Lab Manual

For
Operating System
(CT-257)
(CS-2014)

Bachelor of Technology

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
SCHOOL OF ENGINEERING
SIR PADAMPAT SINGHANIA UNIVERSITY, UDAIPUR
(RAJASTHAN)

Developed By-

Harish Tiwari
Assistant Professor
Department of CSE
School of Engineering
Sir Padampat Singhania University, Udaipur

1. Introduction to subject

The goal of this lab Manual is to provide you the basic information about the operating System. This lab manual comprises all information for the student about the lab like, course outline, Reference book list, evaluation criteria, Marks distribution, and guide line for the student, formats and index and program in the Lab record.

2. Objective of the Subject

An operating system is a program that controls the execution of application programs and acts as an interface between the user of a computer and the computer hardware. In other words" The software that controls the hardware". Some examples of operating systems are UNIX, MS-DOS, MS Windows, Windows/NT, MacOS. Objectives of the operating system are as follows.

- **Convenience** – An operating system makes a computer more convenient to use.
- **Efficiency** – An operating system allows the computer system resources to be used in an efficient manner.
- **Ability to evolve** – An operating system should permit the effective development, testing, and introduction of new system function without interfering with the service.

Services provided by the operating system are as follows.

- Implementing the user interface
- Sharing hardware among users
- Allowing users to share data among themselves
- Preventing users from interfering with one another
- Scheduling resources among users
- Facilitating input/output
- Recovering from errors
- Accounting for resource usage
- Facilitating parallel operations
- Organizing data for secure and rapid access
- Handling network communications.

3. Lab Objective

This manual is used to provide an understanding of the design aspects of operating system. This manual is useful to Study the operating systems functioning and internals and implementation of concepts using C programming language. Student will also gain some understanding of the principles of good program design and program testing.

4. Laboratory Outcome:

At the end of the course, the students should be able to understand basic operating system commands, explore various system calls, write shell scripts and shell commands using kernel APIs, implement and analyze different Process Scheduling Algorithms, Memory

Management algorithms and evaluate process management techniques and deadlock handling using simulator.

5. Evaluation criteria

This subject is offered to the students of VI semester CSE by department of Computer Science and Engineering of the University. The student will be appear in the lab examination twice in the semester, one at the time of Mid Term examination and another at the time of End Term Examination.

The overall marks distribution for the lab is as follows.

S. No	Evaluation Criteria	Mid Term Examination	End Term examination	Total
1.	Practical Files	05	05	10
2.	Lab Attendance	-	05	05
3.	Viva	05	10	15
5.	Lab Written Work	05	15	25
TOTAL				**50**

6. Guidelines to Students:

- Equipment in the lab for the use of student community. Students need to maintain a proper decorum in the computer lab. Students must use the equipment with care. Any damage caused is punishable.
- Students are required to carry their observation / programs book with completed exercises while entering the lab.
- Students are supposed to occupy the machines allotted to them and are not supposed to talk or make noise in the lab. The allocation is put up on the lab notice board.
- Lab can be used in free time / lunch hours by the students who need to use the systems should take prior permission from the lab in-charge.
- Lab records need to be submitted on or before date of submission.
- Students are not supposed to use pen drives/CD.

7. Format of Index

S.No.	**Aim Of the program**	**Date of Performance**	**Date Of Submission**	**Remark**	**Signature**
	write complete aim of the program that student has written in the aim section in every program	*DD/MM/YYYY*	*DD/MM/YYYY*	*Here faculty will write some remark/grade/comment/etc.*	*Signature of the faculty*

8. How to write Program in the Lab Record

Student will need to write program in following format.

Aim : write the complete aim of the program to be developed.

Software Used : what types of different software used to develop this program

1. **IDE/Compiler** :
2. **Operating System** :

Source Code : write complete source code with name of the file in the middle of the sheet and program should be written in proper indentation.

Output : write the complete output with set of inputs entered by user during execution.

9. Program List

1. Explore the internal commands of linux like ls, chdir, mkdir, chown, chmod, chgrp, ps etc.
2. Write shell scripts to do the following:
    a. Display top 10 processes in descending order
    b. Display processes with highest memory usage.
    c. Display current logged in user and log name.
    d. Display current shell, home directory, operating
    e. System type, current path setting, and current working directory.
    f. Display OS version, release number, kernel version.
    g. Illustrate the use of sort, grep, awk, etc.
3. Programs for process management
    a. Create a child process in Linux using the fork( ) system call. From the child process obtain the process ID of both child and parent by using getpid() and getppid() system call.
    b. Explore wait( ) and waitpid( ) before termination of process.
    c. Explore the following system calls: open( ), read( ), write( ), close( ), getpid( ), setpid( ), getuid( ), getgid( ), getegid( ),geteuid( )
4. Implement basic commands of linux like ls, cp, mv and others using kernel APIs.
5. Write a program to implement any two CPU scheduling algorithms like FCFS, SJF, Round Robin etc.
6. Write a program to implement dynamic partitioning placement algorithms i.e Best Fit, Fit, Worst-Fit etc
7. Write a program to implement various page replacement policies.
8. Using the CPU-OS simulator analyze and synthesize the following:
    a. Process Scheduling algorithms.
    b. Thread creation and synchronization.
    c. Deadlock prevention and avoidance.

Module 1 Exploring Linux Commands

The command is followed by options (optional of course) and a list of arguments. The options can modify the behavior of a command. The arguments may be files or directories or some other data on which the command acts. Every command might not need arguments. Some commands work with or without them (e.g. 'ls' command). The options can be provided in two ways: full word options with -- (e.g. --help), or single letter options with - (e.g. -a -b -c or multiple options, -abc).

Linux Basic Commands

Let's start with some simple commands.

1. **pwd command -** 'pwd' command prints the absolute path to current working directory.

```
$ pwd
/home/raghu
```

2. **cal command-** Displays the calendar of the current month.

```
$ cal
July 2012
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
```

'cal ' will display calendar for the specified month and year.

```
$ cal 08 1991
August 1991
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
```

3. **echo command-** This command will echo whatever you provide it.

```
$ echo "spsu.ac.in"
spsu.ac.in
```

The 'echo' command is used to display the values of a variable. One such variable is 'HOME'. To check the value of a variable precede the variable with a $ sign.

```
$ echo $HOME
/home/raghu
```

4. **date command-** Displays current time and date.

```
$ date
Fri Jul 6 01:07:09 IST 2012
```

If you are interested only in time, you can use 'date +%T' (in hh:mm:ss):

```
$ date +%T
01:13:14
```

5. **tty command-** Displays current terminal.

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

6. **whoami command-**This command reveals the user who is currently logged in.

```
$ whoami
raghu
```

7. **id command-** This command prints user and groups (UID and GID) of the current user.

```
$ id
uid=1000(raghu) gid=1000(raghu)
groups=1000(raghu),4(adm),20(dialout),24(cdrom),46(plugdev),112(lp
admin),120(admin),122(sambashare)
```

By default, information about the current user is displayed. If another username is provided as an argument, information about that user will be printed:

```
$ id root
uid=0(root) gid=0(root) groups=0(root)
```

8. **clear command-** This command clears the screen.
9. **Help command-** With almost every command, '--help' option shows usage summary for that command.

```
$ date --help
Usage: date [OPTION]... [+FORMAT] or: date [-u|--utc|--universal]
[MMDDhhmm[[CC]YY][.ss]] Display the current time in the given
FORMAT, or set the system date.
```

10. **whatis command-** This command gives a one line description about the command. It can be used as a quick reference for any command.

```
$ whatis date
date (1) - print or set the system date and time
$ whatis whatis
whatis (1) - display manual page descriptions
```

11. **Manual Pages-** '--help' option and 'whatis' command do not provide thorough information about the command. For more detailed information, Linux provides man pages and info pages. To see a command's manual page, man command is used.

```
$ man date
```

The man pages are properly documented pages. They have following sections:
NAME: The name and one line description of the command.
SYNOPSIS: The command syntax.
DESCRIPTION: Detailed description about what a command does.
OPTIONS: A list and description of all of the command's options.
EXAMPLES: Examples of command usage.
FILES: Any file associated with the command.
AUTHOR: Author of the man page
REPORTING BUGS: Link of website or mail-id where you can report any bug.
SEE ALSO: Any commands related to the command, for further reference.

With -k option, a search through man pages can be performed. This searches for a pattern in the name and short description of a man page.

```
$ man -k gzip
gzip (1) - compress or expand files
lz (1) - gunzips and shows a listing of a gzip'd tar'd archive
tgz (1) - makes a gzip'd tar archive
uz (1) - gunzips and extracts a gzip'd tar'd archive
zforce (1) - force a '.gz' extension on all gzip files
```

12. Info pages

Info documents are sometimes more elaborate than the man pages. But for some commands, info pages are just the same as man pages. These are like web pages. Internal links are present within the info pages. These links are called nodes. Info pages can be navigated from one page to another through these nodes.

```
$ info date
```

File Commands

1. The following Linux Command take you to the '/ home'directory

```
cd /home
```

2. This command go back one level

```
cd ..
```

3. This command takes you two folders back.

```
cd ../..
```

4. This command take you to home directory

```
cd
```

5. This command takes you to the user's home directory

```
cd ~user
```

6. This command takes you to the previous directory

```
cd -
```

"COPY" Commands in Linux

7. This command helps you copy one file to another

```
cp file1 file2
```

8. Copy all files of a directory within the current work directory

```
cp dir/* .
```

9. Copy a directory within the current work directory

```
cp -a /tmp/dir1 .
```

10. Copy a directory

```
cp -a dir1 dir2
```

11. Outputs the mime type of the file as text

```
cp file file1
```

12. Linux Command to create a symbolic link to file or directory

```
ln -s file1 lnk1
```

13. Create a physical link to file or directory

```
ln file1 lnk1
```

14. View files of directory

```
ls
```

15. View files of directory

```
ls -F
```

16. Show details of files and directory

```
ls -l
```

17. Show hidden files

```
ls -a
```

18. Show files and directory containing numbers

```
ls *[0-9]*
```

19. Show files and directories in a tree starting from root

```
lstree
```

20. Create a directory called 'dir1'

```
mkdir dir1
```

21. Create two directories simultaneously

```
mkdir dir1 dir2
```

22. Create a directory tree

```
mkdir -p /tmp/dir1/dir2
```

23. Move a file or directory

```
mv dir/file /new_path
```

24. Show the path of work directory

```
pwd
```

25. Delete file called 'file1'

```
rm -f file1
```

26. Remove a directory called 'dir1' and contents recursively

```
rm -rf dir1
```

27. Remove two directories and their contents recursively

```
rm -rf dir1 dir2
```

28. Delete directory called 'dir1'

```
rmdir dir1
```

30. Modify timestamp of a file or directory - (YYMMDDhhmm)

```
touch -t 0712250000 file1
```

31. Show files and directories in a tree starting from root(1)

```
tree
```

Linux Commands for Process Management

32. The top command gives you information on the processes that currently exist.

```
top
```

33. The htop command is like top, but prettier and smarter.

```
htop
```

34. Use the ps command to list running processes (top and htop list all processes whether active or inactive).

```
ps
```

35. A step up from the simple ps command, pstree is used to display a tree diagram of processes that also shows relationships that exist between them.

```
pstree
```

36. The who command will display a list of all the users currently logged into your Linux system.

```
who
```

37. As its name suggests, kill can be used to terminate a process with extreme prejudice.

```
kill
```

38. The pkill and killall commands can kill a process, given its name.

```
pkill & killall
```

39. pgrep returns the process IDs that match it.

```
pgrep
```

40. With the help of nice command, users can set or change the priorities of processes in Linux.

```
nice
```

41. It is similar to nice command. Use this command to change the priority of an already running process.

```
renice
```

42. Gives the Process ID (PID) of a process

```
pidof
```

43. Gives free hard disk space on your system

```
df
```

44. Gives free RAM on your system

```
free
```

File Permissions

45. **chmod** the command for changing permissions

Syntax: chmod permission dir/file

```
chmod 755 Linux_Directory
chmod 644 Linux_File
```

Different File Permissions

```
rwx rwx rwx = 111 111 111

rw- rw- rw- = 110 110 110

rwx --- --- = 111 000 000

rwx = 111 in binary = 7

rw- = 110 in binary = 6

r-x = 101 in binary = 5

r-- = 100 in binary = 4
```

7 = 4+2+1 (read/write/execute)
6 = 4+2 (read/write)
5 = 4+1 (read/execute)
4 = 4 (read)
3 = 2+1 (write/execute)
2 = 2 (write)
1 = 1 (execute)

Briefing about Permissions in Linux

There is a huge importance with Linux Commands when we discuss about Permissions. No restrictions on permissions. Anybody may do anything. Generally not a desirable setting.

```
777 (rwxrwxrwx)
```

The file's owner may read, write, and execute the file. All others may read and execute the file. This setting is common for programs that are used by all users.

```
755 (rwxr-xr-x)
```

The file's owner may read, write, and execute the file. Nobody else has any rights. This setting is useful for programs that only the owner may use and must be kept private from others.

```
700 (rwx------)
```

All users may read and write the file.

```
666 (rw-rw-rw-)
```

The owner may read and write a file, while all others may only read the file. A common setting for data files that everybody may read, but only the owner may change.

```
644 (rw-r--r--)
```

The owner may read and write a file. All others have no rights. A common setting for data files that the owner wants to keep private.

```
600 (rw-------)
```

How to use "Find Command"

The below Linux Commands gives you better Idea on find commands. You can also check more Find Commands in our other article too.

46. To find a file by name

```
find -name "File1"
```

47. To find a file by name, but ignore the case of the "File1"

```
find -iname "File1"
```

48. To search all files that end in ".conf"

```
find /path -type f -name "*.conf"
```

49. To find all files that are exactly 50 bytes

```
find /path -size 50c
```

50. To find all files less than 50 bytes

```
find /path -size -50c
```

51. To Find all files more than 700 Megabytes

```
find / -size +700M
```

52. To find files that have a modification time of a day ago

```
find / -mtime 1
```

53. To find files that were accessed in less than a day ago

```
find / -atime -1
```

54. To find files that last had their meta information changed more than 3 days ago

```
find / -ctime +3
```

55. To find files that were accessed in less than a minute ago

```
find / -mmin -1
```

56. If we want to match an exact set of permissions

```
find / -perm 644
```

57. If we want to specify anything with at least those permissions

```
find / -perm -644
```

Linux Commands to check Word Count

58. Prints the number of lines in a file.

```
wc -l file_name OR cat file_name | wc -l
```

59. Prints the number of words in a file.

```
wc -w
```

60. Displays the count of bytes in a file.

```
wc -c
```

61. Prints the count of characters from a file.

```
wc -m
```

62. Prints only the length of the longest line in a file.

```
wc -L
```

Linux Commands to know System Information

104. To know only system name, you can use uname command

```
uname
```

105. To view your network hostname

```
uname -n
```

106. To get information about kernel-version

```
uname -v
```

107. To get the information about your kernel release

```
uname -r
```

108. To get the information about your kernel release

```
uname -r
```

109. To print your machine hardware name

```
uname -m
```

110. All this information can be printed at once. The below two commands gives same result.

```
uname -a

cat /proc/version
```

111. Find out information about the Linux distribution and version

```
cat /etc/*release*
```

112. To gather information about file system partitions

```
fdisk -l
```

113. To view mounted file systems.

```
mount
```

114. To view information about your CPU architecture such as number of CPU's, cores, CPU family model, CPU caches, threads, etc. Either of the two below commands gives same output.

```
lscpu

cat /proc/cpuinfo
```

115. To view information about block devices

```
lsblk
```

Extract Information about Hardware Components using "dmidecode"

116. To print information about memory. You can get the similar output with all the below commands.

```
dmidecode -t memory

cat /proc/meminfo

free or free -mt or free -gt
```

117. To print information about system

```
dmidecode -t system
```

118. To print information about BIOS

```
dmidecode -t bios
```

119. To print information about processor

```
dmidecode -t processor
```

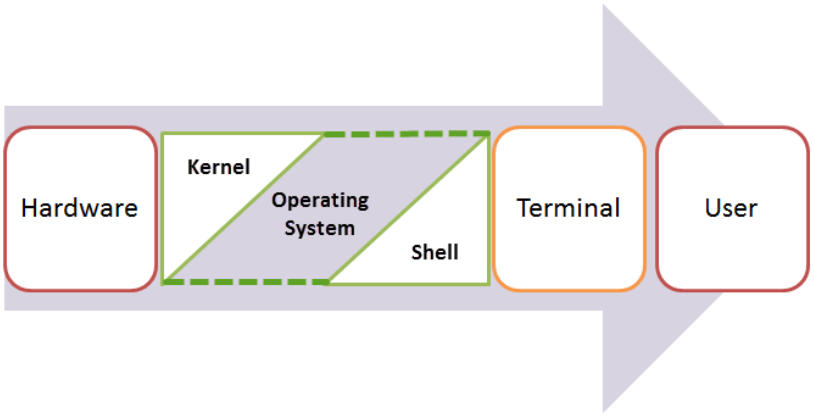
120. To dump all hardware information

```
dmidecode | less
```

Module 2 Exploring Shell Scripts

2.1 Introduction to kernel and shell:

An Operating is made of many components, but its two prime components are Kernel and Shell.



A Kernel is at the nucleus of a computer. It makes the communication between the hardware and software possible. While the Kernel is the innermost part of an operating system, a shell is the outermost one. The kernel is a computer program that is the core of a computer's operating system, with complete control over everything in the system. It manages following resources of the Linux system – File management, Process management, I/O management, Memory management, Device management etc.

A shell is the outermost one. it is special user program which provide an interface to user to use operating system services. Shell accepts human readable commands from user and converts them into something which kernel can understand. The shell gets started when the user logs in or start the terminal. The Shell issues a command prompt (usually $), where you can type your input, which is then executed when you hit the Enter key. The output or the result is thereafter displayed on the terminal.

The Shell wraps around the delicate interior of an Operating system protecting it from accidental damage. Hence the name Shell.

The prompt, $, which is called the command prompt, is issued by the shell. While the prompt is displayed, you can type a command. hell reads your input after you press Enter. You can customize your command prompt using the environment variable PS1.

In UNIX, there are two major types of shells −

- **Bourne shell** − If you are using a Bourne-type shell, the **$** character is the default prompt.
- **C shell** − If you are using a C-type shell, the % character is the default prompt.

The Bourne Shell has the following subcategories −

- Bourne shell (sh)
- Korn shell (ksh)
- Bourne Again shell (bash)
- POSIX shell (sh)

The different C-type shells follow −

- C shell (csh)
- TENEX/TOPS C shell (tcsh)

2.2 Introduction to Shell Scripts

The basic concept of a shell script is a list of commands.Shell scripting is a process of writing a series of command for the shell to execute. It can combine lengthy and repetitive sequences of commands into a single and simple script, which can be stored and executed anytime. This reduces the effort required by the end user.

A shell can also take commands as input from file we can write these commands in a file and can execute them in shell to avoid this repetitive work. These files are called Shell Scripts or Shell Programs. Shell scripts are similar to the batch file in MS-DOS.

2.3 Need shell scripts.

There are many reasons to write shell scripts –

- To avoid repetitive work and automation.
- System admins use shell scripting for routine backups.
- System monitoring.
- Adding new functionality to the shell etc.

2.4 Advantages of shell scripts

- The command and syntax are exactly the same as those directly entered in command line, so programmer do not need to switch to entirely different syntax
- Writing shell scripts are much quicker
- Quick start
- Interactive debugging etc.

2.5 Disadvantages of shell scripts

- Prone to costly errors, a single mistake can change the command which might be harmful
- Slow execution speed
- Design flaws within the language syntax or implementation
- Not well suited for large and complex task
- Provide minimal data structure unlike other scripting languages. Etc

2.6 Example Script

Assume we create a **test.sh** script. Note all the scripts would have the **.sh** extension. Before you add anything else to your script, you need to alert the system that a shell script is being started.

For example −

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

This tells the system that the commands that follow are to be executed by the Bourne shell. It's called a shebang because the # symbol is called a hash, and the ! symbol is called a bang. It directs the script to the interpreter location. So, if we use"#! /bin/sh" the script gets directed to the bourne-shell.

To create a script containing any vGuptad commands, you put the shebang line first and then add the commands −

```
#!/bin/bash
pwd
ls
```

2.6.1 Adding shell comments

In Shell programming, the syntax to add a comment is

```
#comment
```

For example.

```
#!/bin/bash
# Author : Harish Tiwari
# Copyright (c) Spsu.ac.in
# Script follows here:
pwd
ls
```

2.6.2 Saving and Running Shell Script

Each shell script is saved with **.sh** file extension eg. **Example.sh.** make this script file executable as follows:

```
$chmod +x test.sh
```

The shell script is now ready to be executed –

```
$./test.sh
```

Upon execution, you will receive the following result −

```
/home/haish
index.htm  unix-basic_utilities.htm  unix-directories.htm
test.sh    unix-communication.htm    unix-environment.htm
```

Note − To execute a program available in the current directory, use ./program_name

2.7 Unix / Linux - Using Shell Variable

2.7.1 Introduction to shell variable.

Shell variables are used to store information and they can by the shell only. A variable is nothing more than a pointer to the actual data. A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.

The shell enables you to create, assign, and delete variables.

For example, the following creates a shell variable and then prints it:

```
variable ="Hello"
echo $variable
```

Below is a small script which will use a variable.

```
#!/bin/sh
echo "what is your first name?"
read fname
echo " what is your Last name "
read lname
echo "You are $ fname $lname!"
```

2.7.2 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 (_).

By convention, Unix shell variables will have their names in UPPERCASE.

The following examples are vGuptad variable names −

```
_GUPTA
TOKEN_A
VAR_1
VAR_2
```

Following are the examples of invGuptad variable names −

```
2_VAR
-VARIABLE
VAR1-VAR2
VAR_A!
```

The reason you cannot use other characters such as !, *, or - is that these characters have a special meaning for the shell.

2.7.3 Defining Variables

Variables are defined as follows −

```
variable_name=variable_value
```

For example −

```
NAME="Harish Tiwari"
```

The above example defines the variable NAME and assigns the value "Abhishek Gupta" to it. Variables of this type are called scalar variables. A scalar variable can hold only one value at a time.

Shell enables you to store any value you want in a variable. For example −

```
VAR1="Abhishek Gupta"
VAR2=100
```

2.7.4 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 −

```
#!/bin/sh

NAME="Harish Tiwari"
echo $NAME
```

The above script will produce the following value −

```
Harish Tiwari
```

2.7.5 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 −

```
#!/bin/sh

NAME="Abhishek Gupta"
readonly NAME
NAME="Abhishek Agrawal"
```

The above script will generate the following result −

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

2.7.6 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 −

```
unset variable_name
```

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

```
#!/bin/sh

NAME="Abhishek Gupta"
unset NAME
echo $NAME
```

The above example does not print anything. You cannot use the unset command to unset variables that are marked readonly.

2.7.7 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.

2.7.8 Special variable

Unix/Linux are also having some special variable which are having some special predefined meaning. It specifies the reason why we should not use certain non-alphanumeric characters in variable names. This is because those characters are used in the names of special UNIX variables. These variables are reserved for specific functions.

For example, the **$** character represents the process ID number, or PID, of the current shell −

```
$echo $$
```

The above command writes the PID of the current shell −

```
29949
```

The following table shows a number of special variables that you can use in your shell scripts −

Sr.No.	Variable & Description
1	$0- The filename of the current script.
2	$n These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on).
3	$# - The number of arguments supplied to a script.
4	$*- All the arguments are double quoted. If a script receives two arguments,

	$* is equivalent to $1 $2.
5	$@- All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
6	$? -The exit status of the last command executed.
7	$$- The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
8	$!- The process number of the last background command.

2.8 Command-Line Arguments

The command-line arguments $1, $2, $3, ...$9 are positional parameters, with $0 pointing to the actual command, program, shell script, or function and $1, $2, $3, ...$9 as the arguments to the command.

Following script uses various special variables related to the command line −

```
#!/bin/sh

echo "File Name: $0"
echo "First Parameter : $1"
echo "Second Parameter : $2"
echo "Quoted Values: $@"
echo "Quoted Values: $*"
echo "Total Number of Parameters : $#"
```

Here is a sample run for the above script −

```
$./test.sh Abhishek Gupta
File Name : ./test.sh
First Parameter : Abhishek
Second Parameter : Gupta
Quoted Values: Abhishek Gupta
Quoted Values: Abhishek Gupta
Total Number of Parameters : 2
```

2.9 Special Parameters $* and $@

There are special parameters that allow accessing all the command-line arguments at once. $* and $@ both will act the same unless they are enclosed in double quotes, "".

Both the parameters specify the command-line arguments. However, the "$*" special parameter takes the entire list as one argument with spaces between and the "$@" special parameter takes the entire list and separates it into separate arguments.

We can write the shell script as shown below to process an unknown number of commandline arguments with either the $* or $@ special parameters −

```
#!/bin/sh

for TOKEN in $*
do
   echo $TOKEN
done
```

Here is a sample run for the above script −

```
$./test.sh Abhishek Gupta 10 Years Old
Abhishek
Gupta
10
Years
Old
```

Note − Here do...done is a kind of loop that will be covered in a subsequent tutorial.

2.10 Exit Status

The **$?** variable represents the exit status of the previous command.

Exit status is a numerical value returned by every command upon its completion. As a rule, most commands return an exit status of 0 if they were successful, and 1 if they were unsuccessful.

Some commands return additional exit statuses for particular reasons. For example, some commands differentiate between kinds of errors and will return various exit values depending on the specific type of failure.

Following is the example of successful command −

```
$./test.sh Abhishek Gupta
File Name : ./test.sh
First Parameter : Abhishek
Second Parameter : Gupta
Quoted Values: Abhishek Gupta
Quoted Values: Abhishek Gupta
Total Number of Parameters : 2
$echo $?
0
$
```

2.11 Unix / Linux - Shell Basic Operators

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

Bourne shell didn't originally have any mechanism to perform simple arithmetic operations but it uses external programs, either **awk** or **expr**.

The following example shows how to add two numbers −

```
#!/bin/sh

val=`expr 2 + 2`
echo "Total value : $val"
```

The above script will generate the following result −

```
Total value : 4
```

The following points need to be considered while adding −

- There must be spaces between operators and expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
- The complete expression should be enclosed between **‘ ‘**, called the backtick.

2.11.1 Arithmetic Operators

The following arithmetic operators are supported by Bourne Shell.

Assume variable **a** holds 10 and variable **b** holds 20 then −

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator	`expr $a + $b` will give 30
- (Subtraction)	Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
* (Multiplication)	Multiplies values on either side of the operator	`expr $a \* $b` will give 200
/ (Division)	Divides left hand operand by right hand operand	`expr $b / $a` will give 2
% (Modulus)	Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
= (Assignment)	Assigns right operand in left operand	a = $b would assign value of b into a
== (Equality)	Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!= (Not Equality)	Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

It is very important to understand that all the conditional expressions should be inside square braces with spaces around them, for example **[$a == $b]** is correct whereas, **[$a==$b]** is incorrect.

All the arithmetical calculations are done using long integers.

Example: Here is an example which uses all the arithmetic operators −

```
#!/bin/sh

a=10
b=20
```

```
val=`expr $a + $b`
echo "a + b : $val"

val=`expr $a - $b`
echo "a - b : $val"

val=`expr $a \* $b`
echo "a * b : $val"

val=`expr $b / $a`
echo "b / a : $val"

val=`expr $b % $a`
echo "b % a : $val"

if [ $a == $b ]
then
   echo "a is equal to b"
fi

if [ $a != $b ]
then
   echo "a is not equal to b"
fi
```

The above script will produce the following result −

```
a + b : 30
a - b : -10
a * b : 200
b / a : 2
b % a : 0
a is not equal to b
```

2.11.2 Relational Operators

Bourne Shell supports the following relational operators that are specific to numeric values. These operators do not work for string values unless their value is numeric.

For example, following operators will work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".

Assume variable **a** holds 10 and variable **b** holds 20 then −

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[$a -lt $b] is true.

-ge	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -le $b] is true.

It is very important to understand that all the conditional expressions should be placed inside square braces with spaces around them. For example, **[$a <= $b]** is correct whereas, **[$a <= $b]** is incorrect.

Example: Here is an example which uses all the relational operators −

```
#!/bin/sh

a=10
b=20

if [ $a -eq $b ]
then
   echo "$a -eq $b : a is equal to b"
else
   echo "$a -eq $b: a is not equal to b"
fi

if [ $a -ne $b ]
then
   echo "$a -ne $b: a is not equal to b"
else
   echo "$a -ne $b : a is equal to b"
fi

if [ $a -gt $b ]
then
   echo "$a -gt $b: a is greater than b"
else
   echo "$a -gt $b: a is not greater than b"
fi

if [ $a -lt $b ]
then
   echo "$a -lt $b: a is less than b"
else
   echo "$a -lt $b: a is not less than b"
fi

if [ $a -ge $b ]
then
   echo "$a -ge $b: a is greater or  equal to b"
else
   echo "$a -ge $b: a is not greater or equal to b"
fi

if [ $a -le $b ]
```

```
then
   echo "$a -le $b: a is less or  equal to b"
else
   echo "$a -le $b: a is not less or equal to b"
fi
```

The above script will generate the following result −

```
10 -eq 20: a is not equal to b
10 -ne 20: a is not equal to b
10 -gt 20: a is not greater than b
10 -lt 20: a is less than b
10 -ge 20: a is not greater or equal to b
10 -le 20: a is less or  equal to b
```

2.11.3 Boolean Operators

The following Boolean operators are supported by the Bourne Shell. Assume variable **a** holds 10 and variable **b** holds 20 then −

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical **OR**. If one of the operands is true, then the condition becomes true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical **AND**. If both the operands are true, then the condition becomes true otherwise false.	[$a -lt 20 -a $b -gt 100] is false.

Example: Here is an example which uses all the Boolean operators –

```
#!/bin/sh

a=10
b=20

if [ $a != $b ]
then
   echo "$a != $b : a is not equal to b"
else
   echo "$a != $b: a is equal to b"
fi

if [ $a -lt 100 -a $b -gt 15 ]
then
   echo "$a -lt 100 -a $b -gt 15 : returns true"
else
   echo "$a -lt 100 -a $b -gt 15 : returns false"
fi

if [ $a -lt 100 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : returns true"
else
   echo "$a -lt 100 -o $b -gt 100 : returns false"
```

```
fi

if [ $a -lt 5 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : returns true"
else
   echo "$a -lt 100 -o $b -gt 100 : returns false"
fi
```

The above script will generate the following result −

```
10 != 20 : a is not equal to b
10 -lt 100 -a 20 -gt 15 : returns true
10 -lt 100 -o 20 -gt 100 : returns true
10 -lt 5 -o 20 -gt 100 : returns false
```

2.11.4 String Operators

The following string operators are supported by Bourne Shell. Assume variable **a** holds "abc" and variable **b** holds "efg" then −

Operator	Description	Example
=	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a = $b] is not true.
!=	Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero; if it is zero length, then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true.	[-n $a] is not false.
str	Checks if **str** is not the empty string; if it is empty, then it returns false.	[$a] is not false.

Example: Here is an example which uses all the String operators

```
#!/bin/sh

a="abc"
b="efg"

if [ $a = $b ]
then
   echo "$a = $b : a is equal to b"
else
   echo "$a = $b: a is not equal to b"
fi

if [ $a != $b ]
then
   echo "$a != $b : a is not equal to b"
else
```

```
   echo "$a != $b: a is equal to b"
fi

if [ -z $a ]
then
   echo "-z $a : string length is zero"
else
   echo "-z $a : string length is not zero"
fi

if [ -n $a ]
then
   echo "-n $a : string length is not zero"
else
   echo "-n $a : string length is zero"
fi

if [ $a ]
then
   echo "$a : string is not empty"
else
   echo "$a : string is empty"
fi
```

The above script will generate the following result −

```
abc = efg: a is not equal to b
abc != efg : a is not equal to b
-z abc : string length is not zero
-n abc : string length is not zero
abc : string is not empty
```

2.11.5 File Test Operators

We have a few operators that can be used to test various properties associated with a Unix file. Assume a variable **file** holds an existing file name "test" the size of which is 100 bytes and has **read**, **write** and **execute** permission on –

Operator	Description	Example
-b file	Checks if file is a block special file; if yes, then the condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file; if yes, then the condition becomes true.	[-c $file] is false.
-d file	Checks if file is a directory; if yes, then the condition becomes true.	[-d $file] is not true.
-f file	Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true.	[-f $file] is true.
-g file	Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true.	[-g $file] is false.
-k file	Checks if file has its sticky bit set; if yes, then the condition becomes true.	[-k $file] is false.

-p file	Checks if file is a named pipe; if yes, then the condition becomes true.	[-p $file] is false.
-t file	Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true.	[-t $file] is false.
-u file	Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable; if yes, then the condition becomes true.	[-r $file] is true.
-w file	Checks if file is writable; if yes, then the condition becomes true.	[-w $file] is true.
-x file	Checks if file is executable; if yes, then the condition becomes true.	[-x $file] is true.
-s file	Checks if file has size greater than 0; if yes, then condition becomes true.	[-s $file] is true.
-e file	Checks if file exists; is true even if file is a directory but exists.	[-e $file] is true.

Example: Here is an example which uses all the file operators

```
#!/bin/sh

file="/var/www/tutorialspoint/unix/test.sh"

if [ -r $file ]
then
   echo "File has read access"
else
   echo "File does not have read access"
fi

if [ -w $file ]
then
   echo "File has write permission"
else
   echo "File does not have write permission"
fi

if [ -x $file ]
then
   echo "File has execute permission"
else
   echo "File does not have execute permission"
fi

if [ -f $file ]
then
   echo "File is an ordinary file"
else
   echo "This is sepcial file"
fi

if [ -d $file ]
then
   echo "File is a directory"
else
   echo "This is not a directory"
fi
```

```
if [ -s $file ]
then
   echo "File size is not zero"
else
   echo "File size is zero"
fi

if [ -e $file ]
then
   echo "File exists"
else
   echo "File does not exist"
fi
```

The above script will produce the following result −

```
File does not have write permission
File does not have execute permission
This is sepcial file
This is not a directory
File size is not zero
File does not exist
```

2.12 Unix / Linux - Shell Decision Making

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

2.12.1 The if...else statements

If else statements are useful decision-making statements which can be used to select an option from a given set of options.
Unix Shell supports following forms of **if…else** statement −

- if...fi statement
- if...else...fi statement
- if...elif...else...fi statement

if…fi statement

The **if...fi** statement is the fundamental control statement that allows Shell to make decisions and execute statements conditionally.

Syntax

```
if [ expression ]
then
   Statement(s) to be executed if expression is true
fi
```

The Shell expression is evaluated in the above syntax. If the resulting value is true, given statement(s) are executed. If the expression is false then no statement would be executed. Most of the times, comparison operators are used for making decisions.

It is recommended to be careful with the spaces between braces and expression. No space produces a syntax error.

If **expression** is a shell command, then it will be assumed true if it returns **0** after execution. If it is a Boolean expression, then it would be true if it returns true.

Example

```
#!/bin/sh
a=10
b=20
if [ $a == $b ]
then
   echo "a is equal to b"
fi

if [ $a != $b ]
then
   echo "a is not equal to b"
fi
```

The above script will generate the following result –

```
a is not equal to b
```

The if...else...fi statement

The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in a controlled way and make the right choice.

Syntax

```
if [ expression ]
then
   Statement(s) to be executed if expression is true
else
   Statement(s) to be executed if expression is not true
fi
```

The Shell expression is evaluated in the above syntax. If the resulting value is true, given statement(s) are executed. If the expression is false, then no statement will be executed.

Example

The above example can also be written using the *if...else* statement as follows −

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
   echo "a is equal to b"
else
   echo "a is not equal to b"
fi
```

Upon execution, you will receive the following result −

```
a is not equal to b
```

The if...elif...fi statement

The **if...elif...fi** statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.

Syntax

```
if [ expression 1 ]
then
   Statement(s) to be executed if expression 1 is true
elif [ expression 2 ]
then
   Statement(s) to be executed if expression 2 is true
elif [ expression 3 ]
then
   Statement(s) to be executed if expression 3 is true
else
   Statement(s) to be executed if no expression is true
fi
```

This code is just a series of *if* statements, where each *if* is part of the *else* clause of the previous statement. Here statement(s) are executed based on the true condition, if none of the condition is true then *else* block is executed.

Example

```
#!/bin/sh

a=10
b=20
```

```
if [ $a == $b ]
then
   echo "a is equal to b"
elif [ $a -gt $b ]
then
   echo "a is greater than b"
elif [ $a -lt $b ]
then
   echo "a is less than b"
else
   echo "None of the condition met"
fi
```

Upon execution, you will receive the following result −

```
a is less than b
```

2.12.2 The case...esac Statement

You can use multiple **if...elif** statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Unix Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated **if...elif** statements.

There is only one form of **case...esac** statement which has been described in detail here −

case...esac statement

The **case...esac** statement in the Unix shell is very similar to the **switch...case** statement we have in other programming languages like **C** or **C++** and **PERL**, etc.

You can use multiple **if...elif** statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated if...elif statements.

Syntax

The basic syntax of the **case...esac** statement is to give an expression to evaluate and to execute several different statements based on the value of the expression.

The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a default condition will be used.

```
case word in
   pattern1)
      Statement(s) to be executed if pattern1 matches
      ;;
   pattern2)
```

```
      Statement(s) to be executed if pattern2 matches
      ;;
   pattern3)
      Statement(s) to be executed if pattern3 matches
      ;;
   *)
     Default condition to be executed
     ;;
Esac
```

Here the string word is compared against every pattern until a match is found. The statement(s) following the matching pattern executes. If no matches are found, the case statement exits without performing any action.

There is no maximum number of patterns, but the minimum is one.

When statement(s) part executes, the command ;; indicates that the program flow should jump to the end of the entire case statement. This is similar to break in the C programming language.

Example

```
#!/bin/sh

FRUIT="kiwi"

case "$FRUIT" in
   "apple") echo "Apple pie is quite tasty."
   ;;
   "banana") echo "I like banana nut bread."
   ;;
   "kiwi") echo "New Zealand is famous for kiwi."
   ;;
esac
```

Upon execution, you will receive the following result −

```
New Zealand is famous for kiwi.
```

A good use for a case statement is the evaluation of command line arguments as follows −

```
#!/bin/sh

option="${1}"
case ${option} in
   -f) FILE="${2}"
      echo "File name is $FILE"
      ;;
   -d) DIR="${2}"
      echo "Dir name is $DIR"
      ;;
   *)
```

```
      echo "`basename ${0}`:usage: [-f file] | [-d directory]"
      exit 1 # Command to come out of the program with status 1
      ;;
esac
```

Here is a sample run of the above program −

```
$./test.sh
test.sh: usage: [ -f filename ] | [ -d directory ]
$ ./test.sh -f index.htm
$ vi test.sh
$ ./test.sh -f index.htm
File name is index.htm
$ ./test.sh -d unix
Dir name is unix
$
```

2.13 Unix / Linux - Shell Loop Types

A loop is a powerful programming tool that enables you to execute a set of commands repeatedly. In this chapter, we will examine the following types of loops available to shell programmers −

- The while loop
- The for loop
- The until loop
- The select loop

You will use different loops based on the situation. For example, the while loop executes the given commands until the given condition remains true; the until loop executes until a given condition becomes true.

Once you have good programming practice you will gain the expertise and thereby, start using appropriate loop based on the situation. Here, while and for loops are available in most of the other programming languages like C, C++ and PERL, etc.

2.13.1 The while loop

The while loop enables you to execute a set of commands repeatedly until some condition occurs. It is usually used when you need to manipulate the value of a variable repeatedly.

Syntax

```
while command
do
   Statement(s) to be executed if command is true
done
```

Here the Shell command is evaluated. If the resulting value is true, given statement(s) are executed. If command is false then no statement will be executed and the program will jump to the next line after the done statement.

Example

Here is a simple example that uses the while loop to display the numbers zero to nine −

```
#!/bin/sh
a=0
while [ $a -lt 10 ]
do
   echo $a
   a=`expr $a + 1`
done
```

Upon execution, you will receive the following result −

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

Each time this loop executes, the variable a is checked to see whether it has a value that is less than 10. If the value of a is less than 10, this test condition has an exit status of 0. In this case, the current value of a is displayed and later a is incremented by 1.

2.13.2 The for loop

The for loop operates on lists of items. It repeats a set of commands for every item in a list.

Syntax

```
for var in word1 word2 ... wordN
do
   Statement(s) to be executed for every word.
done
```

Here var is the name of a variable and word 1 to word N 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.

Example

Here is a simple example that uses the for loop to span through the given list of numbers −

```
#!/bin/sh
for var in 0 1 2 3 4 5 6 7 8 9
do
   echo $var
done
```

Upon execution, you will receive the following result −

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

Following is the example to display all the files starting with .bash and available in your home. We will execute this script from my root −

```
#!/bin/sh
for FILE in $HOME/.bash*
do
   echo $FILE
done
```

The above script will produce the following result −

```
/root/.bash_history
/root/.bash_logout
/root/.bash_profile
/root/.bashrc
```

2.13.3 The until loop

The while loop is perfect for a situation where you need to execute a set of commands while some condition is true. Sometimes you need to execute a set of commands until a condition is true.

Syntax

```
until command
do
   Statement(s) to be executed until command is true
done
```

Here the Shell command is evaluated. If the resulting value is false, given statement(s) are executed. If the command is true then no statement will be executed and the program jumps to the next line after the done statement.

Example

Here is a simple example that uses the until loop to display the numbers zero to nine −

```
#!/bin/sh
a=0
until [ ! $a -lt 10 ]
do
   echo $a
   a=`expr $a + 1`
done
```

Upon execution, you will receive the following result −

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

2.13.4 The select loop

The select loop provides an easy way to create a numbered menu from which users can select options. It is useful when you need to ask the user to choose one or more items from a list of choices.

Syntax

```
select var in word1 word2 ... wordN
do
   Statement(s) to be executed for every word.
done
```

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.

For every selection, a set of commands will be executed within the loop. This loop was introduced in ksh and has been adapted into bash. It is not available in sh.

Example

Here is a simple example to let the user select a drink of choice −

```
#!/bin/ksh
select DRINK in tea cofee water juice appe all none
do
   case $DRINK in
      tea|cofee|water|all)
         echo "Go to canteen"
         ;;
      juice|appe)
         echo "Available at home"
      ;;
      none)
         break
      ;;
      *) echo "ERROR: Invalid selection"
      ;;
   esac
done
```

The menu presented by the select loop looks like the following −

```
$./test.sh
1) tea
2) cofee
3) water
4) juice
5) appe
6) all
7) none
#? juice
Available at home
#? none
$
```

You can change the prompt displayed by the select loop by altering the variable PS3 as follows −

```
$PS3 = "Please make a selection => " ; export PS3
$./test.sh
1) tea
2) cofee
3) water
4) juice
5) appe
6) all
7) none
Please make a selection => juice
Available at home
Please make a selection => none
$
```

2.13.5 Nesting Loops

All the loops support nesting concept which means you can put one loop inside another similar one or different loops. This nesting can go up to unlimited number of times based on your requirement.

Here is an example of nesting while loop. The other loops can be nested based on the programming requirement in a similar way −

2.13.6 Nesting while Loops

It is possible to use a while loop as part of the body of another while loop.

Syntax

```
while command1 ; # this is loop1, the outer loop
do
   Statement(s) to be executed if command1 is true

   while command2 ; # this is loop2, the inner loop
   do
      Statement(s) to be executed if command2 is true
   done

   Statement(s) to be executed if command1 is true
```

```
done
```

Example

Here is a simple example of loop nesting. Let's add another countdown loop inside the loop that you used to count to nine −

```
#!/bin/sh
a=0
while [ "$a" -lt 10 ]    # this is loop1
do
   b="$a"
   while [ "$b" -ge 0 ]  # this is loop2
   do
      echo -n "$b "
      b=`expr $b - 1`
   done
   echo
   a=`expr $a + 1`
done
```

This will produce the following result. It is important to note how echo -n works here. Here -n option lets echo avoid printing a new line character.

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

2.14 Shell Substitutions

The shell performs substitution when it encounters an expression that contains one or more special characters.

Example- Here, the printing value of the variable is substituted by its value. Same time, "\n" is substituted by a new line −

```
#!/bin/sh

a=10
echo -e "Value of a is $a \n"
```

You will receive the following result. Here the -e option enables the interpretation of backslash escapes.

```
Value of a is 10
```

Following is the result without -e option –

```
Value of a is 10\n
```

The following escape sequences which can be used in echo command −

Sr.No.	Escape & Description
1	**** backslash
2	**\a** alert (BEL)
3	**\b** backspace
4	**\c** suppress trailing newline
5	**\f** form feed
6	**\n** new line
7	**\r** carriage return
8	**\t** horizontal tab
9	**\v** vertical tab

You can use the -E option to disable the interpretation of the backslash escapes (default).

You can use the -n option to disable the insertion of a new line.

2.14.1 Command Substitution

Command substitution is the mechanism by which the shell performs a given set of commands and then substitutes their output in the place of the commands.

Syntax

The command substitution is performed when a command is given as −

```
`command`
```

When performing the command substitution make sure that you use the backquote, not the single quote character.

Example

Command substitution is generally used to assign the output of a command to a variable. Each of the following examples demonstrates the command substitution −

```
#!/bin/sh

DATE=`date`
echo "Date is $DATE"

USERS=`who | wc -l`
echo "Logged in user are $USERS"

UP=`date ; uptime`
echo "Uptime is $UP"
```

Upon execution, you will receive the following result −

```
Date is Thu Jul  2 03:59:57 MST 2009
Logged in user are 1
Uptime is Thu Jul  2 03:59:57 MST 2009
03:59:57 up 20 days, 14:03,  1 user,  load avg: 0.13, 0.07, 0.15
```

2.14.2 Variable Substitution

Variable substitution enables the shell programmer to manipulate the value of a variable based on its state. Here is the following table for all the possible substitutions −

Sr.No.	Form & Description	
1	**${var}**	Substitute the value of *var*.
2	**${var:-word}** value	If *var* is null or unset, *word* is substituted for **var**. The of *var* does not change.
3	**${var:=word}**	If *var* is null or unset, *var* is set to the value of **word**.
4	**${var:?message}** error.	If *var* is null or unset, *message* is printed to standard This checks that variables are set correctly.
5	**${var:+word}**	If *var* is set, *word* is substituted for var. The value of *var* does not change.

Example

Following is the example to show various states of the above substitution −

```
#!/bin/sh

echo ${var:-"Variable is not set"}
echo "1 - Value of var is ${var}"

echo ${var:="Variable is not set"}
echo "2 - Value of var is ${var}"

unset var
echo ${var:+"This is default value"}
echo "3 - Value of var is $var"

var="Prefix"
echo ${var:+"This is default value"}
echo "4 - Value of var is $var"

echo ${var:?"Print this message"}
echo "5 - Value of var is ${var}"
```

Upon execution, you will receive the following result −

```
Variable is not set
1 - Value of var is
```

```
Variable is not set
2 - Value of var is Variable is not set

3 - Value of var is
This is default value
4 - Value of var is Prefix
Prefix
5 - Value of var is Prefix
```

2.15 Shell Script Examples

Aim : List Processes based on %CPU and Memory Usage

Description : This script list the processes based on %CPU and Memory usage, without argument (by default), If you specify the argument (cpu or mem), it lists the processes based on CPU usage or memory usage.

```
#! /bin/bash
#List processes based on %cpu and memory usage

echo "Start Time" `date`
# By default, it display the list of processes based on the cpu and memory usage
#
if [ $# -eq 0 ]
then

      echo "List of processes based on the %cpu Usage"
      ps -e -o pcpu,cpu,nice,state,cputime,args --sort pcpu  # sorted based on
%cpu
      echo "List of processes based on the memory Usage"
      ps -e -orss=,args= | sort -b -k1,1n # sorted bases rss value
      # If arguements are given (mem/cpu)
else
      case "$1" in
      mem)
       echo "List of processes based on the memory Usage"
       ps -e -orss=,args= | sort -b -k1,1n
       ;;
      cpu)
       echo "List of processes based on the %cpu Usage"
       ps -e -o pcpu,cpu,nice,state,cputime,args --sort pcpu
       ;;
      *)
            echo "Invalid Argument Given \n"
            echo "Usage : $0 mem/cpu"
            exit 1
      esac

fi
echo "End Time" `date`
exit 0
```

You can execute the above script as shown below.

```
$ processes.sh
$ processes.sh mem
$ processes.sh cpu
```

Aim : Display Logged in users and who is using high CPU percentage

Description : This script displays few information about the currently logged in users and what they are doing.

```
#! /bin/bash
w > /tmp/a
echo "Total number of unique users logged in currently"
cat /tmp/a|  sed '1,2d' | awk '{print $1}' | uniq | wc -l
echo ""
echo "List of unique users logged in currently"
cat /tmp/a | sed '1,2d'|  awk '{print $1}' | uniq
echo ""
echo "The user who is using high %cpu"
cat /tmp/a | sed '1,2d' | awk   '$7 > maxuid { maxuid=$7; maxline=$0 }; END {
print maxuid, maxline }'

echo ""
echo "List of users logged in and what they are doing"
```

Output

```
$ ./loggedin.sh
Total number of unique users logged in currently
4

List of unique users logged in currently
john
david
raj
reshma

The user who is using high %cpu
0.99s reshma   pts/5    192.168.2.1  15:26    3:01   1.02s  0.99s custom-
download.sh

List of users logged in and what they are doing
 15:53:55 up 230 days,  2:38,  7 users,  load average: 0.19, 0.26, 0.24
USER     TTY      FROM          LOGIN@   IDLE   JCPU   PCPU   WHAT
john     pts/1    192.168.2.9   14:25    1:28m  0.03s  0.03s -bash
john     pts/2    192.168.2.9   14:41    1:11m  0.03s  0.03s -bash
raj      pts/0    192.168.2.6   15:07    9:08   0.11s  0.02s -bash
raj      pts/3    192.168.2.6   15:19    29:29  0.02s  0.02s -bash
john     pts/4    192.168.2.91  15:25    13:47  0.22s  0.20s vim error_log
reshma   pts/5    192.168.2.1   15:26    3:01   1.02s  0.99s custom-download.sh
```

Aim : Display Total, Used and Free Memory
Description : The following script displays the total, used and free memory space.

```
#! /bin/bash
# Total memory space details

echo "Memory Space Details"
free -t -m | grep "Total" | awk '{ print "Total Memory space : "$2 " MB";
print "Used Memory Space : "$3" MB";
print "Free Memory : "$4" MB";
}'

echo "Swap memory Details"
free -t -m | grep "Swap" | awk '{ print "Total Swap space : "$2 " MB";
print "Used Swap Space : "$3" MB";
print "Free Swap : "$4" MB";
}'
```

Output

```
$ ./mem.sh
Memory Space Details
Total Memory space : 4364 MB
Used Memory Space : 451 MB
Free Memory : 3912 MB
Swap memory Details
Total Swap space : 2421 MB
Used Swap Space : 0 MB
Free Swap : 2421 MB
```

Aim : Write shell script to show various system configurations like
1) Currently logged user and his log name
2) Your current shell
3) Your home directory
4) Your operating system type
5) Your current path setting
6) Your current working directory
7) Show Currently logged number of users
8) About your os and version ,release number , kernel version
9) Show all available shells
10) Show mouse settings
11) Show computer cpu information like processor type, speed etc
12) Show memory information
13) Show hard disk information like size of hard-disk, cache memory, model etc
14) File system (Mounted)

```
#!/bin/bash

nouser=`who | wc -l`
echo -e "User name: $USER (Login name: $LOGNAME)" >> /tmp/info.tmp.01.$$$
echo -e "Current Shell: $SHELL"  >> /tmp/info.tmp.01.$$$
echo -e "Home Directory: $HOME" >> /tmp/info.tmp.01.$$$
echo -e "Your O/s Type: $OSTYPE" >> /tmp/info.tmp.01.$$$
echo -e "PATH: $PATH" >> /tmp/info.tmp.01.$$$
echo -e "Current directory: `pwd`" >> /tmp/info.tmp.01.$$$
echo -e "Currently Logged: $nouser user(s)" >> /tmp/info.tmp.01.$$$

if [ -f /etc/redhat-release ]
then
    echo -e "OS: `cat /etc/redhat-release`" >> /tmp/info.tmp.01.$$$
fi

if [ -f /etc/shells ]
then
    echo -e "Available Shells: " >> /tmp/info.tmp.01.$$$
    echo -e "`cat /etc/shells`"  >> /tmp/info.tmp.01.$$$
fi

if [ -f /etc/sysconfig/mouse ]
then
    echo -e "---------------------------------------" >> /tmp/info.tmp.01.$$$
    echo -e "Computer Mouse Information: " >> /tmp/info.tmp.01.$$$
    echo -e "---------------------------------------" >> /tmp/info.tmp.01.$$$
    echo -e "`cat /etc/sysconfig/mouse`" >> /tmp/info.tmp.01.$$$
fi
echo -e "-----------------------------------------" >> /tmp/info.tmp.01.$$$
echo -e "Computer CPU Information:" >> /tmp/info.tmp.01.$$$
echo -e "-----------------------------------------" >> /tmp/info.tmp.01.$$$
cat /proc/cpuinfo >> /tmp/info.tmp.01.$$$

echo -e "-----------------------------------------" >> /tmp/info.tmp.01.$$$
```

```
echo -e "Computer Memory Information:" >> /tmp/info.tmp.01.$$$
echo -e "------------------------------------------" >> /tmp/info.tmp.01.$$$
cat /proc/meminfo >> /tmp/info.tmp.01.$$$

if [ -d /proc/ide/hda ]
then
    echo -e "-------------------------------------" >> /tmp/info.tmp.01.$$$
    echo -e "Hard disk information:" >> /tmp/info.tmp.01.$$$
    echo -e "-------------------------------------" >> /tmp/info.tmp.01.$$$
    echo -e "Model: `cat /proc/ide/hda/model` " >> /tmp/info.tmp.01.$$$
    echo -e "Driver: `cat /proc/ide/hda/driver` " >> /tmp/info.tmp.01.$$$
    echo -e "Cache size: `cat /proc/ide/hda/cache` " >> /tmp/info.tmp.01.$$$
fi
echo -e "------------------------------------------" >> /tmp/info.tmp.01.$$$
echo -e "File System (Mount):" >> /tmp/info.tmp.01.$$$
echo -e "------------------------------------------" >> /tmp/info.tmp.01.$$$
cat /proc/mounts >> /tmp/info.tmp.01.$$$

if which dialog > /dev/null
then
    dialog  --backtitle "Linux Software Diagnostics (LSD) Shell Script Ver.1.0" --title
"Press Up/Down Keys to move" --textbox  /tmp/info.tmp.01.$$$ 21 70
else
    cat /tmp/info.tmp.01.$$$ |more
fi

rm -f /tmp/info.tmp.01.$$$
```

Aim : Write a Program that takes one or more file/directory names as command line input and reports the following information on the file.

A) File type

B) Number of links.

C) Time of last access.

D) Read,Write and Execute permissions.

Program

```
clear
for i in $*
do
if [ -d $i ]
then
echo “Given directory name is found as $i”
fi
if [ -f $i ]
then
echo “Given name is a file as $i “
fi
echo “Type of file/directory $i”
file $i
echo “Last access time is:”
ls -l$i | cut-c 31-46
echo  "no.of links"
```

```
ln $i
if [ -x $i –a -w $i-a –r $i ]
then
echo “$i contains all permission”
else
echo “$i does not contain all permissions”
fi
done
```

Output:

```
student@ubuntu:~$sh prg12.sh ff1
given name is file ff1
Type of file/directory ff1
last access time
2012-07-07 10:1
No.of links
ff1 does not contain all permissions
```

2.16 Program List :

1. Write shell script to display top 10 processes in descending order
2. Write shell script to Display processes with highest memory usage.
3. Write shell script to Display current logged in user and log name.
4. Write shell script to Display current shell, home directory, operating
5. Write shell script to System type, current path setting, and current working directory.
6. Write shell script to Display OS version, release number, kernel version.
7. Write shell script to Illustrate the use of sort, grep, awk, etc.

Module 3 CPU Scheduling Algorithms.

OBJECTIVE :

- To study CPU Scheduling.
- To study CPU Scheduling algorithms such as FCFS, SJF, Priority and Round Robin.

THEORY :

CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it. The CPU is allocated to the selected process by the dispatcher (It is the module that gives control of the CPU to the processes by short-term scheduler). Scheduling is a fundamental operating system function.
In a single processor system, only one process can run at a time; any other process must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

CPU scheduling decisions may take place under the following four circumstances:
- When a process switches from the running state to the waiting state
- When a process switches from the running state to the ready state.
- When a process switches from the waiting state to the ready state.
- When a process terminates.

Depending on the above circumstances the two types of scheduling are:
1. NON-PREEMPTIVE
2. PREEMPTIVE

1. ***NON-PREEMPTIVE*:** Under this scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.
2. ***PREEMPTIVE:*** Under this scheduling, once the CPU has been allocated to a process, the process does not keep the CPU but can be utilized by some other process. This incurs a cost associated with access to shared data. It also affects the design of the operating system kernel.

SCHEDULING CRITERIA:

- CPU utilization - It can range from 0-100%.In a real system, it ranges should range from 40- 90%.
- Throughput: Number of processes that are completed per unit time.
- Turnaround time: How long a process takes to execute. It is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O
- Waiting time: It is the sum of the periods spent waiting in the ready queue.

- Response time: Time from the submission of a request until the first response is produced. It is desirable to maximize CPU utilization and Throughput and minimize Turnaround time, waiting time and Response time.

3.1 SCHEDULING TECHNIQUES:

- FCFS
- SJF
- PRIORITY
- ROUND ROBIN

3.1.1 FCFS (First-Come, First-Served):

- It is the simplest algorithm and NON-PREEMPTIVE.
- The process that requests the CPU first is allocated the CPU first.
- The implementation is easily managed by queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue
- The average waiting time, however, is long. It is not minimal and may vary substantially if the process's CPU burst time varies greatly.
- This algorithm is particularly troublesome for time-sharing systems.

3.1.2 SJF (Shortest Job First):

- This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are same, FCFS is used to break the tie.
- It is also called shortest next CPU burst algorithm or shortest remaining time first scheduling.
- It is provably optimal, in that it gives the minimum average waiting time for a given set of processes.
- The real difficulty with SJF knows the length of the next CPU request.
- It can be either PREEMPTIVE (SRTF- Shortest Remaining Time First) or NON-PREEMPTIVE.

3.1.3 PRIORITY SCHEDULING:

- The SJF is a special case of priority scheduling.
- In priority scheduling algorithm, a priority is associated with each process, and the CPU is allocated to the process with the highest priority.
- Equal-priority processes are scheduled in FCFS order.
- It can be either PREEMPTIVE or NON-PREEMPTIVE.

- Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. We use as low numbers represent high priority.
- A major problem with priority scheduling algorithms is indefinite blocking, or starvation.
- A solution to starvation is AGING. It is a technique of gradually increasing the priority of process that wait in the system for long time. For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes.

3.1.4 ROUND ROBIN SCHEDULING:

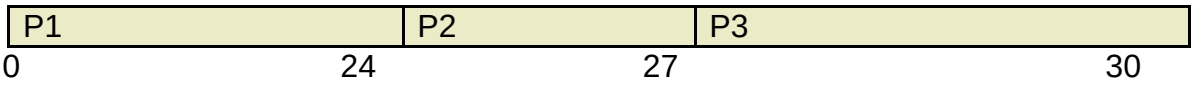
- It is designed especially for time-sharing systems.
- It is similar to FCFS, but preemption is added to switch between processes.
- A time quantum is defined.
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum. If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue.

3.2 Scheduling Algorithms Examples

1. First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P1	24
P2	3
P3	3

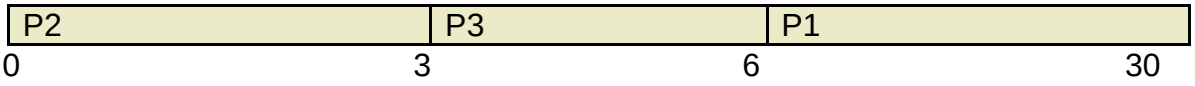
Suppose that the processes arrive in the order: P1 , P2 , P3 .The Gantt Chart for the schedule is



Waiting time for P1 = 0; P2 = 24; P3 = 27
Average waiting time: (0 + 24 + 27)/3 = 17

Suppose that the processes arrive in the order P2 , P3 , P1 .

The Gantt chart for the schedule is:



Waiting time for P1 = 6; P2 = 0; P3 = 3
Average waiting time: (6 + 0 + 3)/3 = 3

Much better than previous case. Convoy effect short process behind long process. So.

Process	Burst time	waiting time	Turnaround time.
P1	24	0	24
P2	3	24	27
P3	3	27	30

Average 17 27

2. Shortest-Job-First (SJR) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used.

As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Waiting Time	Turnaround Time
P1	6	3	9
P2	8	16	24
P3	7	9	16
P4	3	0	3
Average	**-**	**7**	**13**

The Gantt chart is as follows

P4	P1	P3	P2
0	3	9	16 24

There are 2 schemes for SJF:

- **Non preemptive** – once CPU given to the process it cannot be preempted until completes its CPU burst.
- **Preemptive** – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF).

SJF is optimal – gives minimum average waiting time for a given set of processes.

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

SJF (non-preemptive)

Then the Gantt chart for SJF algorithm(Non preemptive) is as follows.

P1	P3	P2	P4
0	7	8	12	16

So

Process	Arrival Time	Burst time	waiting time	Turnaround time.
P1	0	7	0	7
P2	2	4	8	12
P3	4	1	7	8
P4	5	4	12	16

Average waiting time = [0 +(8-2)+(7-4) +(12-5)] /4 =4

Example of Preemptive SJF

Proces	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

Then the Gantt chart for SJF algorithm (preemptive)

P1	P2	P3	P2	P4	P1
0	2	4	5	7	11	16

Average waiting time = (9 + 1 + 0 +2)/4 =3

3. Priority Scheduling

A priority number (integer) is associated with each process
The CPU is allocated to the process with the highest priority (smallest integer =highest priority).

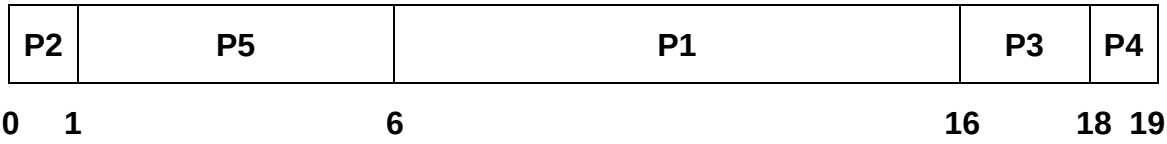
1. Preemptive
2. nonpreemptive

SJF is a priority scheduling where priority is the predicted next CPU burst time.

Problem ≡ Starvation – low priority processes may never execute.
Solution ≡ Aging – as time progresses increase the priority of the process.

As an example, consider the following set of processes, assumed to have arrived at time 0, in the order P1, P2, P3, P4, P5, with the length of the CPU burst given in milliseconds:

The Gantt chart is



So

Process	Priority	Burst time	waiting time	Turnaround time.
P1	3	10	6	16
P2	1	1	0	1
P3	4	2	16	18
P4	5	1	18	19
P5	2	5	1	6
Average			**8.2**	**12**

4. Round Robin (RR)

Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

To implement RR scheduling,

- We keep the ready queue as a FIFO queue of processes.
- New processes are added to the tail of the ready queue.
- The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.
- The process may have a CPU burst of less than 1 time quantum.
- In this case, the process itself will release the CPU voluntarily.
- The scheduler will then proceed to the next process in the ready queue.
- Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum,
- The timer will go off and will cause an interrupt to the OS.
- A context switch will be executed, and the process will be put at the tail of the ready queue.
- The CPU scheduler will then select the next process in the ready queue.

Performance

- q large _ FIFO
- q small _ q must be large with respect to context switch, otherwise overhead is too high.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds: (a time quantum of 4 milliseconds)

Process	Burst Time
P1	24
P2	3
P3	3

The Gantt chart is:

P1	P2	P3	P1	P1	P1	P1	P1

0 4 7 10 14 18 22 26 30

Average waiting time = [(30-24)+4+7]/3 = 17/3 =5.66

Process	Burst time	waiting time	Turnaround time.
P1	24	6	30
P2	3	4	7
P3	3	7	10
Average		**5.66**	**15.66**

3.3 IMPLEMENTATION

- **FIRST COME FIRST SERVE SCHEDULING ALGORITHM**

Step 1: Declare necessary variables.
Step 2: Get the number of processes to be inserted
Step 3: Get the value for burst time of each process from the user
Step 4: Having allocated the burst time(bt) for individual processes , Start
Step 5: with the first process from its initial position let other process to be in queue
Step 6: Calculate the waiting time(wt) and turnaround time(tat) as
Step 7: Wt(pi) = wt(pi-1) + tat(pi-1) (i.e wt of current process = wt of
previous process + tat of previous process)
Step 8: tat(pi) = wt(pi) + bt(pi) (i.e tat of current process = wt of current
Step 9: process + bt of current process)
Step 10: Calculate the total and average waiting time and turn around time
Step 11: Display the values
Step 12: Stop the process

- **SHORTEST JOB FIST SCHEDULING ALGORITHM**

Step 1: Start the process
Step 2: Get the number of processes to be inserted
Step 3: Sort the processes according to the burst time and allocate the one with shortest burst to execute first
Step 4: If two process have same burst length then FCFS scheduling algorithm is used
Step 5: Calculate the total and average waiting time and turn around time
Step 6: Display the values
Step 7: Stop the process.

- **PRIORITY SCHEDULING ALGORITHM**

Step 1: Start the process
Step 2: Get the number of processes to be inserted
Step 3: Get the corresponding priority of processes
Step 4: Sort the processes according to the priority and allocate the one with highest priority to execute first
Step 5: If two process have same priority then FCFS scheduling algorithm is used
Step 6: Calculate the total and average waiting time and turn around time
Step 7: Display the values
Step 8: Stop the process

- **ROUND ROBIN SCHEDULING ALGORITHM**

Step 1: Start the process
Step 2: Get the number of elements to be inserted
Step 3: Get the value for burst time for individual processes
Step 4: Get the value for time quantum
Step 5: Make the CPU scheduler go around the ready queue allocating CPU to each process for the time interval specified
Step 6: Make the CPU scheduler pick the first process and set time to interrupt after quantum. And after it's expiry dispatch the process

Step 7: If the process has burst time less than the time quantum then the process is released by the CPU.
Step 8: If the process has burst time greater than time quantum then it is interrupted by the OS and the process is put to the tail of ready queue

Module 5 System Calls

5.1 Process related System call

5.1.1 fork () System call

Used to create new processes. The new process consistsof a copy of the address space of the original process. The value of process id for the child process is zero, whereas the value of process id for the parent is an integer value greater than zero.

5.1.2 execlp( )

Used after the fork() system call by one of the two processes to replace the process‟ memory space with a new program. It loads a binary file into memory destroying the memory image of the program containing the execlp system call and starts its execution. The child process overlays its address space with the UNIX command /bin/ls using the execlp system call.

5.1.3 wait( )

The parent waits for the child process to complete using the wait system call. The wait system call returns the process identifier of a terminated child, so that the parent can tell which of its possibly many children has terminated.

5.1.4 exit( )

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit system call. At that point, the process may return data (output) to its parent process (via the wait system call).

AIM : To write the program to create a Child Process using system call fork().

ALGORITHM :

Step 1 : Declare the variable pid.
Step 2 : Get the pid value using system call fork().
Step 3 : If pid value is less than zero then print as “Fork failed”.
Step 4 : Else if pid value is equal to zero include the new process in the system‟s file using execlp system call.
Step 5 : Else if pid is greater than zero then it is the parent process and it waits till the child completes using the system call wait()
Step 6 : Then print “Child complete”.

PROGRAM CODING :

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

void main(int argc,char *arg[])
{
int pid;
pid=fork();
if(pid<0)
	{
	printf("fork failed");
	exit(1);
	}
	else if(pid==0)
		{
		execlp("whoami","ls",NULL);
		exit(0);
		}
		else
		{
			printf("\n Process id is -%d\n",getpid());
			wait(NULL);
			exit(0);
		}
	}
}
```

5.1.5 getppid() and getpid() in Linux

Both getppid() and getpid() are inbuilt functions defined in **unistd.h** library.

- **getppid() :** returns the process ID of the parent of the calling process. If the calling process was created by the **fork()** function and the parent process still exists at the time of the getppid function call, this function returns the process ID of the parent process. Otherwise, this function returns a value of 1 which is the process id for **init** process.
  Syntax: pid_t getppid(void);
  Return type: getppid() returns the process ID of the parent of the current process. It never throws any error therefore is always successful.

```
#include <iostream>
#include <unistd.h>
using namespace std;

// Driver Code
int main()
{
    int pid;
    pid = fork();
```

```
    if (pid == 0)
    {
        cout << "\nParent Process id : "
             << getpid() << endl;
        cout << "\nChild Process with parent id : "
             << getppid() << endl;
    }
    return 0;
}
```

- **getpid() :** returns the process ID of the calling process. This is often used by routines that generate unique temporary filenames.

  Syntax: `pid_t getpid(void);`
  Return type: getpid() returns the process ID of the current process. It never throws any error therefore is always successful.

```
#include <iostream>
#include <unistd.h>
using namespace std;

// Driver Code
int main()
{
    int pid = fork();
    if (pid == 0)
        cout << "\nCurrent process id of Process : "
             << getpid() << endl;
    return 0;
}
```

AIM : To write the program to implement the system calls getpid() and getppid().

ALGORITHM :

Step 1 : Declare the variables pid , parent pid , child id and grand chil id.
Step 2 : Get the child id value using system call fork().
Step 3 : If child id value is less than zero then print as “error at fork() child”.
Step 4 : If child id !=0 then using getpid() system call get the process id.
Step 5 : Print “I am parent” and print the process id.
Step 6 : Get the grand child id value using system call fork().
Step 7 : If the grand child id value is less than zero then print as “error at fork() grand child”.
Step 8 : If the grand child id !=0 then using getpid system call get the process id.
Step 9 : Assign the value of pid to my pid.
Step 10 : Print “I am child” and print the value of my pid.
Step 11 : Get my parent pid value using system call getppid().
Step 12 : Print “My parent‟s process id” and its value.

Step 13 : Else print “I am the grand child”.
Step 14 : Get the grand child‟s process id using getpid() and print it as “my process id”.
Step 15 : Get the grand child‟s parent process id using getppid() and print it as “my parent‟s process id

PROGRAM CODING:

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main( )
{
int pid;
pid=fork( );
if(pid==-1)
{
    perror(“fork failed”); exit(0);
}
if(pid==0)
{
    printf(“\n Child process is under execution”);
    printf(“\n Process id of the child process is %d”, getpid());
    printf(“\n Process id of the parent process is %d”, getppid());
}
Else
{
    printf(“\n Parent process is under execution”);
    printf(“\n Process id of the parent process is %d”, getpid());
    printf(“\n Process id of the child process in parent is %d”, pid());
    printf(“\n Process id of the parent of parent is %d”, getppid());
}
    return(0);
}
```

5.1.6 getuid() , geteuid(), getgid(), getegid

Syntax:

```
#include <sys/types.h>
#include <unistd.h>

uid_t getuid(void);
uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);
```

Description

The **getuid()** function returns the real user ID of the calling process. The real user ID identifies the person who is logged in.

The **geteuid()** function returns the effective user ID of the calling process. The effective user ID gives the process various permissions during execution of "set-user-ID" mode processes which use getuid() to determine the real user ID of the process that invoked them.

The **getgid() function** returns the real group ID of the calling process.

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main(void)
{
printf("Get a real user ID:%\n", getuid());
printf("Get the effective user ID:%\n", geteuid());
printf("Get the real group ID:%\n", getgid());
printf("Get the effective group ID:%\n", getegid());
}
```

5.2 File and Directory related System calls

Basically there are total 5 types of I/O system calls:

5.2.1 create () System Call : Used to Create a new empty file.

Syntax in C language: int creat(char *filename, mode_t mode)

Parameter :

- **filename :** name of the file which you want to create
- **mode :** indicates permissions of new file.

Returns :

- return first unused file descriptor (generally 3 when first creat use in process beacuse 0, 1, 2 fd are reserved) return -1 when error

How it work in OS

- Create new empty file on disk
- Create file table entry
- Set first unused file descriptor to point to file table entry
- Return file descriptor used, -1 upon failure

5.2.2 open () System Call : Used to Open the file for reading, writing or both.

Syntax in C language

```
#include<sys/types.h>
#includ<sys/stat.h>
#include <fcntl.h>
int open (const char* Path, int flags [, int mode ]);
```

Parameters

- **Path :** path to file which you want to use absolute path begin with "/", when you are not work in same directory of file. Use relative path which is only file name with extension, when you are work in same directory of file.

- **flags :** How you like to use
  - **O_RDONLY**: read only,
  - **O_WRONLY**: write only,
  - **O_RDWR**: read and write,
  - **O_CREAT**: create file if it doesn't exist,
  - **O_EXCL**: prevent creation if it already exists

How it works in OS

- Find existing file on disk
- Create file table entry
- Set first unused file descriptor to point to file table entry
- Return file descriptor used, -1 upon failure

```
#include<stdio.h>
#include<fcntl.h>
#include<errno.h>
extern int errno;
int main()
{
    // if file does not have in directory, then file foo.txt is created.
    int fd = open("foo.txt", O_RDONLY | O_CREAT);
    printf("fd = %d/n", fd);
    if (fd ==-1)
    {   // print which type of error have in a code
        printf("Error Number % d\n", errno);
        // print program detail "Success or failure"
        perror("Program");
    }
    return 0;
}
```

Output:

```
fd = 3
```

5.2.3 Close () System Call :

Tells the operating system you are done with a file descriptor and Close the file which pointed by fd.

Syntax in C language

```
#include <fcntl.h>
int close(int fd);
```

Parameter

fd :file descriptor

Return

- **0** on success.
- **-1** on error.

How it works in the OS

- Destroy file table entry referenced by element fd of file descriptor table – As long as no other process is pointing to it!
- Set element fd of file descriptor table to **NULL**

```
// C program to illustrate close system Call
#include<stdio.h>
#include <fcntl.h>
int main()
{
    int fd1 = open("foo.txt", O_RDONLY);
    if (fd1 < 0)
    {
        perror("c1");
        exit(1);
    }
    printf("opened the fd = % d\n", fd1);

    // Using close system Call
    if (close(fd1) < 0)
    {
        perror("c1");
        exit(1);
    }
    printf("closed the fd.\n");
}
```

Output:

```
opened the fd = 3
closed the fd.
```

```
// C program to illustrate close system Call
#include<stdio.h>
#include<fcntl.h>
int main()
{
    // assume that foo.txt is already created
    int fd1 = open("foo.txt", O_RDONLY, 0);
    close(fd1);

    // assume that baz.tzt is already created
    int fd2 = open("baz.txt", O_RDONLY, 0);

    printf("fd2 = % d\n", fd2);
    exit(0);
}
```

Output:

```
fd2 = 3
```

Here, In this code first open() returns **3** because when main process created, then fd **0, 1, 2** are already taken by **stdin**, **stdout** and **stderr**. So first unused file descriptor is **3** in file descriptor table. After that in close() system call is free it this **3** file descriptor

and then after set **3** file descriptor as **null**. So when we called second open(), then first unused fd is also **3**. So, output of this program is **3**.

5.2.4 Read () System Call :

From the file indicated by the file descriptor fd, the read() function reads cnt bytes of input into the memory area indicated by buf. A successful read() updates the access time for the file.

Syntax in C language

```
size_t read (int fd, void* buf, size_t cnt);
```

Parameters

- **fd:** file descripter
- **buf:** buffer to read data from
- **cnt:** length of buffer

Returns: How many bytes were actually read

- return Number of bytes read on success
- return 0 on reaching end of file
- return -1 on error
- return -1 on signal interrupt

Important points

- **buf** needs to point to a valid memory location with length not smaller than the specified size because of overflow.
- **fd** should be a valid file descriptor returned from open() to perform read operation because if fd is NULL then read should generate error.
- **cnt** is the requested number of bytes read, while the return value is the actual number of bytes read. Also, some times read system call should read less bytes than cnt.

```
 // C program to illustrate
// read system Call
#include<stdio.h>
#include <fcntl.h>
int main()
{
  int fd, sz;
  char *c = (char *) calloc(100, sizeof(char));

  fd = open("foo.txt", O_RDONLY);
  if (fd < 0) { perror("r1"); exit(1); }

  sz = read(fd, c, 10);
  printf("called read(% d, c, 10).  returned that"
         " %d bytes  were read.\n", fd, sz);
  c[sz] = '\0';
  printf("Those bytes are as follows: % s\n", c);
}
```

Output:

```
called read(3, c, 10).  returned that 10 bytes  were read.
```

```
Those bytes are as follows: 0 0 0 foo.
```

Suppose that foobar.txt consists of the 6 ASCII characters “foobar”. Then what is the output of the following program?

```
// C program to illustrate
// read system Call
#include<stdio.h>
#include<fcntl.h>

int main()
{
    char c;
    int fd1 = Open("foobar.txt", O_RDONLY, 0);
    int fd2 = Open("foobar.txt", O_RDONLY, 0);
    Read(fd1, &c, 1);
    Read(fd2, &c, 1);
    printf("c = % c\n", c);
    exit(0);
}
```

Output:

```
c = f
```

The descriptors ***fd1*** and ***fd2*** each have their own open file table entry, so each descriptor has its own file position for ***foobar.txt***. Thus, the read from ***fd2*** reads the first byte of ***foobar.txt***, and the output is **c = f**, not **c = o**.

5.2.5 write () System Call :

Writes cnt bytes from buf to the file or socket associated with fd. cnt should not be greater than INT_MAX (defined in the limits.h header file). If cnt is zero, write() simply returns 0 without attempting any other action.

Syntax in C language

```
#include <fcntl.h>
size_t write (int fd, void* buf, size_t cnt);
```

Parameters

- **fd:** file descripter
- **buf:** buffer to write data to
- **cnt:** length of buffer

Returns: How many bytes were actually written

- return Number of bytes written on success
- return 0 on reaching end of file
- return -1 on error
- return -1 on signal interrupt

Important points

- The file needs to be opened for write operations
- **buf** needs to be at least as long as specified by cnt because if buf size less than the cnt then buf will lead to the overflow condition.

- **cnt** is the requested number of bytes to write, while the return value is the actual number of bytes written. This happens when **fd** have a less number of bytes to write than cnt.
- If write() is interrupted by a signal, the effect is one of the following: -If write() has not written any data yet, it returns -1 and sets errno to EINTR. -If write() has successfully written some data, it returns the number of bytes it wrote before it was interrupted.

```
// C program to illustrate
// write system Call
#include<stdio.h>
#include <fcntl.h>
main()
{
  int sz;
  int fd = open("foo.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  if (fd < 0)
  {
     perror("r1");
     exit(1);
  }
  sz = write(fd, "hello geeks\n", strlen("hello geeks\n"));
  printf("called write(% d, \"hello geeks\\n\", %d)."
    " It returned %d\n", fd, strlen("hello geeks\n"), sz);
  close(fd);
}
```

Output:

```
called write(3, "hello geeks\n", 12).  it returned 11
```

Here, when you see in the file foo.txt after running the code, you get a "*hello geeks*". If foo.txt file already have some content in it then write system call overwrite the content and all previous content are ***deleted*** and only "*hello geeks*" content will have in the file.

AIM : To write the program to implement the system calls open( ),read( ) and write( ).

ALGORITHM :

Step 1 : Declare the structure elements.
Step 2 : Create a temporary file named temp1.
Step 3 : Open the file named "test" in a write mode.
Step 4 : Enter the strings for the file.
Step 5 : Write those strings in the file named "test".
Step 6 : Create a temporary file named temp2.
Step 7 : Open the file named "test" in a read mode.
Step 8 : Read those strings present in the file "test" and save it in temp2.
Step 9 : Print the strings which are read.

PROGRAM CODING:

```
#include<stdio.h>
#include<unistd.h>
#include<string.h>
#include<fcntl.h>
main( )
{
int fd[2];
char buf1[25]= ”just a test\n”; char
buf2[50];
fd[0]=open(“file1”, O_RDWR);
fd[1]=open(“file2”, O_RDWR);
write(fd[0], buf1, strlen(buf1));
printf(“\n Enter the text now….”);
gets(buf1);
write(fd[0], buf1, strlen(buf1));
lseek(fd[0], SEEK_SET, 0);
read(fd[0], buf2, sizeof(buf1));
write(fd[1], buf2, sizeof(buf2));
close(fd[0]);
close(fd[1]);
printf(“\n”);
return0;
}
```

OUTPUT:

```
Enter the text now….progress
Cat file1 Just a
test progress
Cat file2 Just a test progress
```

Aim：Implement following commands of Unix in C programming Language.

A). cat B). ls C). mv

A) cat

```
#include<sys/types.h>
#include<sys/stat.h>
#include<stdio.h>
#include<fcntl.h>
main( int argc,char *argv[3] )
{
	int fd,i;
	char buf[2];
	fd=open(argv[1],O_RDONLY,0777);
	if(fd==-argc)
	{
		printf("file open error");
	}
	else
```

```
	{
		while((i=read(fd,buf,1))>0)
		{
			printf("%c",buf[0]);
		}
		close(fd);
	}
}
```

Output

```
student@ubuntu:~$gcc –o prgcat.out prgcat.c
student@ubuntu:~$cat > ff
hello spsu
hello udaipur
student@ubuntu:~$./prgcat.out ff
hello spsu
hello udaipur
```

B) ls

```
#include <sys/types.h>
#include <sys/dir.h>
#include <sys/param.h>
#include <stdio.h>
#define FALSE 0
#define TRUE 1
extern  int alphasort();
char pathname[MAXPATHLEN];
main()
{
	int count,i;
	struct dirent **files;
	int file_select();
	if (getwd(pathname) == NULL )
	{
		printf("Error getting pathn");
		exit(0);
	}
	printf("Current Working Directory = %sn",pathname);
	count = scandir(pathname, &files, file_select, alphasort);
	if (count <= 0)
	{
		printf("No files in this directoryn");
		exit(0);
	}
	printf("Number of files = %dn",count);
	for (i=1;i<count 1;  i)
		printf("%s  \n",files[i-1]->d_name);
	}
	int file_select(struct direct   *entry)
	{
		if ((strcmp(entry->d_name, ".") == 0) ||(strcmp(entry->d_name, "..") == 0))
			return (FALSE);
		else
			return (TRUE);
	}
```

Output:

```
Student@ubuntu:~$ gcc list.c
Student@ubuntu:~$ ./a.out
Current working directory=/home/student/
Number of files=57
```

C) mv

```
#include<sys/types.h>
#include<sys/stat.h>
#include<stdio.h>
#include<fcntl.h>
main( int argc,char *argv[] )
{
       int i,fd1,fd2;
       char *file1,*file2,buf[2];
       file1=argv[1];
       file2=argv[2];
       printf("file1=%s file2=%s",file1,file2);
       fd1=open(file1,O_RDONLY,0777);
       fd2=creat(file2,0777);
       while(i=read(fd1,buf,1)>0)
              write(fd2,buf,1);
       remove(file1);
       close(fd1);
       close(fd2);
}
```

Output:

```
student@ubuntu:~$gcc –o mvp.out mvp.c
student@ubuntu:~$cat > ff
hello spsu
hello udaipur
student@ubuntu:~$./mvp.out ff ff1
student@ubuntu:~$cat  ff
cat:ff:No such file or directory
student@ubuntu:~$cat ff1
hello spsu
hello udaipur
```

AIM: Write a C program to emulate the Unix ls-l command.

Program:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
int main()
{
       int pid;            //process id
       pid = fork();     //create another process
       if ( pid < 0 )
       {                                  //fail
              printf("\nFork failed\n");
              exit (-1);
       }
       else if ( pid == 0 )
```

```
            {      //child
                   execlp ( “/bin/ls”, “ls”, “-l”, NULL );  //execute ls
            }
            else
            {      //parent
                   wait (NULL);        //wait for child
                   printf(“\nchild complete\n”);
                   exit (0);
            }
}
```

Output:

```
guest-glcbIs@ubuntu:~$gcc –o lsc.out lsc.c
guest-glcbIs@ubuntu:~$./lsc.out
total 100
-rwxrwx—x 1 guest-glcbls guest-glcbls 140 2012-07-06 14:55 f1
drwxrwxr-x 4 guest-glcbls guest-glcbls 140 2012-07-06 14:40 dir1
child complete
```

AIM: Write a C Program that demonstrates redirection of standard output to a file EX:ls>f1.

Program:

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
main(int argc, char *argv[])
{
       char d[50];
       if(argc==2)
       {
              bzero(d,sizeof(d));
              strcat(d,"ls ");
              strcat(d,"> ");
              strcat(d,argv[1]);
              system(d);
       }
       else
              printf("\nInvalid No. of inputs");
       }
```

output:

```
student@ubuntu:~$ gcc –o std.out std.c
student@ubuntu:~$ls
downloads    documents    listing.c      listing.out          std.c    std.out
student@ubuntu:~$ cat > f1
^z
student@ubuntu:~$./std.out f1
student@ubuntu:~$cat f1
downloads
documents
listing.c
listing.out
std.c
std.out
```

AIM:

Write a C program to create a child process and allow the parent to display "parent" and the child to display "child" on the screen.

Program:

```
#include <stdio.h>
#include <sys/wait.h>  /* contains prototype for wait */
int main(void)
{
	int pid;
	int status;
	printf("Hello World!\n");
	pid = fork( );
	if(pid == -1) /* check for error in fork */
	{
		perror("bad fork");
		exit(1);
	}
	if (pid == 0)
		printf("I am the child process.\n");
	else
	{
		wait(&status); /* parent waits for child to finish */
		printf("I am the parent process.\n");
	}
}
```

Output:

```
student@ubutnu:$gcc –o child.out child.c
student@ubutnu: ./child.out
Hello World!
I am the child process.
I am the parent process
```