse.txt
debug1.sh
Desktop
Documents
Downloads
examples.desktop
in.txt
loop1.sh
Music
Pictures
Public
sircrr.txt
Templates
time.sh
tmp
typescript
Videos
Sir C R Reddy College of Engineering
```

UNIX PROGRAMMING UNIT – 6

The Process

The Meaning of a Process:

A **process** is a program in execution in memory or in other words, an instance of a program in memory. Any program executed creates a process. A program can be a command, a shell script, or any binary executable or any application. However, not all commands end up in creating process, there are some exceptions. Similar to how a file created has properties associated with it, a process also has lots of properties associated to it.

Process attributes:

A process has some properties associated to it:

PID : Process-Id. Every process created in Unix/Linux has an identification number associated to it which is called the process-id. This process id is used by the kernel to identify the process similar to how the inode number is used for file identification. The PID is unique for a process at any given point of time. However, it gets recycled.

PPID : Parent Process Id: Every process has to be created by some other process. The process which creates a process is the parent process, and the process being created is the child process. The PID of the parent process is called the parent process id(PPID).

TTY: Terminal to which the process is associated to. Every command is run from a terminal which is associated to the process. However, not all processes are associated to a terminal. There are some processes which do not belong to any terminal. These are called daemons.

UID: User Id- The user to whom the process belongs to. And the user who is the owner of the process can only kill the process(Of course, root user can kill any process). When a process tries to access files, the accessibility depends on the permissions the process owner has on those files.

List the processes:

```
~$ ps
  PID TTY          TIME CMD
  441 pts/0    00:00:00 bash
  519 pts/0    00:00:00 ps
~$
```

ps is the Unix / Linux command which lists the active processes and its status. By default, it lists the processes belonging to the current user being run from the current terminal.

The ps command output shows 4 things:

PID : The unique id of the process

TTY: The terminal from which the process or command is executed.

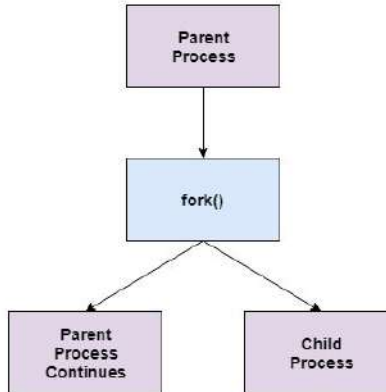
TIME: The amount of CPU time the process has taken

CMD: The command which is executed.

UNIX PROGRAMMING UNIT – 6

Parent and Child Processes:

Every process in Unix has to be created by some other process. Hence, the ps command is also created by some other process. The 'ps' command is being run from the login shell, bash. The bash shell is a process running in the memory right from the moment the user logged in. So, for all the commands triggered from the login shell, the login shell will be the parent process and the process created for the command executed will be the child process. In the same lines, the 'bash' is the parent process for the child process 'ps'.



The below command shows the process list along with the PPID.

```
~$ ps -o pid,ppid,args
  PID   PPID COMMAND
   441     9 /bin/bash
   574   441 ps -o pid,ppid,args
~$
```

The PID of the bash is same as the PPID of the ps command which means the bash process is the parent of the ps command. The '-o' option of the ps command allows the user to specify only the fields which he needs to display.

Types of Processes:

Processes in Unix are classified into 3 types

1. Interactive Process
2. Non – interactive Process
3. Daemon Process

Interactive Process:

These are also called as background processes. Certain processes can be made to run independent of terminals. Such processes that run without any attachment to a terminal are called non-interactive processes.

UNIX PROGRAMMING UNIT – 6

Non – interactive Process:

These are also called as background processes. Certain processes can be made to run independent of terminals; such processes that run without any attachment to a terminal are called non-interactive processes.

Daemon Process :

They are system-related background processes that often run with the permissions of root and services requests from other processes, they most of the time run in the background and wait for processes it can work along with for ex print daemon. When ps –ef is executed, the process with ? in the tty field are daemon processes.

```
root@kali:~# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
user         1      0  0 12:04 ?        00:00:00 /cocalc/bin/tini -- sh -c env -i /cocalc/init/init.sh $COCALC_PROJECT_ID $KUCALC_IMAGE_NAME $COCALC_EXTRA_ENV
user         8      1  0 12:04 ?        00:00:00 sh -c env -i /cocalc/init/init.sh $COCALC_PROJECT_ID $KUCALC_IMAGE_NAME $COCALC_EXTRA_ENV
user         9      8  0 12:04 ?        00:00:00 node /cocalc/src/smc-project/local_hub.js --tcp_port 6000 --raw_port 6001 --kucalc
user        21      9  0 12:04 ?        00:00:00 /usr/sbin/sshd -D -p 2222 -h /tmp/.cocalc/ssh_host_rsa_key -o PidFile=/tmp/.cocalc/ssh.pid -f /cocalc/init/sshd_config
user        441     9  0 12:08 pts/0    00:00:00 /bin/bash
user        721    441  0 12:21 pts/0    00:00:00 ps -ef
root@kali:~#
```

More about Foreground and Background processes

When a command is given, the shell parses, rewrites and hands it over to the kernel for execution. The shell keeps waiting for the kernel to complete the execution. During the shell – waiting period the user cannot issue any other command because the terminal is held up with the command under execution.

Commands that hold up the terminal during their execution are called foreground processes. No further commands can be given in the terminal as long as the older one is running. When a currently running process takes a lot of time for processing.

It is possible to make processes to run without using the terminal are called background processes. Such processes take their input from some file and process it without holding up the terminal and write their output on to another file are called background processes. Typical jobs that could be run in background are sorting a large database file or locating a file in a big file system by using the find command and so on. A command is made to run in the background by terminating the command line with an ampersand (&) character

Example:

```
$ sort -o student.lst student.lst &
567
$
```

The shell immediately returns the PID as well as the shell prompt \$. Here 567 is the PID of the just submitted background job.

UNIX PROGRAMMING UNIT – 6

Problems using background processes:

The success or failure of the background processes is not reported. The user has to find it out using PID. The output has to be redirected to a file as otherwise the display on the monitor gets mixed up. Too many processes running in the background degrades the overall efficiency of the system. There is a danger of the user logging out when some processes are still running in the background.

Internal and External Commands:

The classification of commands depending on whether they generate separate processes or not upon their running. Most of the commands such as cat, who and others generate separate processes as soon as they are used. Commands that generate separate processes upon their running are called external commands. Some commands such as mkdir, rm, cd and other do not generate new processes when they are used. Such commands are called internal commands.

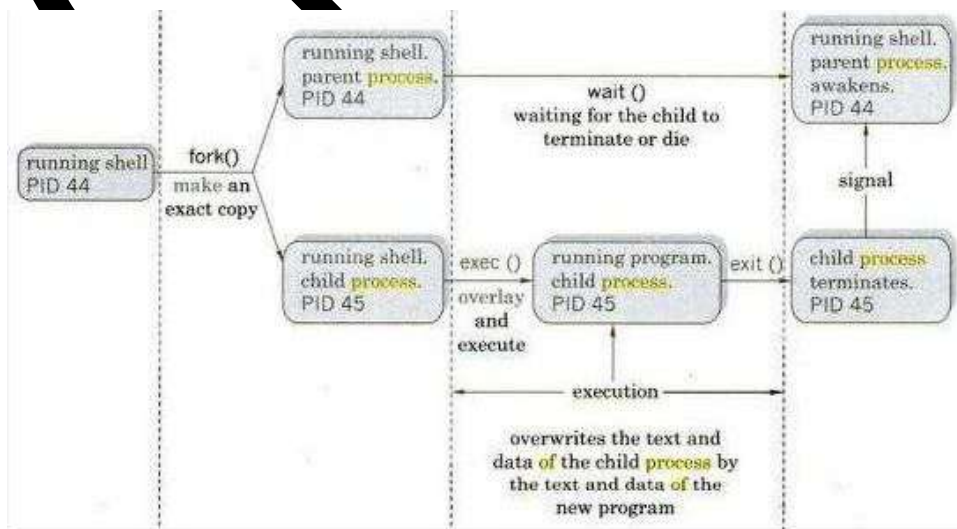
Process Creation:

There are three distinct phases in the creation of a process.

- Forking
- Overlaying and Execution
- Waiting

Mechanism of process creation is depicted as follows.

Forking is the first phase in the creation of a process by a process. The calling process makes a call to the system routine `fork()` which then makes an exact copy of itself. After the `fork()` there will be two processes. The `fork()` of the parent process returns the PID of the new process, that is the child process. The `fork()` of the child returns a 0 (zero). Immediately after forking, the parent makes a system call to one of the `wait()` functions. The parent keeps waiting for the child process to complete its task. The second phase, the `exec()` function overwrites the text and data of the child process. The `exit()` function terminates the child process. The parent awakes only when it receives a complete signal from the child, after which it will be free to continue with its other functions.



UNIX PROGRAMMING UNIT – 6

The ps command (knowing process attributes):

The ps command is used to displays the attributes of processes that are running currently. When used with no options, the ps command lists out certain attributes associated with the terminal.

Example:

```
~$ ps
  PID TTY          TIME CMD
  369 pts/0        00:00:00 bash
  411 pts/0        00:00:00 ps
```

The options used in ps command are

Option	Meaning
-a	All users
-f	Full format
-u	User
-t	Terminal
-e	Every

Example:

```
~$ ps -a
  PID TTY          TIME CMD
  484 pts/0        00:00:00 ps

~$ ps -f
UID          PID    PPID  C  STIME TTY          TIME CMD
user         369      9   0  12:59 pts/0        00:00:00 /bin/bash
user         485     369   0  13:03 pts/0        00:00:00 ps -f

~$ ps -u
USER          PID  %CPU  %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
user          369   0.0   0.0  22264  6000 pts/0    Ss   12:59   0:00 /bin/bash
user          495   0.0   0.0  36152  3276 pts/0    R+   13:04   0:00 ps -u

~$ ps -t
  PID TTY          STAT TIME COMMAND
  369 pts/0        Ss   0:00 /bin/bash
  496 pts/0        R+   0:00 ps -t

~$ ps -e
  PID TTY          TIME CMD
    1 ?           00:00:00 tini
    8 ?           00:00:00 sh
    9 ?           00:00:04 node
   21 ?           00:00:00 sshd
  234 ?           00:00:00 node
  369 pts/0        00:00:00 bash
  497 pts/0        00:00:00 ps
```


UNIX PROGRAMMING UNIT – 6

Signals:

A signal is a message sent to a program under execution, that is a process. It occurs in 2 occasions, under some error conditions or the user interruption, the kernel generates signals, during inter-process communication between two or more processes. The participating process generated these signals, for example, a child process sends a signal to its parent process upon its termination. In Unix, signals are identified by integer. They have names these names are in uppercase and starts with SIG.

Example Signals:

Signal Number	Name	Function
1	SIGHUP	Hangup: close process communication links.
2	SIGINT	Interrupt: tells process to exit
3	SIGQUIT	Quit: forces the process to quit
9	SIGKILL	Sure kill: can not be trapped or ignored
15	SIGTERM	Software termination default signal for the kill command
24	SIGSTOP	Stop

The trap command:

Normally signals are used to terminate the execution of a process either intentionally or unintentionally. The trap command is used to trap one or more signals and then decide about the further course of action. If no action is mentioned, then the signal or signals are just trapped and the execution of the program resumes from the point from where it had been left off.

Syntax:

```
$trap [commands] signal_numbers
```

The command part is optional. If commands used must be enclosed using either single or double quotation marks. Multiple commands in the commands part are separated by the ;(semicolon) character.

Example:

```
$trap "echo killed by signal 15; exit" 15
```

When a process receives a kill command, causing signal 15, it gives the message killed by signal 15 and then terminates the current process because of the execution of the exit command.

Example:

```
$trap "ls -l" 1 2 3
```

When a process generates any one of the signals 1,2 or 3, a long list of the current working directory is generated and then execution of the process resumes from the point where it had been left off.

UNIX PROGRAMMING UNIT – 6

Example:

```
$trap "" 1 2 3 15
```

This command just traps the signal numbers 1,2,3 and 15. Certain signals like signal number 9 (the surekill) can not be trapped.

Resetting Traps:

Normally a trap command changes the default actions of the signals. There are some situations like one might need to trap a certain signal in one part of a script and need the same signals not be trapped in some other part. The command to trap the signal is given as,

Example:

```
$trap "exit" 2 3 15
```

The effect of the signals 2,3 and 15 are restored by using the trap command without the command part in it.

Example:

```
$trap 2 3 15
```

The stty command:

One of the most widely used methods to communicate with a system is to use terminals, that is via keyboards. There are certain combinations of keys, on these terminals, which control the behavior of any program in execution.

Keys	Function
<ctrl-m> (^m)	It is the <RETURN> key to end a command line and execute the command
<ctrl-c> (^c)	To interrupt a current process and to come back to the shell
<ctrl-s> (^s)	To pause display on the monitor
<ctrl-d> (^d)	To indicate end of file and so on

The stty command is used to see r verify the settings of different keys on the keyboard. The user can have a short listing of the setting by using this command without any arguments. In order to see all the settings it has to be used with the -a (all) option.

Example:

```
$stty -a
```

UNIX PROGRAMMING UNIT – 6

```
root@kali:~# stty -a
speed 38400 baud; rows 40; columns 205; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = ^M; eol2 = ^M-^?; swtch = <undef>; start = ^O; stop = ^S; susp = ^Z; rprint = ^R; werase = ^W; lnext = ^V; discard = ^O; min = 1; time = 0;
-parenb -parodd -cmspar cs8 hupcl -cstopb cread -clocal -crtscts
-ignbrk brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff -iuclic ixany inxchel iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ffo
isig icanon ixexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprnt echoctl echoke -flusho -extproc
```

From the output one can see that the terminal speed is 38400 bauds, ^U is used for killing a line, ^D is used to indicate end of file.

The kill command:

There are certain situations when one likes to terminate a process prematurely. Some of these situations are

- When the machine has hung.
- When a running program has gone into an endless loop.
- When a program is doing unintended things.
- When the system performance goes below acceptable levels because of too many background processes.

Terminating a process prematurely is called killing. Killing foreground process is straight forward. This is done by using the DEL key or the BREAK key. To kill a background process a kill command is used. This command is given with the PID of the process to be killed as its argument. If the PID is not known the ps command is used to know the same.

Example:

```
$kill 555
```

Here, 555 is the PID of the process

More than one process can be terminated using a single kill command

Example:

```
$kill 330 333 375 #here 330, 333, 375 are PID's
```

A kill command when invoked, sends a termination signal of the process being killed

When used without any option, it sends 15 as its default signal number. The signal number 15 is known as the software termination signal and is ignored by many processes. For example, the shell process sh, ignores signal 15. Use signal 9, the sure kill signal, to terminate a process forcibly.

Example:

```
$kill -9 666 # 666 is PID
```

UNIX PROGRAMMING UNIT – 6

All the processes of a user (except his/her login shell) can be terminated by using a 0 as the argument of the kill command.

Example:

```
Skill 0      #kills all the processes except the login shell
```

Using 9 as option and 0 as argument, all processes including the shell can be killed.

Example:

```
Skill -9 0   #kills all the processes including the login shell.
```

#! And \$\$ system variables:

The special system variable \$! holds the PID value of the last background job. The last background job can be killed by using the command

```
Skill $!
```

The special system variable \$\$ holds the PID value of the current shell. The current shell can be killed using the sure kill command.

```
Skill -9 $$
```

The wait command:

Sometimes it is necessary to wait for either all the background jobs or a specific job to be executed completely before any further action is initiated. These situations are handled by the wait command.

Example:

```
$wait      #waits till all the background processes are completely executed.
```

```
$wait 227  #waits till the PID 227 process to be completely executed.
```

Job Control:

A command or command line with a number of commands put together or a script is referred to as a job. In UNIX, as one can run commands in the background, there could be a number of commands that are processes running in the background. Also there could be a command -a process – running in the foreground.

The jobs command:

A list of all the current jobs is obtained using the jobs command.

Example:

```
[ksh] jobs      #the {ksh} prompt has been used intentionally
```

K Varada Rajkumar

Assistant Professor

Department of CSE

Sir C R Reddy College Of Engineering

UNIX PROGRAMMING UNIT – 6

[1] + Running vi sample.sh &

[2] – Running sleep 30 &

[ksh]

-Here, a + (plus) and a - (minus) that appear after the job number mark the current and previous jobs respectively. The word running indicated that the job is currently being executed.

The fg command:

This command is used to bring a job that is being executed in the background currently to the foreground. This command can be either used without any argument or with a job number as its argument.

Example 1:

[ksh] fg #Brings the most recent background process to the foreground.

Example 2:

[ksh] fg %2 #Brings job number 2 to the foreground.

Example 3:

[ksh] fg %sort #Brings the job the name of which begins with sort to the foreground.

Whenever a job number is used as an argument with a job-control command, it must be preceded by a percent sign (%). The current job may be referred by using %1 (or) %+ (or) %%.

The bg command:

A new job can be made to run in the background by using the & (ampersand) at the end of a command line. The currently running foreground process is first suspended by using the <ctrl-z> keys, and then making it run in the background by using the bg command.

Example:

[ksh] bg %1 #resumes job number 1 in the background.