BANNARI AMMAN INSTITUTE OF TECHNOLOGY

Autonomous Institution, Accredited by NAAC With 'A' Grade

SPECIAL LABS - SOURCE BOOK

Linux Module 1 – Linux Internals

Revision History:

First Edition – Nov 2022

Prepared By:

Dr P Dhivya

Ms R Mekala

Mr K Balasamy

Dr Sangeetha S N

Dr J Nirmaladevi

Mr M Pandiyan

Dr S Lakshmanaprakash

Mr R Sathishkannan

Ms M Nithya

Ms A Abirami

Incharge **WSTC Coordinator** **Dean PDS**

TRAINING SCHEDULE

Skill Name	**Module 1 Linux Internals**
Day	**Content**
Day 1	Linux Introduction and basic commands
Day 2	Linux Shell scripting & Logical Structures
Day 3	Writing, compiling, and executing a C program on Linux
Day 4	Make files & File Operation
Day 5	Linux Signals and Handling Signals & Process in Linux
Day 6	Linux Scheduler & Memory Management
Day 7	Multithreading and Interthread communication between threads
Day 8	Data sharing between multiple processes using IPC mechanism
Day 9	Linux Networking commands – Developing Client server-based networking application
TP	Test Project and Assessment

Day 01	Linux Introduction and basic commands

Objective:

To impart the knowledge of Linux operating System, evolution, GNU License and installation of Linux in the virtual environment.

Outcomes:

To understand the concepts of Linux OS, Linux installation and implements the basic commands.

Resources Required	**:**	Computer system
Safety Precautions	**:**	Nil
Prerequisites	**:**	

- System Settings
- Installation procedures

Theory:

1. Linux Introduction

Linux is a Unix-like, open source and community-developed operating system (OS) for computers, servers, mainframes, mobile devices and embedded devices. It is supported on almost every major computer platform, including x86, ARM and SPARC, making it one of the most widely supported operating systems.

The kernel is a computer program at the core of a computer's operating system and generally has complete control over everything in the system. It is the portion of the operating system code that is always resident in memory and facilitates interactions between hardware and software components. A full kernel controls all hardware resources (e.g.,. I/O, memory, cryptography) via device drivers, arbitrates conflicts between processes concerning such resources, and optimizes the utilization of common resources e.g. CPU & cache usage, file systems, and network sockets. On most systems, the kernel is one of the first programs loaded on startup (after the bootloader). It handles the rest of startup as well as memory, peripherals, and input/output (I/O) requests from software, translating them into data-processing instructions for the central processing unit.

The GNU General Public License (GNU GPL or simply GPL) is a series of widely used free software licenses that guarantee end users the four freedoms to run, study, share, and modify the software.[7] The license was the first copyleft for general use and was originally written by the founder of the Free Software Foundation (FSF), Richard Stallman, for the GNU Project. The license grants the recipients of a computer program the rights of the Free Software Definition.[8] These GPL series are all copyleft licenses, which means that any derivative woust be distributed under the same or equivalent license terms. It is more restrictive than the Lesser General Public License and even further distinct from the more widely used permissive software licenses BSD, MIT, and Apache.

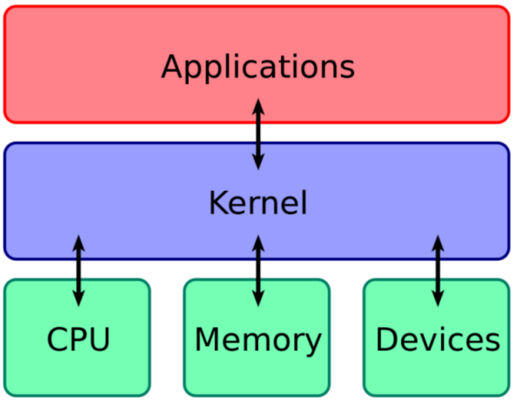


Figure 1.1- Kernel connects application software to the hardware of a computer

2. Linux Installation:

Reference Link : https://www.youtube.com/watch?v=mej9mQHWMKE

3. Linux Commands

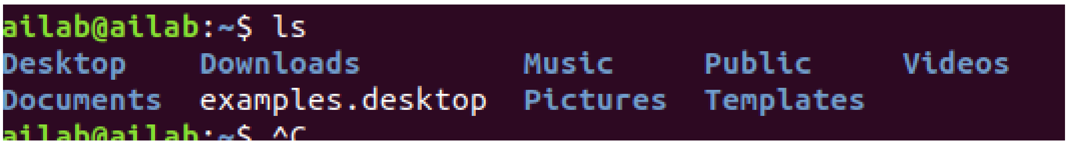
a) List

Getting started with Linux basic commands for directory operations, displaying directory structure in tree format etc.

1) ls

list directory contents

Output:



```
ailab@ailab:~$ ls
Desktop     Downloads         Music     Public     Videos
Documents   examples.desktop  Pictures  Templates
ailab@ailab:~$ ^C
```

SYNOPSIS

ls [OPTION]... [FILE]...

Description:

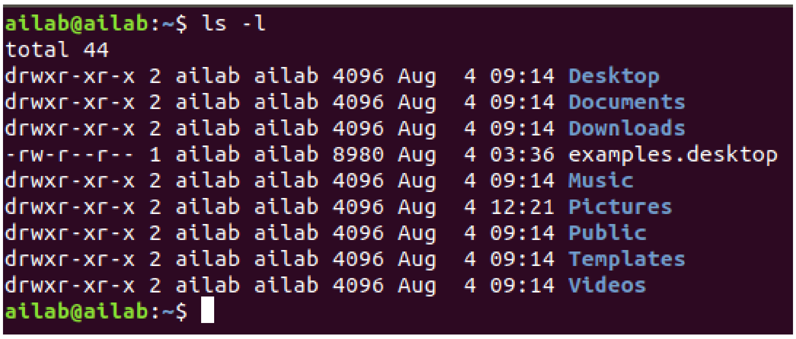
List information about the FILEs (the current directory by default). Sort entries alphabetically if none of -cftuvSUX nor –sort is specified.

Options available for the command

2) ls –l

use a long listing format

Output:

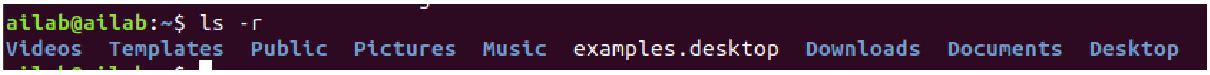


```
ailab@ailab:~$ ls -l
total 44
drwxr-xr-x 2 ailab ailab 4096 Aug  4 09:14 Desktop
drwxr-xr-x 2 ailab ailab 4096 Aug  4 09:14 Documents
drwxr-xr-x 2 ailab ailab 4096 Aug  4 09:14 Downloads
-rw-r--r-- 1 ailab ailab 8980 Aug  4 03:36 examples.desktop
drwxr-xr-x 2 ailab ailab 4096 Aug  4 09:14 Music
drwxr-xr-x 2 ailab ailab 4096 Aug  4 12:21 Pictures
drwxr-xr-x 2 ailab ailab 4096 Aug  4 09:14 Public
drwxr-xr-x 2 ailab ailab 4096 Aug  4 09:14 Templates
drwxr-xr-x 2 ailab ailab 4096 Aug  4 09:14 Videos
ailab@ailab:~$
```

3) ls -r

--reverse - reverse order while sorting

Output:

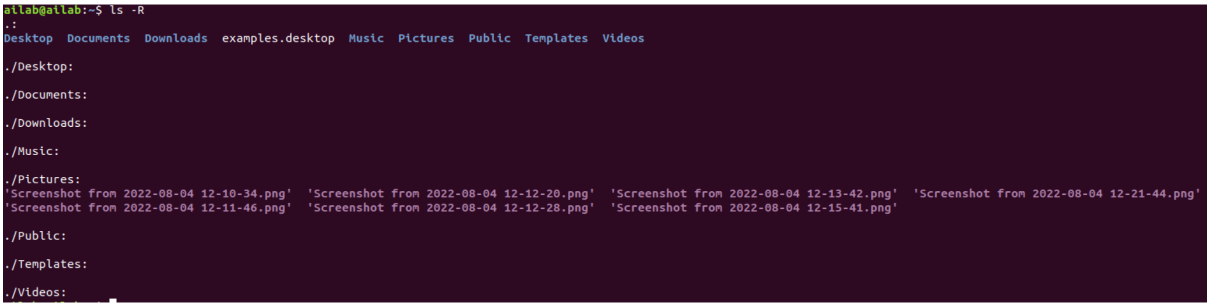


```
ailab@ailab:~$ ls -r
Videos  Templates  Public  Pictures  Music  examples.desktop  Downloads  Documents  Desktop
ailab@ailab:~$
```

4) ls -R

--recursive list subdirectories recursively

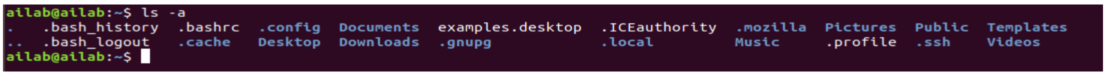
Output:



```
ailab@ailab:~$ ls -R
.:
Desktop  Documents  Downloads  examples.desktop  Music  Pictures  Public  Templates  Videos

./Desktop:

./Documents:

./Downloads:

./Music:

./Pictures:
'Screenshot from 2022-08-04 12-10-34.png'  'Screenshot from 2022-08-04 12-12-20.png'  'Screenshot from 2022-08-04 12-13-42.png'  'Screenshot from 2022-08-04 12-21-44.png'
'Screenshot from 2022-08-04 12-11-46.png'  'Screenshot from 2022-08-04 12-12-28.png'  'Screenshot from 2022-08-04 12-15-41.png'

./Public:

./Templates:

./Videos:
```

5) ls -a

--all - do not ignore entries starting with .



```
ailab@ailab:~$ ls -a
.   .bash_history  .bashrc  .config  Documents  examples.desktop  .ICEauthority  .mozilla  Pictures  Public  Templates
..  .bash_logout   .cache   Desktop  Downloads  .gnupg            .local         Music     .profile  .ssh    Videos
ailab@ailab:~$ 
```

b) clear command

clear

clear the terminal screen

Synopsis

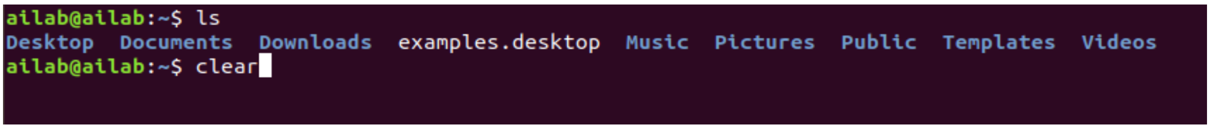
clear

Description

- clear will clears your screen if this is possible, including its scrollback buffer (if the extended "E3" capability is defined).
- clear looks in the environment for the terminal type and then in the terminfo database to determine how to clear the screen.
- clear ignores any command-line parameters that may be present.

Output:

Before executing command:



```
ailab@ailab:~$ ls
Desktop  Documents  Downloads  examples.desktop  Music  Pictures  Public  Templates  Videos
ailab@ailab:~$ clear
```

After executing command:

```
ailab@ailab:~$ 
```

c) Make directory command

mkdir

- make directories

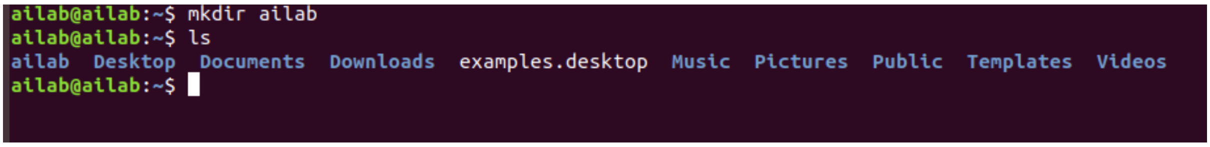
Synopsis

mkdir [OPTION]... DIRECTORY...

Description

- Create the DIRECTORY(ies), if they do not already exist.
- Mandatory arguments to long options are mandatory for short options too.

Output:



```
ailab@ailab:~$ mkdir ailab
ailab@ailab:~$ ls
ailab  Desktop  Documents  Downloads  examples.desktop  Music  Pictures  Public  Templates  Videos
ailab@ailab:~$ 
```

d) Remove directory command

rmdir

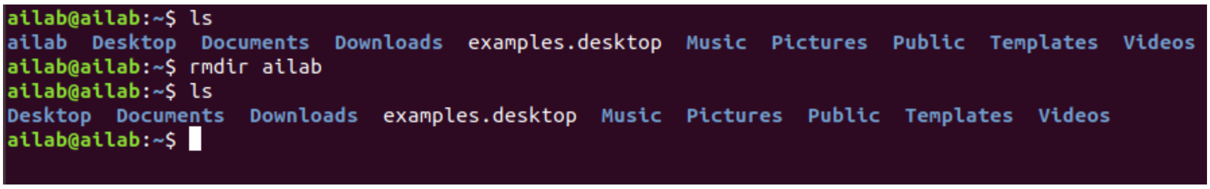
Synopsis

rmdir [OPTION]... DIRECTORY...

Description

Remove the DIRECTORY(ies), if they are empty.

Output:



```
ailab@ailab:~$ ls
ailab  Desktop  Documents  Downloads  examples.desktop  Music  Pictures  Public  Templates  Videos
ailab@ailab:~$ rmdir ailab
ailab@ailab:~$ ls
Desktop  Documents  Downloads  examples.desktop  Music  Pictures  Public  Templates  Videos
ailab@ailab:~$ 
```

e) Remove files or directories

rm

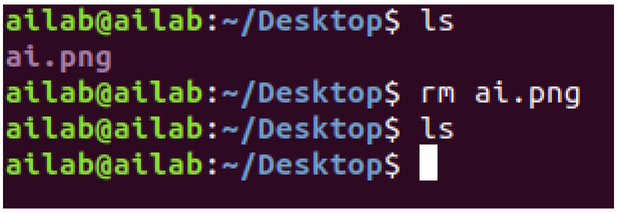
Synopsis

rm [OPTION]... FILE...

Description

This manual page documents the GNU version of rm. rm removes each specified file. By default, it does not remove directories.

Output:



```
ailab@ailab:~/Desktop$ ls
ai.png
ailab@ailab:~/Desktop$ rm ai.png
ailab@ailab:~/Desktop$ ls
ailab@ailab:~/Desktop$ 
```

f) Changing Timestamp

touch

- change file timestamps

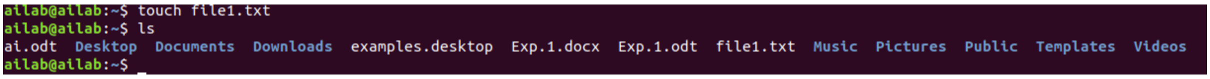
Synopsis

touch [OPTION]... FILE...

Description

- Update the access and modification times of each FILE to the current time.
- A FILE argument that does not exist is created empty, unless -c or -h is supplied.
- A FILE argument string of - is handled specially and causes touch to change the times of the file associated with standard output.

Output:



```
ailab@ailab:~$ touch file1.txt
ailab@ailab:~$ ls
ai.odt  Desktop  Documents  Downloads  examples.desktop  Exp.1.docx  Exp.1.odt  file1.txt  Music  Pictures  Public  Templates  Videos
ailab@ailab:~$ _
```

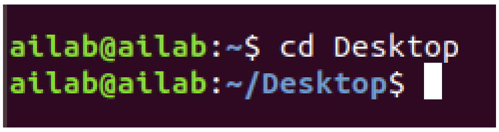
g) Changing the directory

cd <directory name>

cd command will allow you to change directories. When you open a terminal you will be in your home directory.

One of the most used commands of Ubuntu; you can change the directories in the terminal using the “cd” command. For instance, the following command will change the pwd to desktop.

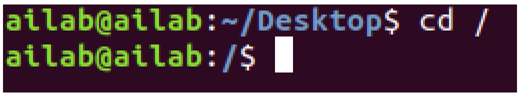
Output:



```
ailab@ailab:~$ cd Desktop
ailab@ailab:~/Desktop$ 
```

There are multiple uses of this command: one can change the present directory to root directory or home directory using this command. When you open a fresh terminal, you are in the home directory.

To change directory to root. For instance, we are in the Desktop directory and want to switch to the root directory:



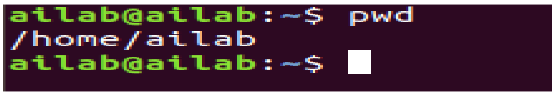
```
ailab@ailab:~/Desktop$ cd /
ailab@ailab:/$ 
```

h) Command to know the present working directory

pwd

-- present working directory

Output:



```
ailab@ailab:~$ pwd
/home/ailab
ailab@ailab:~$ 
```

i) cat command

cat

-- concatenate files and print on the standard output

cat <filename>

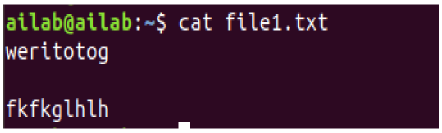
To view file content in terminal

cat > filename concatenates contents typed in terminal to already existing file or newly created file with name "filename"

Description

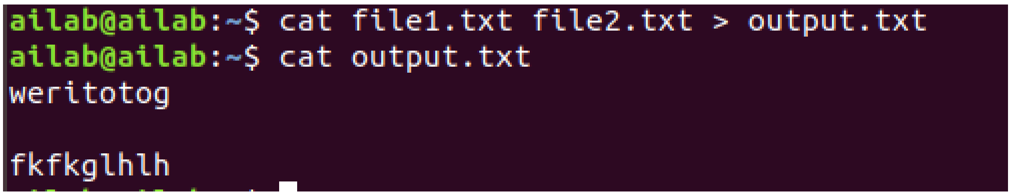
Concatenate FILE(s), or standard input, to standard output.

Output:



```
ailab@ailab:~$ cat file1.txt
weritotog

fkfkglhlh
```

Or you can use this command to save the content of multiples files to one file:



```
ailab@ailab:~$ cat file1.txt file2.txt > output.txt
ailab@ailab:~$ cat output.txt
weritotog

fkfkglhlh
```

j) cp command

cp

-- copy files and directories

Synopsis

cp [OPTION]... [-T] SOURCE DEST

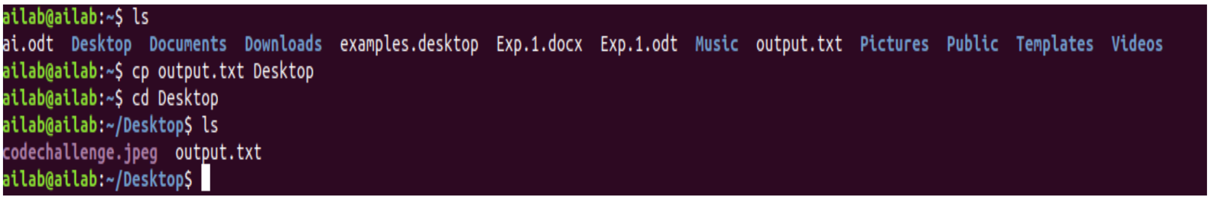
cp [OPTION]... SOURCE... DIRECTORY

cp [OPTION]... -t DIRECTORY SOURCE...

Description

- Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.
- Mandatory arguments to long options are mandatory for short options too.

Output:



```
ailab@ailab:~$ ls
ai.odt  Desktop  Documents  Downloads  examples.desktop  Exp.1.docx  Exp.1.odt  Music  output.txt  Pictures  Public  Templates  Videos
ailab@ailab:~$ cp output.txt Desktop
ailab@ailab:~$ cd Desktop
ailab@ailab:~/Desktop$ ls
codechallenge.jpeg  output.txt
ailab@ailab:~/Desktop$
```

k) Renaming the files

rename

-- rename files

Synopsis

rename [options] expression replacement file...

Description

rename will rename the specified files by replacing the first occurrence of expression in their name by replacement.

Options

-s, --symlink

Do not rename a symlink but its target.

-v, --verbose

Show which files where renamed, if any.

-n, --no-act

Do not make any changes.

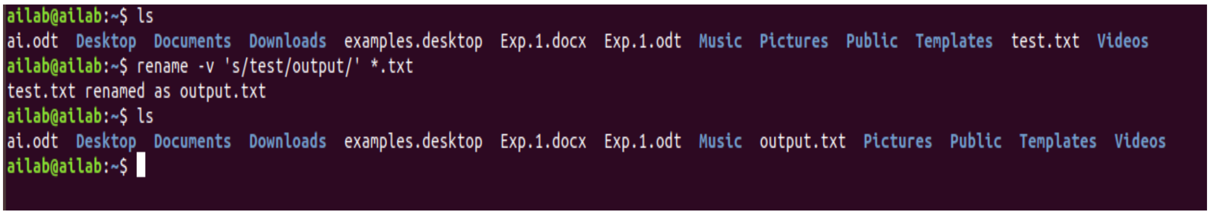
-V, --version

Display version information and exit.

-h, --help

Display help text and exit.

Output:



```
ailab@ailab:~$ ls
ai.odt  Desktop  Documents  Downloads  examples.desktop  Exp.1.docx  Exp.1.odt  Music  Pictures  Public  Templates  test.txt  Videos
ailab@ailab:~$ rename -v 's/test/output/' *.txt
test.txt renamed as output.txt
ailab@ailab:~$ ls
ai.odt  Desktop  Documents  Downloads  examples.desktop  Exp.1.docx  Exp.1.odt  Music  output.txt  Pictures  Public  Templates  Videos
ailab@ailab:~$ 
```

Schedule:

Day / Time	8:45 am to 9:35am	9:35 am to 10:25 am	10:25 am to 10:40 am	10:40 am to 11:30 am	11:30 am to 12:20 pm	12:20 pm to 1:30pm	1:30 pm to 2:20pm	2:20 pm to 3:10pm	3:10 pm to 3:25pm	3:25 pm to 4:15pm
01	Introduction-to Linux		**Tea Break**	Implementation of Basic Linux Commands		**Lunch Break**	Implementation of basic commands		**Tea Break**	Task Assessment

Description of the task:

Task1: Installation of Linux in virtual environment

Task 2: Execution of basic commands in Linux

Assessment Pattern:

Sl. No.	**Aspect Type (M or J)**	**Aspect of description**	**Extra aspect of description**	**Requirement (only for M)**	**Max. marks**
1	J	List files and directories	Yes - **1** No - **0**	-	1
2	J	Sorting the files	Yes - **1** No - **0**	-	1
	J	Clear the entries	Yes - **1** No - **0**	-	1
	J	Make directories	Yes - **1** No - **0**	-	1
	J	Remove files	Yes - **1** No - **0**	-	1
	J	Remove directories	Yes - **1** No - **0**	-	1
	J	Change the directories	Yes - **1** No - **0**	-	1
	J	Copy files	Yes - **1** No - **0**	-	1
	J	Copy directories	Yes - **1** No - **0**	-	1
	J	Rename files	Yes - **1** No - **0**	-	1
Total Marks					**10**

Day 02	Linux Shell scripting & Logical Structures

Objective:

To understand the concepts of basic shell scripting using variable, operators, command line arguments and control statements.

Outcomes:

To implement the shell scripting using variables, operators, CLA, looping and control statements.

Resources Required	**:**	Linux Virtual environment
Safety Precautions	**:**	Nil

Prerequisites:

- Basic shell commands
- Dir & File Commands
- System Commands
- Misc Commands

Theory:

Reference Link:

Shell Scripting is an open-source computer program designed to be run by the Unix/Linux shell. Shell Scripting is a program to write a series of commands for the shell to execute. It can combine lengthy and repetitive sequences of commands into a single and simple script that can be stored and executed anytime which, reduces programming efforts.

Shell is a UNIX term for an interface between a user and an operating system service. Shell provides users with an interface and accepts human-readable commands into the system and executes those commands which can run automatically and give the program's output in a shell script.

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

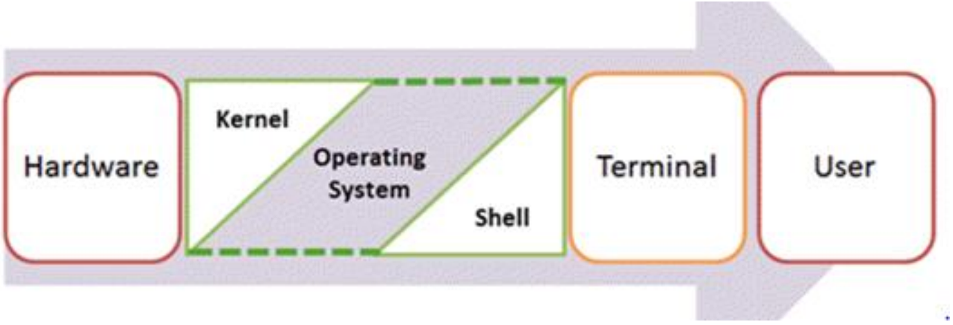


Figure 2.1- Components of Shell Program

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. A shell in a Linux operating system takes input from you in the form of commands, processes it, and then gives an output. It is the interface through which a user works on the programs, commands, and scripts. A shell is accessed by a terminal which runs it.

When you run 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.

Types of Shell

There are two main shells in Linux:

1. The Bourne Shell: The prompt for this shell is $ and its derivatives are listed below:

- POSIX shell also is known as sh
- Korn Shell also knew as sh
- Bourne Again SHell also knew as bash (most popular)

2. The C shell: The prompt for this shell is %, and its subcategories are:

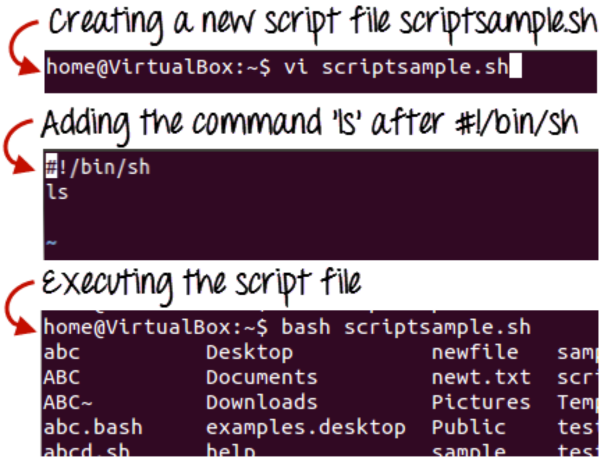
- C shell also is known as csh
- Tops C shell also is known as tcsh

Write Shell Script in Linux/Unix

Shell Scripts are written using text editors. On your Linux system, open a text editor program, open a new file to begin typing a shell script or shell programming, then give the shell permission to execute your shell script and put your script at the location from where the shell can find it.

Steps to write a shell script

- Create a file using a vi editor(or any other editor). Name script file with extension .sh
- Start the script with #! /bin/sh
- Write some code.
- Save the script file as filename.sh
- For executing the script type bash filename.sh
- “#!” is an operator called shebang which directs the script to the interpreter location. we use”#! /bin/sh” the script gets directed to the bourne-shell.
  - #!/bin/sh
  - ls



Creating a new script file scriptsample.sh

```
home@VirtualBox:~$ vi scriptsample.sh
```

Adding the command 'ls' after #!/bin/sh

```
#!/bin/sh
ls

~
```

Executing the script file

```
home@VirtualBox:~$ bash scriptsample.sh
abc          Desktop          newfile    sam
ABC          Documents        newt.txt   scr
ABC~         Downloads        Pictures   Tem
abc.bash     examples.desktop Public     tes
abcd.sh      help             sample     tes
```

Figure 2.2- Steps to Create Shell Script in Linux/Unix

Adding shell comments

Commenting is important in any program. In Shell programming, the syntax to add a comment is

#comment

Shell Variables

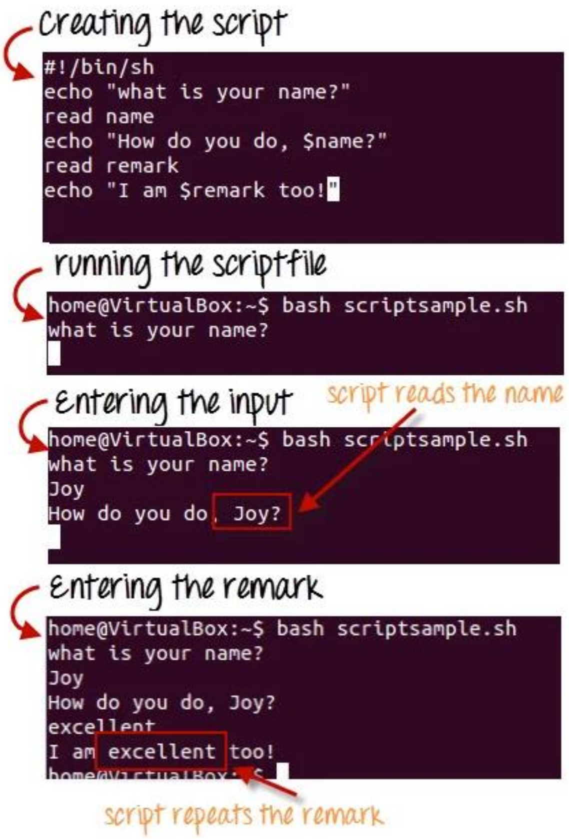
Variables store data in the form of characters and numbers. Similarly, Shell variables are used to store information and they can by the shell only.

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

```
variable ="Hello"

echo $variable
```

Example:

```
#!/bin/sh

echo "what is your name?"

read name

echo "How do you do, $name?"

read remark

echo "I am $remark too!"
```



Creating the script

```
#!/bin/sh
echo "what is your name?"
read name
echo "How do you do, $name?"
read remark
echo "I am $remark too!"
```

running the scriptfile

```
home@VirtualBox:~$ bash scriptsample.sh
what is your name?
```

Entering the input

script reads the name

```
home@VirtualBox:~$ bash scriptsample.sh
what is your name?
Joy
How do you do, Joy?
```

Entering the remark

```
home@VirtualBox:~$ bash scriptsample.sh
what is your name?
Joy
How do you do, Joy?
excellent
I am excellent too!
home@VirtualBox:~$
```

script repeats the remark

Figure 2.3- Steps to create and execute the script

I. BASIC OPERATORS IN SHELL SCRIPTING

There are 5 basic operators in bash/shell scripting:

1. Arithmetic Operators
2. Relational Operators
3. Boolean Operators
4. Bitwise Operators
5. File Test Operators

1. Arithmetic Operators:

These operators are used to perform normal arithmetic /mathematical operations. There are 7 arithmetic operators:

- Addition (+): Binary operation used to add two operands.
- Subtraction (-): Binary operation used to subtract two operands.
- Multiplication (*): Binary operation used to multiply two operands.
- Division (/): Binary operation used to divide two operands.
- Modulus (%): Binary operation used to find remainder of two operands.
- Increment Operator (++): Unary operator used to increase the value of operand by one.
- Decrement Operator (- -): Unary operator used to decrease the value of a operand by one

Program:

```
#!/bin/bash

#reading data from the user

read - p 'Enter a : ' a

read - p 'Enter b : ' b

add = $((a + b))

echo Addition of a and b are $add
```

sub = $((a - b))

echo Subtraction of a and b are $sub

mul = $((a * b))

echo Multiplication of a and b are $mul

div = $((a / b))

echo division of a and b are $div

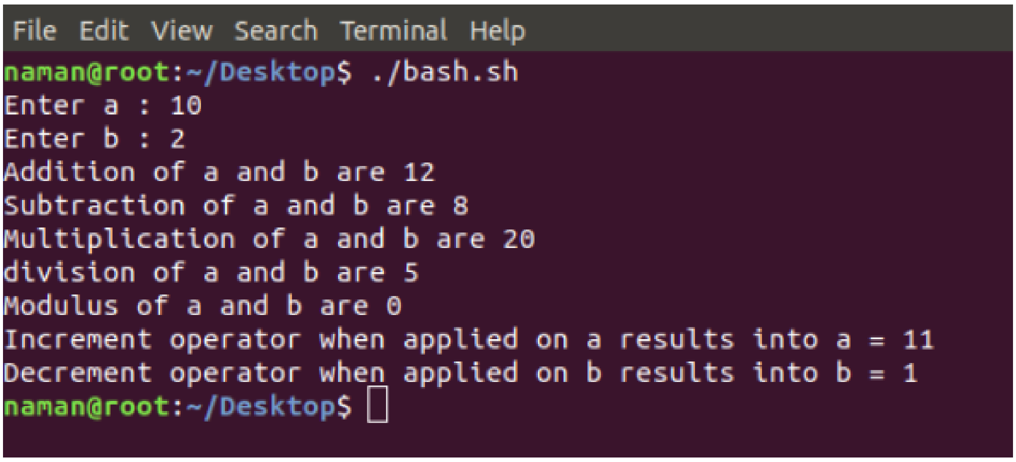
mod = $((a % b))

echo Modulus of a and b are $mod((++a))

echo Increment operator when applied on "a" results into a = $a((--b))

echo Decrement operator when applied on "b" results into b = $b

Output:



```
naman@root:~/Desktop$ ./bash.sh
Enter a : 10
Enter b : 2
Addition of a and b are 12
Subtraction of a and b are 8
Multiplication of a and b are 20
division of a and b are 5
Modulus of a and b are 0
Increment operator when applied on a results into a = 11
Decrement operator when applied on b results into b = 1
naman@root:~/Desktop$
```

Figure 2.4- Sample Output

2. Relational Operators:

Relational operators are those operators which define the relation between two operands. They give either true or false depending upon the relation.

They are of 6 types:

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

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

‘<‘ Operator: Less than operator returns true if first operand is less than second operand otherwise returns false.

‘<=’ Operator: Less than or equal to operator returns true if first operand is less than or equal to second operand otherwise returns false

‘>’ Operator: Greater than operator return true if the first operand is greater than the second operand otherwise return false.

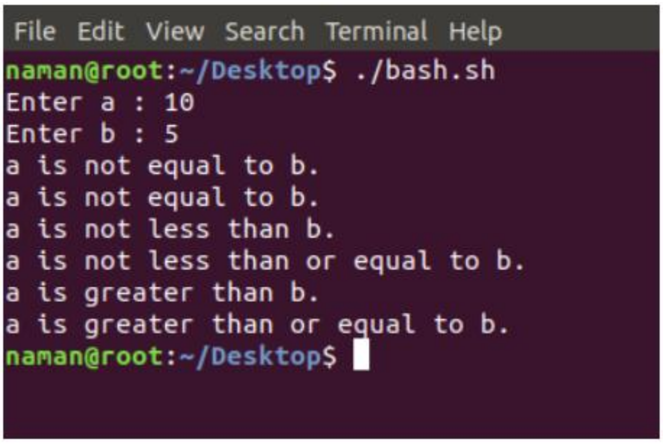
‘>=’ Operator: Greater than or equal to operator returns true if first operand is greater than or equal to second operand otherwise returns false

Program:

```
#!/bin/bash

#reading data from the user

read -p 'Enter a : ' a

read -p 'Enter b : ' b

if(( $a==$b ))

then

   echo a is equal to b.

else

   echo a is not equal to b.

fi

if(( $a!=$b ))

then

   echo a is not equal to b.

else

   echo a is equal to b.

fi

if(( $a<$b ))

then
```

```
    echo a is less than b.
else
    echo a is not less than b.
fi
if(( $a<=$b ))
then
    echo a is less than or equal to b.
else
    echo a is not less than or equal to b.
fi
if(( $a>$b ))
then
    echo a is greater than b.
else
    echo a is not greater than b.
fi
if(( $a>=$b ))
then
    echo a is greater than or equal to b.
else
    echo a is not greater than or equal to b.
fi
```

Output:



```
naman@root:~/Desktop$ ./bash.sh
Enter a : 10
Enter b : 5
a is not equal to b.
a is not equal to b.
a is not less than b.
a is not less than or equal to b.
a is greater than b.
a is greater than or equal to b.
naman@root:~/Desktop$
```

Figure 2.5- Sample Output

3. Logical Operators:

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

Logical AND (&&):

This is a binary operator, which returns true if both the operands are true otherwise returns false.

Logical OR (||):

This is a binary operator, which returns true is either of the operand is true or both the operands are true and return false if none of then is false.

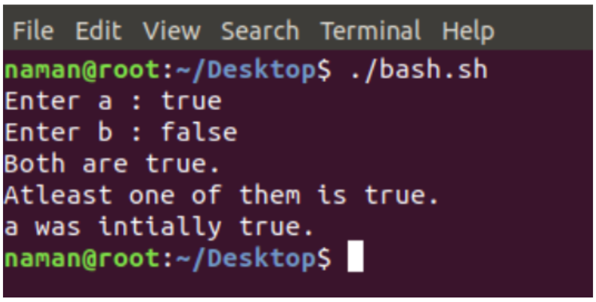
Not Equal to (!):

This is a unary operator which returns true if the operand is false and returns false if the operand is true.

Program:

```
#!/bin/bash

#reading data from the user

read -p 'Enter a : ' a

read -p 'Enter b : ' b

if(($a == "true" & $b == "true" ))
```

```
then
    echo Both are true.
else
    echo Both are not true.
fi
if(($a == "true" || $b == "true" ))
then
    echo Atleast one of them is true.
else
    echo None of them is true.
fi
 if(( ! $a == "true"  ))
then
    echo "a" was initially false.
else
     echo "a" was initially true.
 fi
```



```
naman@root:~/Desktop$ ./bash.sh
Enter a : true
Enter b : false
Both are true.
Atleast one of them is true.
a was intially true.
naman@root:~/Desktop$ 
```

***Figure 2.6* - **Sample Output

4. Bitwise Operators:

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

Bitwise And (&):

Bitwise & operator performs binary AND operation bit by bit on the operands.

Bitwise OR (|):

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

Bitwise XOR (^):

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

Bitwise complement (~):

Bitwise ~ operator performs binary NOT operation bit by bit on the operand.

Left Shift (<<):

This operator shifts the bits of the left operand to left by number of times specified by right operand.

Right Shift (>>):

This operator shifts the bits of the left operand to right by number of times specified by right operand.

Program:

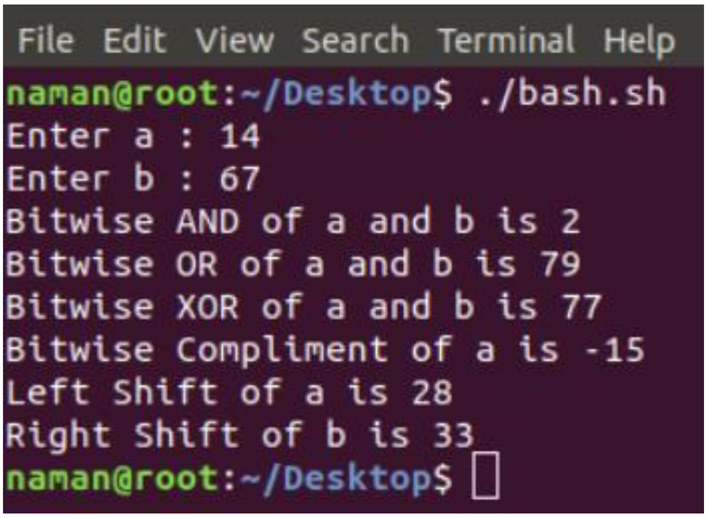
```
#!/bin/bash

#reading data from the user

read -p 'Enter a : ' a

read -p 'Enter b : ' b

bitwiseAND=$(( a&b ))

echo Bitwise AND of a and b is $bitwiseAND

bitwiseOR=$(( a|b ))

echo Bitwise OR of a and b is $bitwiseOR

bitwiseXOR=$(( a^b ))

echo Bitwise XOR of a and b is $bitwiseXOR

bitiwiseComplement=$(( ~a ))

echo Bitwise Compliment of a is $bitiwiseComplement
```

leftshift=$((a<<1))

echo Left Shift of a is $leftshift

rightshift=$((b>>1))

echo Right Shift of b is $rightshift



```
naman@root:~/Desktop$ ./bash.sh
Enter a : 14
Enter b : 67
Bitwise AND of a and b is 2
Bitwise OR of a and b is 79
Bitwise XOR of a and b is 77
Bitwise Compliment of a is -15
Left Shift of a is 28
Right Shift of b is 33
naman@root:~/Desktop$
```

Figure 2.7 **-** Sample Output

5. File Test Operator:

These operators are used to test a particular property of a file.

-b operator:

This operator check whether a file is a block special file or not. It returns true if the file is a block special file otherwise false.

-c operator:

This operator checks whether a file is a character special file or not. It returns true if it is a character special file otherwise false.

-d operator:

This operator checks if the given directory exists or not. If it exists then operators return true otherwise false.

-e operator:

This operator checks whether the given file exists or not. If it exits this operator returns true otherwise false.

-r operator:

This operator checks whether the given file has read access or not. If it has read access then it returns true otherwise false.

-w operator:

This operator checks whether the given file has written access or not. If it has written then it returns true otherwise false.

-x operator:

This operator checks whether the given file has executed access or not. If it has executed access then it returns true otherwise false.

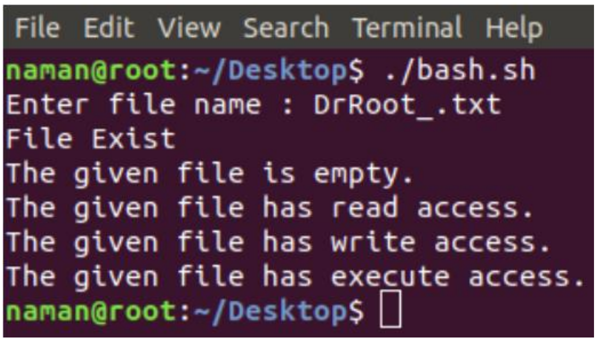
-s operator:

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

Program:

```
#!/bin/bash

#reading data from the user

read -p 'Enter file name : ' FileName

if [ -e $FileName ]

then

   echo File Exist

else

   echo File doesnot exist

fi

if [ -s $FileName ]

then

   echo The given file is not empty.

else

   echo The given file is empty.
```

```
fi
if [ -r $FileName ]
then
   echo The given file has read access.
else
   echo The given file does not has read access.
fi
if [ -w $FileName ]
then
   echo The given file has write access.
else
   echo The given file does not has write access.
fi
if [ -x $FileName ]
then
   echo The given file has execute access.
else
   echo The given file does not has execute access.
fi
```



```
naman@root:~/Desktop$ ./bash.sh
Enter file name : DrRoot_.txt
File Exist
The given file is empty.
The given file has read access.
The given file has write access.
The given file has execute access.
naman@root:~/Desktop$
```

Figure 2.8 **-** Sample Output

II. COMMAND LINE ARGUMENTS IN UNIX SHELL SCRIPT

The UNIX shell is used to run commands, and it allows users to pass run time arguments to these commands. These arguments, also known as command line parameters that allows the users to either control the flow of the command or to specify the input data for the command.

```
$ sh hello how to do you do
```

Here $0 would be assigned sh

$1 would be assigned hello

$2 would be assigned how

1. set

```
$ set how do you do
$ echo $1 $2
how do
```

2. shift

```
$ set hello good morning how do you do welcome to UNIX tutorial.
```

Here, ‘hello’ is assigned to $1, ‘good’ to $2 and so on to ‘to’ being assigned to $9. Now the shift command can be used to shift the parameters ‘N’ places.

Code:

```
$ shift 2
$ echo $1
```

Now $1 will be ‘morning’ and so on to $8 being ‘unix’ and $9 being ‘tutorial’.

III. Decision making statements:

Unix Shell supports conditional statements which are used to perform different actions based on different conditions.

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

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

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

Program:

```
#!/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
```

Output:

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

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.

Program:

```
#!/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
```

Output:

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

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.

Program:

```
#!/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
```

Output:

```
a is less than b
```

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.

Program:

```
#!/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
```

Output:

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

IV. LOOPING STRUCTURES

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.

1. The while loop
2. The for loop
3. The until loop
4. The select loop

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.

1. while Loops

Program 1:

```
#!/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.

Output:

```
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
```

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

Program 2:

```
#!/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
```

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.

Program 3:

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

Output:

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

8

9

2. 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.

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.

Program:

```
#!/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

$
```

V. SHELL LOOP CONTROL

All the loops have a limited life and they come out once the condition is false or true depending on the loop.A loop may continue forever if the required condition is not met. A loop that executes forever without terminating executes for an infinite number of times. For this reason, such loops are called infinite loops.

Program:

```
#!/bin/sh

a=10

until [ $a -lt 10 ]

do

        echo $a

        a=`expr $a + 1`

done
```

1. The break Statement

The break statement is used to terminate the execution of the entire loop, after completing the execution of all of the lines of code up to the break statement. It then steps down to the code following the end of the loop.

Program:

```
#!/bin/sh
a=0
while [ $a -lt 10 ]
do
        echo $a
        if [ $a -eq 5 ]
        then
                break
        fi
        a=`expr $a + 1`
done
```

Upon execution, you will receive the following result −

```
0
1
2
3
4
5
```

2. The continue statement

The continue statement is similar to the break command, except that it causes the current iteration of the loop to exit, rather than the entire loop. This statement is useful when an error has occurred but you want to try to execute the next iteration of the loop.

Program:

```
#!/bin/sh

NUMS="1 2 3 4 5 6 7"

for NUM in $NUMS

do

        Q=`expr $NUM % 2`

        if [ $Q -eq 0 ]

        then

                echo "Number is an even number!!"

                continue

        fi

        echo "Found odd number"

done
```

Output:

```
Found odd number

Number is an even number!!

Found odd number

Number is an even number!!

Found odd number

Number is an even number!!

Found odd number
```

Schedule:

Day/ Time	8:45 am to 9:35 am	9:35am to 10:25 am	10:25 am to 10:40am	10:40 am to 11:30 am	11:30 am to 12:20 pm	12:20 pm to 1:30 pm	1:30 pm to 2:20 pm	2:20 pm to 3:10 pm	3:10 pm to 3:25 pm	3:25 pm to 4:15 pm
02	Basic shell scripting using variable		**Tea Break**	Operators in Shell scripting		**Lunch Break**	Control statements		**Tea Break**	Task Asses -sment

Description of the task:

Task 1: Write a program to perform the addition, subtraction, multiplication and division.

Task 2: Write a Shell script to find factorial of a given integer.

Task 3: Write a script to find sum of total salary of all employees.

Task 4: Write a program to select DRINK in tea, coffee, water, juice and apple.

Task 5: Write a program to verify whether the given two numbers is equal or not.

Assessment Pattern:

Sl. No.	**Aspect Type (M or J)**	**Aspect of description**	**Extra aspect of description**	**Requirement (only for M)**	**Max. marks**
1.	J	Creation of new script file	Yes **- 1** No **- 0**	-	1
2.	J	Implementation of Arithmetic Operators	Yes **- 1** No **- 0**	-	1
3.	J	Implementation of Relational Operators	Yes **- 1** No **- 0**	-	1
4.	J	Implementation of Logical and Bitwise Operators	If Yes**, then** a. Logical operators are used- **0.5** b. Logical operators is used- **0.5** No **- 0**	-	1

5.	J	Shows the result of Command Line Arguments	Yes **- 1** No **- 0**	-	1
6.	J	Implementation of Shell script to find factorial of a given integer	Yes **- 1** No **- 0**	-	1
7.	J	Use of if...elif...else...fi statement	Yes **- 1** No **- 0**	-	1
8.	J	Implementation of looping • While • For • Until • select	If yes, then a. While loop is used – **0.25** b. for loop is used – **0.25** c. Until loop is used – **0.25** d. Select loop is used – **0.25** No **- 0**	-	1
9.	J	Implementation of shell loop control.	If yes, then a. The break statement is used – **0.5** b. The continue statement is used – **0.5** No - 0	-	1
10.	J	Script to find sum of total salary of all employees	Yes **- 1** No **- 0**	-	1
Total Marks					**10**

Day 03	Writing, compiling, and executing a C program on Linux

Objective

The aim of the task is to write, compile and execute a C program on LINUX.

Outcome

Students will be able to apply the C programming concepts in Linux Platform.

Resources Required	**:**	LINUX Platform
Safety Precautions	**:**	Nil
Prerequisites	**:**	Knowledge about C- Programming Language

Theory:

C Programming language is a well-known programming language because of its rich library of functions. C program does not execute without a compiler in Linux. Therefore, a dedicated compiler is needed to compile programming languages in Linux distribution. In this session, we will learn what C programming is and how it is used to compile C programs in Linux.

Modes of Text Editor:

https://docs.google.com/document/d/1uJsjUUNTxABJ-O7_xcA-H8NEUhhwK72BfeSvkohe7rI/edit?usp=sharing

https://www.javatpoint.com/linux-text-editors#:~:text=There%20are%20two%20types%20of,)%2C%20Kwrite%2C%20and%20more

Writing a C program in Linux using vi editor:
https://docs.google.com/document/d/1r8PT8oh3OjBdJcWzNGKj7GgZV09xcETEtkUMLYSz-rY/edit?usp=sharing

https://www.scaler.com/topics/c/how-to-compile-c-program-in-linux/

https://vitux.com/how-to-write-and-run-a-c-program-in-linux/

Regular steps

1. Connect to Linux server using telnet 172.16.22.5

2. Enter your login and passwd.

3. Type ls to verify your directory name.

4. To enter your first-year directory cd directory name.

5. vi filename.c - opens the editor with the name given.

6. Esc i for typing source code. Save it with Esc: wq

7. cc filename.c (or) gcc filename.c- to compile the file

8. a.out (or) ./a.out (or) to get the output.

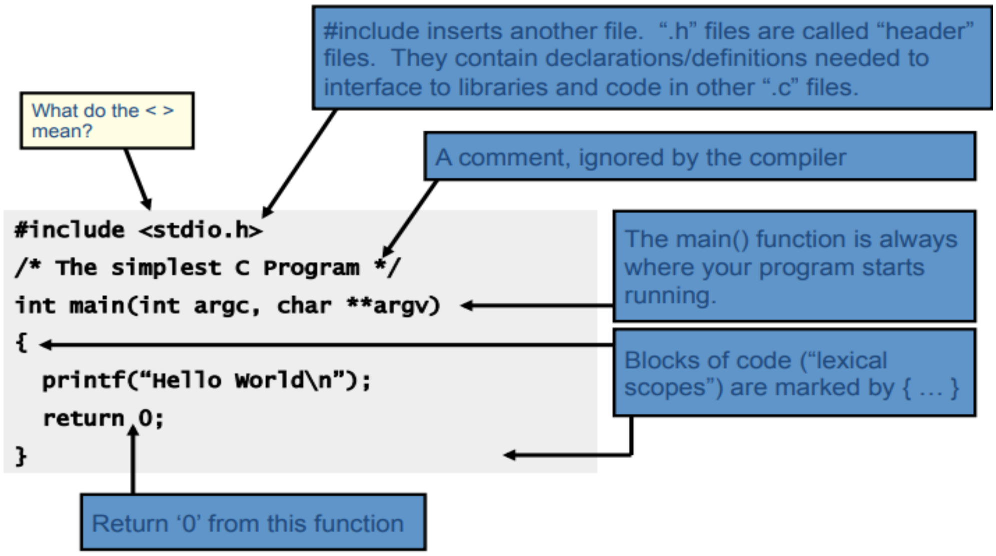
C Programs:

Program 1: Print Hello World

In the command mode, open the editor file with name hello.c by using the following command.

$vi hello.c

Type the following program by pressing ‘i’ to change into the insert mode



```
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
  printf(“Hello World\n”);
  return 0;
}
```

#include inserts another file. “.h” files are called “header” files. They contain declarations/definitions needed to interface to libraries and code in other “.c” files.

What do the < > mean?

A comment, ignored by the compiler

The main() function is always where your program starts running.

Blocks of code (“lexical scopes”) are marked by { … }

Return ‘0’ from this function

To save and quit : Press Esc: wq

To compile : $ cc hello.c

To Execute : $ ./a.out

Hello World

Program 2: Read user input

```
#include <stdio.h>

int main()

{

        char name[100];

        int age;

        printf("Enter your name: ");

        scanf("%s",name);

        printf("Enter your age: ");

        scanf("%d",&age);

        printf("Hello, %s,You are %d years old", name, age);

}
```

Output:

```
Enter your name: Riyaan

Enter your age: 23

Hello, Ryanoid are 23 years old
```

Program 3: Read command-line arguments

```
#include <stdio.h>
int main(int argc,char* argv[])

{

        printf("Total number of arguments = %d\n",argc);
        printf("Argument No. 1 = %s\n",argv[0]);

        printf("Argument No. 2 = %s\n",argv[1]);
        printf("Argument No. 3 = %s\n",argv[2]);

}
```

Output:

```
Total number of arguments = 1
```

```
Argument No. 1 = /tmp/uawjnLp746.o

Argument No. 2 = (null)

Argument No. 3 = KUBERNETES_SERVICE_PORT=443
```

Program 4: Compare string using conditional statements

```
#include <stdio.h>
#include <string.h>
int main()
{
        int n1, n2, result;
        char operator[10];
        printf("Enter first number :");
        scanf("%d",&n1);
        printf("Enter second number :");
        scanf("%d",&n2);
        printf("Enter operation name :");
        scanf("%s",operator);
        if(strcmp(operator,"add") == 0)
                result = n1 + n2;
        else if(strcmp(operator,"sub") == 0)
                result = n1 - n2;
        else
                result=0;
                printf("The result is : %d\n",result);
}
```

Output:

```
Enter first number :100

Enter second number :150

Enter operation name: add
```

```
The result is: 250
```

Program 5: Iterate a list of string using for loop

```
#include <stdio.h>
int main()
{
        char flowers[10][20] = {"Rose", "Poppy", "Lily", "Tulip", "Marigold"};
        int total=sizeof(flowers)/sizeof(flowers[0]);
        for (int n = 0; n <total; n++)
        {
                printf("%s\n",flowers[n]);
        }
}
```

Output:

```
Rose

Poppy

Lily

Tulip

Marigold
```

Program 6: Find even numbers from a list using while loop

```
#include <stdio.h>
int main(){
int numbers[10] = { 21, 78, 62, 90, 55, 10, 85, 45 };
int i = 0;
printf("The even numbers from the list are:\n");
while(i < 10)
{
        if((numbers[i] % 2) == 0)

                printf("%d\n", numbers[i]);
                i++;

        }
}
```

Output:

```
The even numbers from the list are:

78

62

90

10

0

0
```

Program 7: Find out the area of a rectangle using the function

```
#include <stdio.h>
int area(int h, int w);
int area(int h, int w)
{

        int area = h * w;

        return area;

}
int main()
{

        int height, width;
        printf("Enter the height of the rectangle:");
        scanf("%d", &height);
        printf("Enter the width of the rectangle:");
        scanf("%d", &width);
        printf("The area of the rectangle = %d\n",area(height,width));
}
```

Output:

```
Enter the height of the rectangle:5
Enter the width of the rectangle:3
The area of the rectangle = 15
```

Schedule:

Day/ Time	8:45 am to 9:35 am	9:35 am to 10:25 am	10:25 am to 10:40 am	10:40 am to 11:30 am	11:30 am to 12:20 pm	12:20 pm to 1:30 pm	1:30 pm to 2:20 pm	2:20 pm to 3:10 pm	3:10 pm to 3:25 pm	3:25 pm to 4:15 pm
03	Introduction to Vi Editor & Executing a simple program		**Tea Break**	Executing C programs includes data types, variables, loop, operator, arrays, strings		**Lunch Break**	Executing C programs includes data types, variables, loop, operator, arrays, strings		**Tea Break**	**Task**

Description of the task:

Task 1: Write a C program to take a number as age value of a person and print the person is a teenager or young or old.

Task 2: Write a C program to find out a particular string in a list.

Task 3: Write a C Program using the function to calculate the area of trapezium.

Assessment Pattern:

Sl. No.	**Aspect Type (M or J)**	**Aspect of description**	**Extra aspect of description**	**Requirement (only for M)**	**Max. marks**
1.	J	Installing the vi editor	Installing the vi Editor Yes **- 1** No **- 0**	-	1
2.	J	Checking the version of the vi editor	Checking the version of the vi editor Yes **- 1** No **- 0**	-	1
3.	J	**Open the vi editor**	If able to open the vi editor Yes **- 1** No **- 0**	**-**	1

4.	J	Accessing the insert mode to type the text into the file	If accessing the insert mode to type the text into the file Yes **- 1** No **- 0**	**-**	1
5.	J	Save the file with .c extension	If saving the file with .c extension Yes **- 1** No **- 0**	**-**	1
6.	J	Compile the program	If compiling the c program Yes **- 1** No **- 0**	**-**	1
7.	J	Execute the output of the program	If executing the output of the program Yes **- 1** No **- 0**	-	1
8.	J	If error occurs, again open the vi file and edit	If error occurs, again open the vi file and edit Yes **- 1** No **- 0**	-	1
9.	J	Time Management	**0 -** Exceeded 45 mins **0.5 -** Completed within 30 to 45 mins **1 -** Completed within 30 min	-	1
10.	J	**Coding Ethics** Proper Indentation Overall design look	**0 -** If none of the aspect is found **0.5 -** If exhibiting any one aspect **1 -** If exhibiting all two aspects	-	1
Total arks					**10**

Day 04	Make Files and File Operation

Objective:

- To understand the concepts of make files and its rules and dependencies.
- To create applications to access files in Linux.
- System Calls used in Linux to control special files like device nodes

Outcomes:

- To implement make file using variables, dependencies and rules.
- To understand the Basic File related commands, and create files with various access permissions.

Resources Required : Linux Virtual environment

Safety Precautions : Nil

Prerequisites:

- Basic shell commands
- Dir & File Commands
- System Commands
- Misc Commands
- Basics of Operating Systems.
- Basic Linux Commands.
- File commands

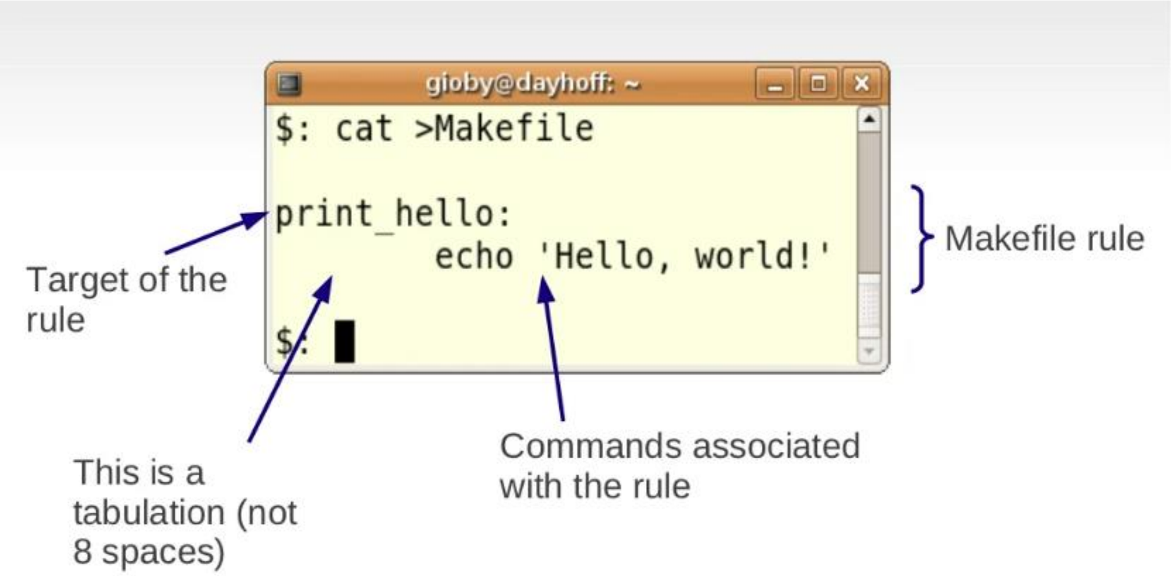
Theory:

I. MAKE FILES

Compiling your source code files can be tedious, especially when you want to include several source files and have to type the compiling command every time you want to do it. Make files are special format files that together with the make utility will help you to automatically build and manage your projects.

The make utility is a software tool for managing and maintaining computer programs consisting many component files. The make utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them.

- Make reads its instruction from Make file (called the descriptor file) by default.
- Make file sets a set of rules to determine which parts of a program need to be recompile, and issues command to recompile them.
- Make file is a way of automating software building procedure and other complex tasks with dependencies.
- Make file contains: dependency rules, macros and suffix (or implicit) rules.



```
$: cat >Makefile

print_hello:
	echo 'Hello, world!'

$: 
```

gioby@dayhoff: ~
Target of the rule
Makefile rule
This is a tabulation (not 8 spaces)
Commands associated with the rule

1. The make utility:

To invoke make, simply type:

Make

This program will look for a file named Make file in your directory and execute it.

If you have several make files, then you can execute them with the command:

make -f MyMakefile

2. Make file rules:

A make file primarily consisting of rules formatted like this:

target: dependencies [tab] system command

3. The basic Make file:

The trivial way to compile the files and obtain an executable, is by running the command:

```
        g++ main.cpp hello.cpp factorial.cpp -o hello
```

To automatically execute this command, you can create a simple Makefile with these contents:

```
all:
        g++ main.cpp hello.cpp factorial.cpp -o hello
```

On this first example we see that our target is called all. This is the default target for make files. The make utility will execute this target if no other one is specified. We also see that there are no dependencies for target all, so make will execute the specified system command(s). In this case, the command compiles the program according to the command line we gave it.

4. Using Dependencies:

For larger projects, it can be helpful to use different targets. This is because if you modify a single file in your project, you don't have to recompile everything, only what you modified.

To accomplish this, it requires breaking the build process into two steps:

a. Compilation of source code files into an object file.

b. Linking the object files into an executable.

Here is an example:

```
all: hello
hello: main.o factorial.o hello.o
g++ main.o factorial.o hello.o -o hello
main.o: main.cpp
g++ -c main.cpp
factorial.o: factorial.cpp
g++ -c factorial.cpp
hello.o: hello.cpp
g++ -c hello.cpp
clean:
rm -rf *.o hello
```

Now we see that the target all has only dependencies, but no system commands. In order for make to execute correctly, it has to meet all the dependencies of the called target (in this case all). Each of the dependencies are searched through all the targets available and executed if found. In this example we see a target called clean. It is useful to have such target if you want to have a fast way to get rid of all the object files and executables.

5. Using Variables:

You can also use variables when writing Make files. It comes in handy in situations where you want to change the compiler, or the compiler options.

```
CC=g++
        CFLAGS=-c -Wall
all: hello
        hello: main.o factorial.o hello.o
$(CC) main.o factorial.o hello.o -o hello
        main.o: main.cpp
$(CC) $(CFLAGS) main.cpp
        factorial.o: factorial.cpp
$(CC) $(CFLAGS) factorial.cpp
        hello.o: hello.cpp
$(CC) $(CFLAGS) hello.cpp
clean:
        rm -rf *.o hello
```

II. LINUX FILE SYSTEMS

The Linux file system considers everything as a file in Linux; whether it is text file images, partitions, compiled programs, directories, or hardware devices.

In Linux, various file formats are used such as:

- text file
- audio file
- video file
- image file
- doc file
- pdf file or any other file contents

Linux files are case sensitive, so test.txt and Test.txt will be considered as two different files.

There are multiple ways to create a file in Linux. Some conventional methods are as follows:

- using cat command
- using touch command
- using redirect '>' symbol
- using echo command
- using printf command
- using a different text editor like vim, nano, vi

1. Using cat command

The cat command is one of the most used commands in Linux. It is used to create a file, display the content of the file, concatenate the contents of multiple files, display the line numbers, and more.

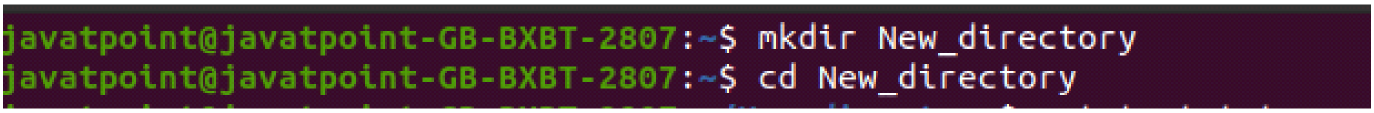
Create a directory and named it as New_directory, execute the mkdir command as follows:

    mkdir New_directory

Change directory to it:

    cd New_directory

Output:



```
javatpoint@javatpoint-GB-BXBT-2807:~$ mkdir New_directory
javatpoint@javatpoint-GB-BXBT-2807:~$ cd New_directory
```

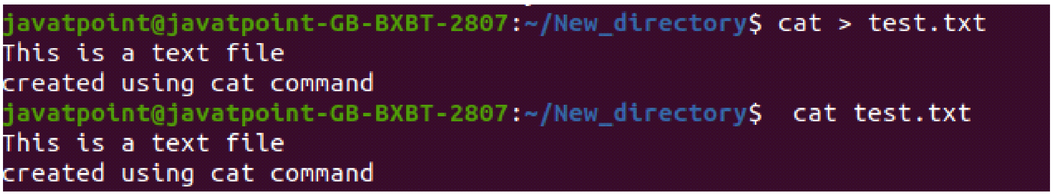
Now execute the cat command to create a file:

cat > test.txt

The above command will create a text file and will enter in the editor mode. Now, enter the desired text and press **CTRL + D** key to save and exit the file and it will return to the command line.

To display the content of the file, execute the cat command as follows:

Output:



```
javatpoint@javatpoint-GB-BXBT-2807:~/New_directory$ cat > test.txt
This is a text file
created using cat command
javatpoint@javatpoint-GB-BXBT-2807:~/New_directory$  cat test.txt
This is a text file
created using cat command
```

2. Linux Create File - Using the touch command

The touch command is also one of the popular commands in Linux. It is used to create a new file, update the time stamp on existing files and directories. It can also create empty files in Linux.

The touch command is the simplest way to create a new file from the command line. We can create multiple files by executing this command at once.

(i) Single File:

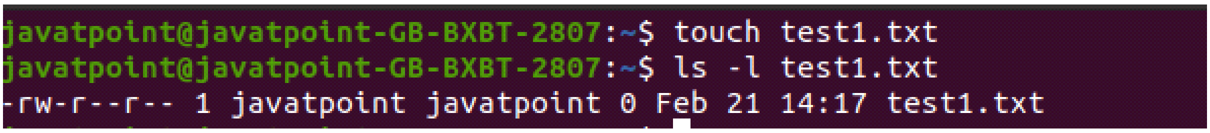
To create a file, execute the touch command followed by the file name as given below:

touch test1.txt

To list the information of the created file, execute the below command:

ls - l test1.txt

Output:



```
javatpoint@javatpoint-GB-BXBT-2807:~$ touch test1.txt
javatpoint@javatpoint-GB-BXBT-2807:~$ ls -l test1.txt
-rw-r--r-- 1 javatpoint javatpoint 0 Feb 21 14:17 test1.txt
```

(ii) Multiple File:

To create multiple files at once, specify files and their extensions after the touch command along with a single space. Execute the below command to create three files at once:

touch test1.txt test2.txt test3.txt

To create two different types of file, execute the command as follows:

touch test4.txt test.odt

The above command will create two different files named as test4.txt and test.odt.

3. Linux - File Permission / Access Modes

File ownership is an important component of Unix that provides a secure method for storing files.

Every file in Unix has the following attributes:

Owner permissions:

The owner's permissions determine what actions the owner of the file can perform on the file.

Group permissions:

The group's permissions determine what actions a user, who is a member of the group that a file belongs to, can perform on the file.

Other (world) permissions:

The permissions for others indicate what action all other users can perform on the file.

The Permission Indicators:

The permissions are broken into groups of threes, and each position in the group denotes a specific permission, in this order:

read (r), write (w), execute (x) –

1. The first three characters (2-4) represent the permissions for the file's owner. For example, **-rwxr-xr--** represents that the owner has read (r), write (w) and execute (x) permission.
2. The second group of three characters (5-7) consists of the permissions for the group to which the file belongs. For example, **-rwxr-xr--** represents that the group has read (r) and execute (x) permission, but no write permission.
3. The last group of three characters (8-10) represents the permissions for everyone else. For example, **-rwxr-xr--** represents that there is **read (r)** only permission.

4. File Access Modes

The permissions of a file are the first line of defense in the security of a Unix system.

The basic building blocks of Unix permissions are the **read**, **write**, and **execute** permissions, which have been described below:

Read

Grants the capability to read, i.e., view the contents of the file.

Write

Grants the capability to modify, or remove the content of the file.

Execute

Users with execute permissions can run a file as a program.

5. Directory Access Modes

Directory access modes are listed and organized in the same manner as any other file.

There are a few differences that need to be mentioned:

Read

Access to a directory means that the user can read the contents. The user can look at the **filenames** inside the directory.

Write

Access means that the user can add or delete files from the directory.

Execute

Executing a directory doesn't really make sense, so think of this as a traverse permission.

A user must have **execute** access to the **bin** directory in order to execute the **ls** or the **cd** command.

6. Changing Permissions

To change the file or the directory permissions, you use the **chmod** (change mode) command. There are two ways to use chmod — the symbolic mode and the absolute mode.

(i) Using chmod in Symbolic Mode

The easiest way for a beginner to modify file or directory permissions is to use the symbolic mode. With symbolic permissions you can add, delete, or specify the permission set you want by using the operators in the following table.

S No	Chmod operator & Description
1	**+** Adds the designated permission(s) to a file or directory.
2	**-** Removes the designated permission(s) from a file or directory.
3	**=** Sets the designated permission(s).

Example using testfile:

Running **ls -1** on the testfile shows that the file's permissions are as follows:

```
$ls -l testfile
-rwxrwxr--  1 amrood   users 1024  Nov 2 00:10  testfile
```

Then each example **chmod** command from the preceding table is run on the testfile, followed by **ls –l**, so you can see the permission changes:

```
$chmod o+wx testfile
$ls -l testfile
-rwxrwxrwx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod u-x testfile
$ls -l testfile
-rw-rwxrwx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod g = rx testfile
$ls -l testfile
-rw-r-xrwx  1 amrood   users 1024  Nov 2 00:10  testfile
```

Now, combine these commands on a single line:

```
$chmod o+wx,u-x,g = rx testfile
$ls -l testfile
-rw-r-xrwx  1 amrood   users 1024  Nov 2 00:10  testfile
```

(ii) Using chmod with Absolute Permissions

The second way to modify permissions with the chmod command is to use a number to specify each set of permissions for the file.

Each permission is assigned a value, as the following table shows, and the total of each set of permissions provides a number for that set.

Number	Octal Permission Representation	Ref
0	No permission	---
1	Execute permission	--x
2	Write permission	-w-
3	Execute and write permission: 1 (execute) + 2 (write) = 3	-wx
4	Read permission	r--
5	Read and execute permission: 4 (read) + 1 (execute) = 5	r-x

6	Read and write permission: 4 (read) + 2 (write) = 6	rw-
7	All permissions: 4 (read) + 2 (write) + 1 (execute) = 7	rwx

Here's an example using the testfile. Running **ls -1** on the testfile shows that the file's permissions are as follows:

```
$ls -l testfile
-rwxrwxr--  1 amrood   users 1024  Nov 2 00:10  testfile
```

Then each example **chmod** command from the preceding table is run on the testfile, followed by **ls –l**, so you can see the permission changes:

```
$ chmod 755 testfile
$ls -l testfile
-rwxr-xr-x  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod 743 testfile
$ls -l testfile
-rwxr---wx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod 043 testfile
$ls -l testfile
----r---wx  1 amrood   users 1024  Nov 2 00:10  testfile
```

(iii) Changing Owners and Groups:

While creating an account on Unix, it assigns a **owner ID** and a **group ID** to each user. All the permissions mentioned above are also assigned based on the Owner and the Groups.

Two commands are available to change the owner and the group of files:

- **chown** − The **chown** command stands for **"change owner"** and is used to change the owner of a file.
- **chgrp** − The **chgrp** command stands for **"change group"** and is used to change the group of a file.

(a) Changing Ownership:

The **chown** command changes the ownership of a file. The basic syntax is as follows:

```
$ chown user filelist
```

The value of the user can be either the **name of a user** on the system or the **user id (uid)** of a user on the system.

Example:

```
$ chown amrood testfile
```

$Changes the owner of the given file to the user **amrood**.

(b) Changing Group Ownership:

The **chgrp** command changes the group ownership of a file. The basic syntax is as follows −

```
$ chgrp group filelist
```

The value of group can be the **name of a group** on the system or **the group ID (GID)** of a group on the system.

Example:

```
$ chgrp special testfile
```

$Changes the group of the given file to **special** group.

Schedule:

Day/ Time	8:45 am to 9:35 am	9:35 am to 10:25 am	10:25 am to 10:40 am	10:40 am to 11:30 am	11:30 am to 12:20 pm	12:20 pm to 1:30 pm	1:30 pm to 2:20 pm	2:20 pm to 3:10 pm	3:10 pm to 3:25 pm	3:25 pm to 4:15 pm
04	Basic shell commands in make files		**Tea Break**	File handling commands		**Lunch Break**	File modes and changing permissions		**Tea Break**	Task Assessment

Description of the task:

Task 1 : Write a program for making a simple make file.

Task 2 : Write a program for how make process a make file.

Task 3 : Write a command that arguments to specify the make file.

Task 4 : Write a command for defining and redefining pattern rules.

Task 5 : Create Files using all basic commands, apply various Access Permissions and modify the privileges of the files.

Assessment Pattern:

Sl. No.	**Aspect Type (M or J)**	**Aspect of description**	**Extra aspect of description**	**Requirement (only for M)**	**Max. marks**
1.	J	Creation, Reading and Naming a file	Yes - **1** No - **0**	-	1
2.	J	Implementation of target and make utility	Yes - **1** No - **0**	-	1
3.	J	Usage of variables and dependencies	Yes - **1** No - **0**	-	1
4.	J	Implementing pattern rules	Yes - **1** No - **0**	-	1
5.	J	Create different files using Touch and echo command	Yes - **1** No - **0**	-	1
6.	J	Show the directory hierarchy in a tree-like format	Yes - **1** No - **0**	-	1
7.	J	Verify the Group ID	Yes - **1** No - **0**	-	1

8.	J	revokes all the read(r), write(w) and execute(x) permission of the file	If yes, then a. read(r), is used – **0.5** b. write(w) is used **– 0.5** No **- 0**	-	1
9.	J	Time Management	**0 -** Exceeded 45 mins **0.5 -** Completed within 30 to 45 mins **1 -** Completed within 30 min	-	1
10.	J	**Coding Ethics** Proper Indentation Overall design look	**0 -** If none of the aspect is found **0.5 -** If exhibiting any one aspect **1 -** If exhibiting all two aspects	-	1
Total Marks					**10**

Day 05	Linux Signals and Handling Signals & Process in Linux

Objective:

To understand the basic concepts of Signals, Basic Signals handling in Linux.

Outcomes:

To learn and handle signals and process in Linux and Catch a signal using C programming.

Resources Required	**:**	Linux Virtual environment
Safety Precautions	**:**	Nil
Prerequisites	**:**	C programming

Theory:

Reference link:

https://docs.google.com/document/d/1qPLOcUiX3jFo4mo1fSKUUqejJ8GQIGf0/edit?usp=share_link&ouid=102585827095202023013&rtpof=true&sd=true

https://www.javatpoint.com/linux-signals

A feature of LINUX programming is the idea of sending and receiving signals. A signal is a kind of (usually software) interrupt, used to announce asynchronous events to a process.

Signals are software interrupts sent to a program to indicate that an important event has occurred. The events can vary from user requests to illegal memory access errors. Some signals, such as the interrupt signal, indicate that a user has asked the program to do something that is not in the usual flow of control. The name of a LINUX signal begins with "SIG". Although signals are numbered, we normally refer to them by their names.

For example:

- SIGINT is a signal generated when a user presses Control-C. This will terminate the program from the terminal.
- SIGALRM is generated when the timer set by the alarm function goes off.

- SIGABRT is generated when a process executes the abort function.
- SIGSTOP tells LINUX to pause a process to be resumed later.
- SIGCONT tells LINUX to resume the processed paused earlier.
- SIGSEGV is sent to a process when it has a segmentation fault.
- SIGKILL is sent to a process to cause it to terminate at once.
- Some signals terminate the receiving process: SIGHUP, SIGINT, SIGTERM, SIGKILL. There are signals that terminate the process along with a core dump to help programmers debug what went wrong: SIGABRT (abort signal), SIGBUS (bus error), SIGILL (illegal instruction), SIGSEGV (invalid memory reference), SIGSYS (bad system call). Other signals stop the process: SIGSTOP, SIGTSTP. SIGCONT is a signal that resumes a stopped process.

1. Listing all the signals using list command:

The Command ”kill –l” lists all the 64 signal names.

```
        user@user-System-Product-Name:~$ kill -l
 1) SIGHUP    2) SIGINT      3) SIGQUIT             4) SIGILL    5) SIGTRAP
 6) SIGABRT    7) SIGBUS    8) SIGFPE    9) SIGKILL    10) SIGUSR1
11) SIGSEGV    12) SIGUSR2    13) SIGPIPE    14) SIGALRM    15) SIGTERM
16) SIGSTKFLT    17) SIGCHLD    18) SIGCONT    19) SIGSTOP    20) SIGTSTP
21) SIGTTIN    22) SIGTTOU    23) SIGURG    24) SIGXCPU    25) SIGXFSZ
26) SIGVTALRM    27) SIGPROF    28) SIGWINCH    29) SIGIO    30) SIGPWR
31) SIGSYS    34) SIGRTMIN    35) SIGRTMIN+1    36) SIGRTMIN+2    37) SIGRTMIN+3
38) SIGRTMIN+4    39) SIGRTMIN+5    40) SIGRTMIN+6    41) SIGRTMIN+7
42) SIGRTMIN+8 43) SIGRTMIN+9    44) SIGRTMIN+10    45) SIGRTMIN+11
46) SIGRTMIN+12 47) SIGRTMIN+13 48) SIGRTMIN+14    49) SIGRTMIN+15
50) SIGRTMAX-14    51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11
54) SIGRTMAX-10    55) SIGRTMAX-9    56) SIGRTMAX-8 57) SIGRTMAX-7
```

```
58) SIGRTMAX-6    59) SIGRTMAX-5    60) SIGRTMAX-4    61) SIGRTMAX-3
62) SIGRTMAX-2 63) SIGRTMAX-1    64) SIGRTMAX
```

2. Identifying the sleeping Process and killing the same using 'SIGTERM' / CODE command

```
user@user-System-Product-Name:~$ sleep 4545 &
[1] 7826
```

Here the following code 15 refers termination Command used for terminating the process

```
user@user-System-Product-Name:~$ kill -15 7826
[1]+  Terminated                    sleep 4545
```

3. Stopping the process using SIGSTOP command

```
user@user-System-Product-Name:~$ sleep 4545 &
[1] 7848
user@user-System-Product-Name:~$ kill -SIGSTOP 7848
[1]+  Stopped              sleep 4545
user@user-System-Product-Name:~$ sleep 4545 &
[2] 7955
```

4. Continuing the process using SIGCONT command

```
user@user-System-Product-Name:~$ ps -C sleep
PID TTY         TIME CMD
7848 pts/0      00:00:00 sleep
7955 pts/0      00:00:00 sleep
user@user-System-Product-Name:~$ kill -l
```

```
 1) SIGHUP     2) SIGINT     3) SIGQUIT     4) SIGILL     5) SIGTRAP
 6) SIGABRT     7) SIGBUS     8) SIGFPE     9) SIGKILL    10) SIGUSR1
11) SIGSEGV    12) SIGUSR2    13) SIGPIPE    14) SIGALRM    15) SIGTERM
16) SIGSTKFLT    17) SIGCHLD    18) SIGCONT    19) SIGSTOP    20) SIGTSTP
21) SIGTTIN    22) SIGTTOU    23) SIGURG    24) SIGXCPU    25) SIGXFSZ
26) SIGVTALRM    27) SIGPROF    28) SIGWINCH    29) SIGIO    30) SIGPWR
31) SIGSYS    34) SIGRTMIN    35) SIGRTMIN+1    36) SIGRTMIN+2    37) SIGRTMIN+3
38) SIGRTMIN+4    39) SIGRTMIN+5    40) SIGRTMIN+6    41) SIGRTMIN+7
42) SIGRTMIN+8 43) SIGRTMIN+9    44) SIGRTMIN+10    45) SIGRTMIN+11
46) SIGRTMIN+12    47) SIGRTMIN+13 48) SIGRTMIN+14    49) SIGRTMIN+15
50) SIGRTMAX-14    51) SIGRTMAX-13    52) SIGRTMAX-12 53) SIGRTMAX-11
54) SIGRTMAX-10    55) SIGRTMAX-9    56) SIGRTMAX-8    57) SIGRTMAX-7
58) SIGRTMAX-6    59) SIGRTMAX-5    60) SIGRTMAX-4    61) SIGRTMAX-3
62) SIGRTMAX-2   63) SIGRTMAX-1    64) SIGRTMAX
```

```
user@user-System-Product-Name:~$ kill  -SIGCONT 7848
user@user-System-Product-Name:~$ jobs
[1]-  Running              sleep 4545 &
[2]+  Running              sleep 4545 &
user@user-System-Product-Name:~$ kill SIGTERM 7848
```

Note:

```
bash: kill: SIGTERM: arguments must be process or job IDs
user@user-System-Product-Name:~$ kill SIGTERM [2]
bash: kill: [2]: arguments must be process or job IDs
[1] - Terminated                    sleep 4545
```

```
user@user-System-Product-Name:~$ sleep 4548 &
[3] 8014
user@user-System-Product-Name:~$ jobs
[2]-  Running                 sleep 4545 &
[3]+  Running                 sleep 4548 &
user@user-System-Product-Name:~$ pstree  -s 8014
systemd───systemd───gnome-terminal-───bash───sleep
```

5. C program to catch Signal:

Whenever ctrl+c is pressed, a signal SIGINT is sent to the process. The default action of this signal is to terminate the process. But this signal can also be handled.

Program:

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
void sig_handler(int signo)
{
        if (signo == SIGINT)
                printf("received SIGINT\n");
}
int main(void)
{
        if (signal(SIGINT, sig_handler) == SIG_ERR)
                printf("\ncan't catch SIGINT\n");
        // A long long wait so that we can easily issue a signal to this process
        while(1)
        sleep (1);
        return 0;
}
```

In the code above, we have simulated a long running process using an infinite while loop.

- A function sig_handler is used a s a signal handler. This function is registered to the kernel by passing it as the second argument of the system call 'signal' in the main() function.
- The first argument to the function 'signal' is the signal we intend the signal handler to handle which is SIGINT in this case.

Sleep(1) function has been used in the while loop so that while loop executes after some time (ie one second in this case). This becomes important because otherwise an infinite while loop running wildly may consume most of the CPU making the computer very slow.

When the process is run and terminate the process using Ctrl+C:

```
$ ./sigfunc
^Creceived SIGINT
^Creceived SIGINT
^Creceived SIGINT
^Creceived SIGINT
^Creceived SIGINT
^Creceived SIGINT
^Creceived                                                        SIGINT
```

The key combination ctrl+c tried several times but each time the process didn't terminate. This is because the signal was handled in the code and this was confirmed from the print we got on each line

6. SIGKILL, SIGSTOP and User Defined Signals

Apart from handling the standard signals (like INT, TERM etc) that are available. The user defined signals that can be sent and handled. Following is the code handling a user defined signal USR1:

```
#include<stdio.h>

#include<signal.h>

#include<unistd.h>

void sig_handler(int signo)

{

        if (signo == SIGUSR1)

                printf("received SIGUSR1\n");

        else if (signo == SIGKILL)

                printf("received SIGKILL\n");

        else if (signo == SIGSTOP)

                printf("received SIGSTOP\n");

}

int main(void)

{

        if (signal(SIGUSR1, sig_handler) == SIG_ERR)

                printf("\ncan't catch SIGUSR1\n");

        if (signal(SIGKILL, sig_handler) == SIG_ERR)

                printf("\ncan't catch SIGKILL\n");

        if (signal(SIGSTOP, sig_handler) == SIG_ERR)

                printf("\ncan't catch SIGSTOP\n");

        // A long long wait so that we can easily issue a signal to this process

        while(1)

        sleep(1);
        return 0;
}
```

We have tried to handle a user defined signal USR1. Also, two signals KILL and STOP cannot be handled. So, we have also tried to handle these two signals so as to see how the ‘signal’ system call responds in this case.

Output:

```
        $ ./sigfunc
        can't catch SIGKILL
        can't catch SIGSTOP
```

The system call ‘signal’ tries to register handler for KILL and STOP signals, the signal function fails indicating that these two signals cannot be caught.

Now we try to pass the signal USR1 to this process using the kill command.

```
        $ kill -USR1 2678
        $ ./sigfunc
        can't catch SIGKILL
        can't catch SIGSTOP
        received SIGUSR1
```

The user defined signal USR1 was received in the process and was handled properly.

II. PROCESS MANAGEMENT

Theory:

A process means a program in execution. It generally takes an input, processes it, and gives us the appropriate output.

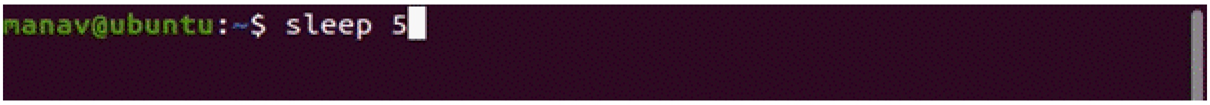
Reference link:

https://docs.google.com/document/d/1qUXAK61HfSDU8oXmwdpudbo85uepyKLMAkL7IOSnQ2U/edit?usp=sharing

1. Managing the Processes

Step 1. How to execute foreground process.

sleep 5



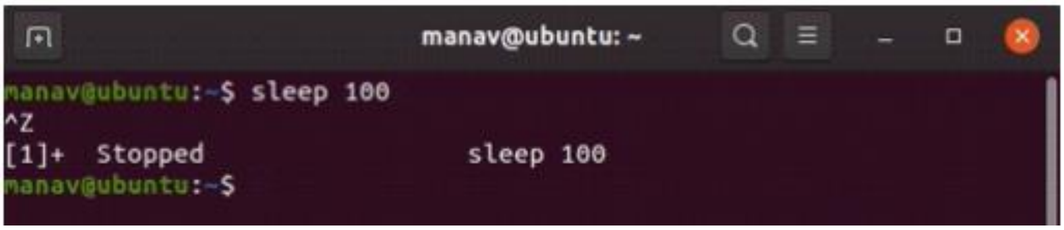
```
manav@ubuntu:~$ sleep 5
```

This command will be executed in the terminal and we would be able to execute another command after the execution of the above command.

Step 2. Stopping a process in between of its execution.

To stop a foreground process in between of its execution we may press CTRL+Z to force stop it.

sleep 100

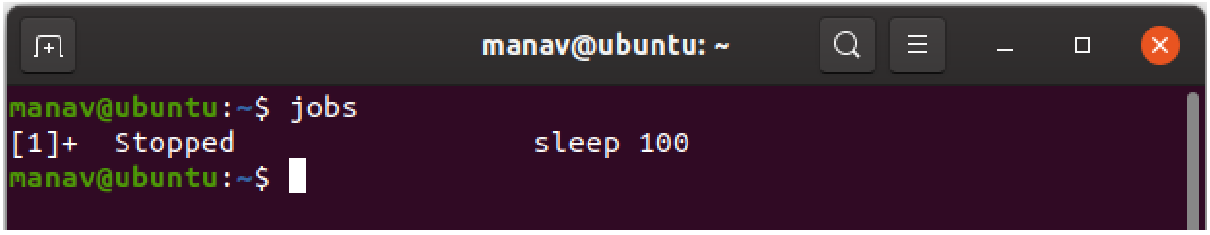


```
manav@ubuntu:~$ sleep 100
^Z
[1]+  Stopped                 sleep 100
manav@ubuntu:~$
```

Pressing CTRL+Z in between the execution of the command will stop it.

Step 3. To get the list of jobs that are either running or stopped.

jobs

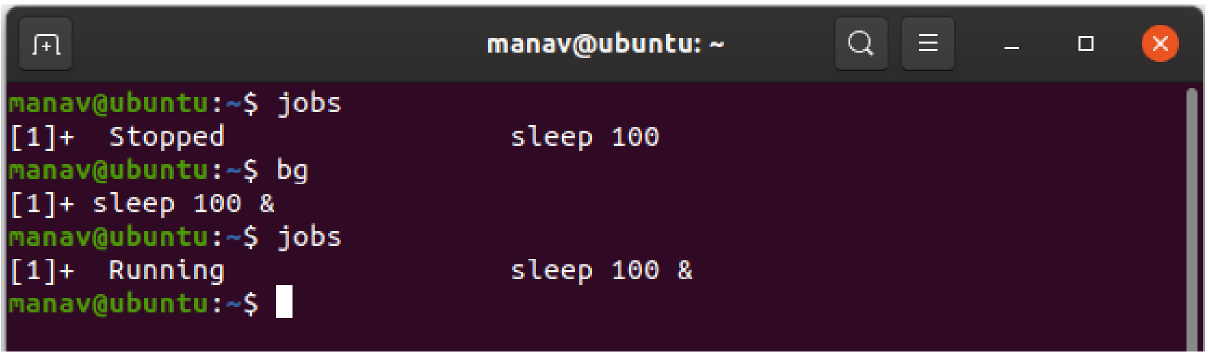


```
manav@ubuntu:~$ jobs
[1]+  Stopped                 sleep 100
manav@ubuntu:~$ 
```

It will display the stopped processes in this terminal and even the pending ones.

Step 4. To run all the pending and force stopped jobs in the background.

bg

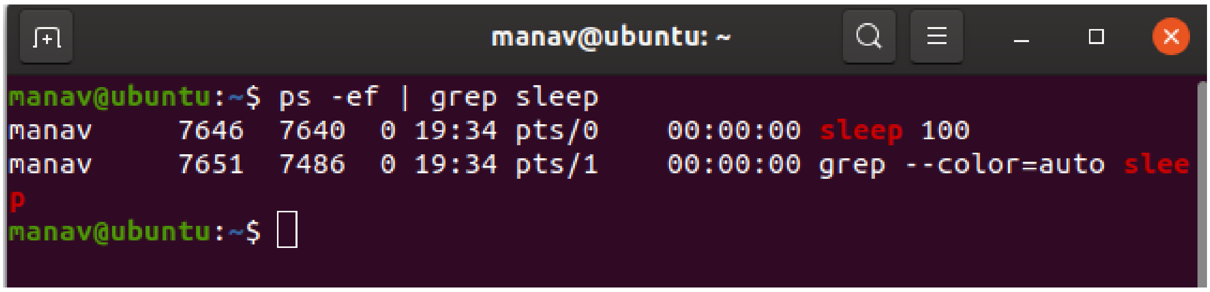


```
manav@ubuntu:~$ jobs
[1]+  Stopped                 sleep 100
manav@ubuntu:~$ bg
[1]+ sleep 100 &
manav@ubuntu:~$ jobs
[1]+  Running                 sleep 100 &
manav@ubuntu:~$ 
```

This will start the stopped and pending processes in the background.

Step 5. To get details of a process running in background.

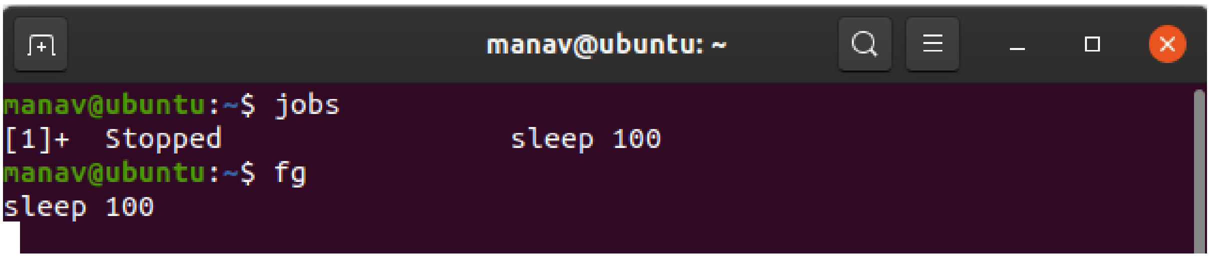
ps -ef | grep sleep



```
manav@ubuntu:~$ ps -ef | grep sleep
manav     7646  7640  0 19:34 pts/0    00:00:00 sleep 100
manav     7651  7486  0 19:34 pts/1    00:00:00 grep --color=auto slee
p
manav@ubuntu:~$ 
```

Step 6. To run all the pending and force stopped jobs in the foreground.

fg

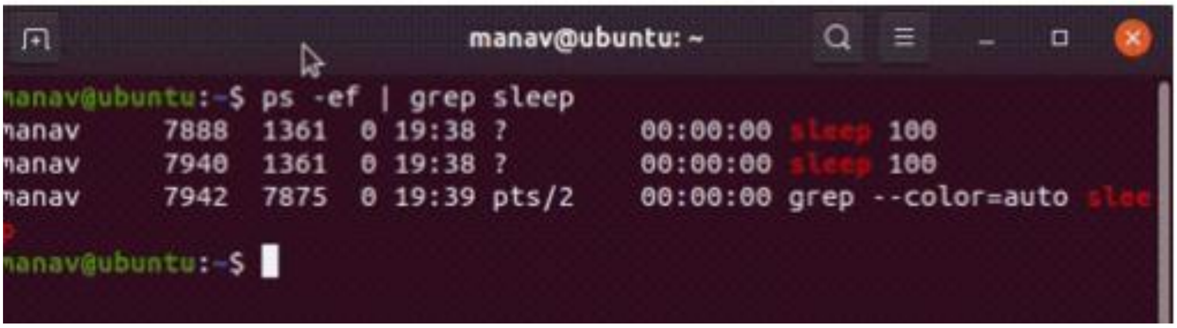


```
manav@ubuntu:~$ jobs
[1]+  Stopped                 sleep 100
manav@ubuntu:~$ fg
sleep 100
```

This will start the stopped and pending processes in the foreground.

Step 7. To run a process in the background without getting impacted by the closing of the terminal.

nohup sleep 100 &

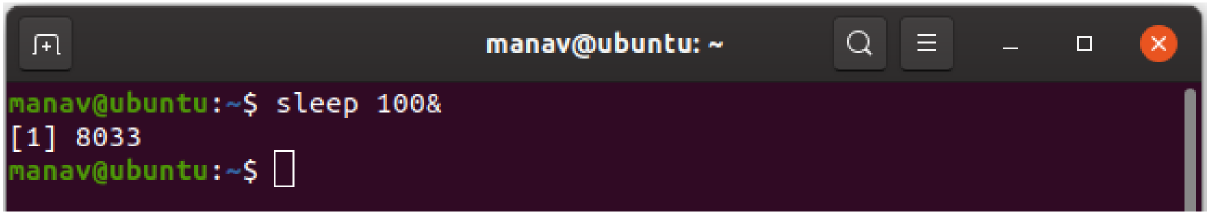


```
nanav@ubuntu:~$ ps -ef | grep sleep
nanav     7888  1361  0 19:38 ?        00:00:00 sleep 100
nanav     7940  1361  0 19:38 ?        00:00:00 sleep 100
nanav     7942  7875  0 19:39 pts/2    00:00:00 grep --color=auto slee
p
nanav@ubuntu:~$ 
```

While executing, it will even store all the output after execution in nohup.out file.

Step 8. To run some processes in the background directly.

sleep 100&

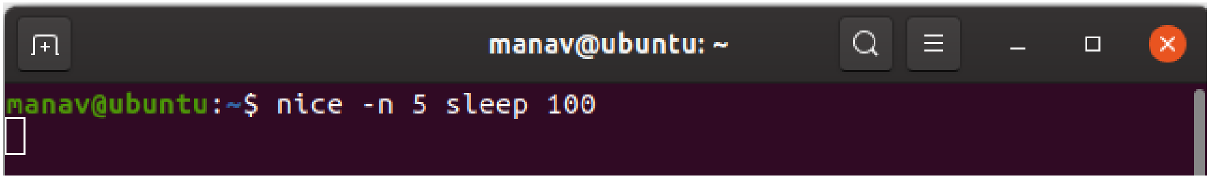


```
manav@ubuntu:~$ sleep 100&
[1] 8033
manav@ubuntu:~$ 
```

This will run the process in the background and will display the process id of the process.

Step 9. To run processes with priority.

nice -n 5 sleep 100

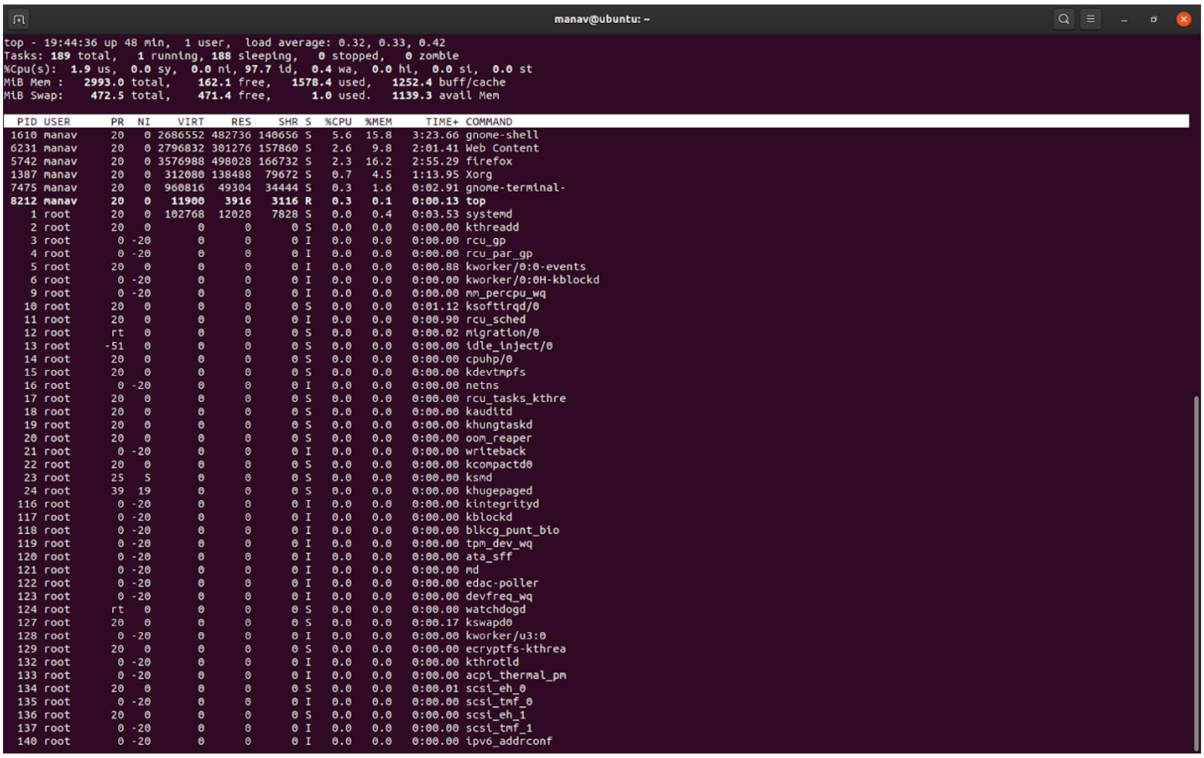


```
manav@ubuntu:~$ nice -n 5 sleep 100
```

The top priority is -20 but as it may affect the system processes so we have used the priority 5.

Step 10. To get the list of all the running processes on your Linux machine.

Top



```
top - 19:44:36 up 48 min,  1 user,  load average: 0.32, 0.33, 0.42
Tasks: 189 total,   1 running, 188 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1.9 us,  0.0 sy,  0.0 ni, 97.7 id,  0.4 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   2993.0 total,    162.1 free,   1578.4 used,   1252.4 buff/cache
MiB Swap:    472.5 total,    471.4 free,      1.0 used.   1139.3 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 1610 manav     20   0 2686552 482736 140656 S   5.6  15.8   3:23.66 gnome-shell
 6231 manav     20   0 2796832 301276 157860 S   2.6   9.8   2:01.41 Web Content
 5742 manav     20   0 3576988 498028 166732 S   2.3  16.2   2:55.29 firefox
 1387 manav     20   0  312080 138488  79672 S   0.7   4.5   1:13.95 Xorg
 7475 manav     20   0  960816  49304  34444 S   0.3   1.6   0:02.91 gnome-terminal-
 8212 manav     20   0   11900   3916   3116 R   0.3   0.1   0:00.13 top
    1 root      20   0  102768  12020   7828 S   0.0   0.4   0:03.53 systemd
    2 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kthreadd
    3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
    4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
    5 root      20   0       0      0      0 I   0.0   0.0   0:00.88 kworker/0:0-events
    6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/0:0H-kblockd
    9 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_percpu_wq
   10 root      20   0       0      0      0 S   0.0   0.0   0:01.12 ksoftirqd/0
   11 root      20   0       0      0      0 I   0.0   0.0   0:00.90 rcu_sched
   12 root      rt   0       0      0      0 S   0.0   0.0   0:00.02 migration/0
   13 root     -51   0       0      0      0 S   0.0   0.0   0:00.00 idle_inject/0
   14 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/0
   15 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kdevtmpfs
   16 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 netns
   17 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_tasks_kthre
   18 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kauditd
   19 root      20   0       0      0      0 S   0.0   0.0   0:00.00 khungtaskd
   20 root      20   0       0      0      0 S   0.0   0.0   0:00.00 oom_reaper
   21 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 writeback
   22 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kcompactd0
   23 root      25   5       0      0      0 S   0.0   0.0   0:00.00 ksmd
   24 root      39  19       0      0      0 S   0.0   0.0   0:00.00 khugepaged
  116 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kintegrityd
  117 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kblockd
  118 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 blkcg_punt_bio
  119 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 tpm_dev_wq
  120 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 ata_sff
  121 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 md
  122 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 edac-poller
  123 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 devfreq_wq
  124 root      rt   0       0      0      0 S   0.0   0.0   0:00.00 watchdogd
  127 root      20   0       0      0      0 S   0.0   0.0   0:00.17 kswapd0
  128 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/u3:0
  129 root      20   0       0      0      0 S   0.0   0.0   0:00.00 ecryptfs-kthrea
  132 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kthrotld
  133 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 acpi_thermal_pm
  134 root      20   0       0      0      0 S   0.0   0.0   0:00.01 scsi_eh_0
  135 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 scsi_tmf_0
  136 root      20   0       0      0      0 S   0.0   0.0   0:00.00 scsi_eh_1
  137 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 scsi_tmf_1
  140 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 ipv6_addrconf
```

2. Process, Parent Process, and Child Process

Running program is a process. From this process, another process can be created. There is a parent-child relationship between the two processes. This can be achieved using a library function called fork(). fork() function splits the running process into two processes, the existing one is known as parent and the new process is known as a child. Here is a program that demonstrates this:

Program 1: Demonstration Program in C

```
// C program to demonstrate the above concept

#include <sys/types.h>

#include<stdio.h>

#include <unistd.h>

// Driver code

int main()

{

        printf ("Before Forking\n");

        fork();

        printf ("After Forking\n");

}
```

Output:

```
Before Forking

After Forking

Before Forking

After Forking
```

Program 2:

The task here is to show how to perform two different but interrelated jobs simultaneously. Hence, the actual code for file copying and playing the animated GIF file has been skipped only the approach for performing 2 jobs simultaneously is shown.

```
// C program for the above approach

#include <sys/types.h>

// Driver Code

int main( )

{

        int pid;

        pid = fork();

        if (pid == 0)

        {

                printf ("In child process\n");

                /* code to play animated GIF file */

        }

        else

        {

                printf ("In parent process\n");

                /* code to copy file */

        }

}
```

3. fork(), vfork(), exec() and clone() commands:

System calls

System calls provide an interface to the services made available by an operating system. The system calls fork(), vfork(), exec(), and clone() are all used to create and manipulate processes.

(i) fork()

Processes execute the fork() system call to create a new child process.

Executing fork()   ---- (1)

Create a file named fork_test.c with the Nano editor:

```
$ nano fork_test.cCopy
```

Next, we add this content:

```
#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

int main(int argc, char **argv)

{

        pid_t pid = fork();

        if (pid==0)

        {

                printf("This is the Child process and pid is: %d\n",getpid());

                exit(0);

        }

        else if (pid > 0)

        {

                printf("This is the Parent process and pid is: %d\n",getpid());

        }

        else

        {

                printf("Error while forking\n");

                exit(EXIT_FAILURE);
        }
        return 0;
}
```

Copy ---- (2)

Here, we're starting a new process and using the variable pid to store the process identifier of the child process created by the fork() call. We then proceed to check if the value of pid returned by the fork() call is equal to zero. The fork() call returns the value of the child process as zero to differentiate it from its parent. The actual value of the child process identifier is the value returned to the parent process. Finally, we check for errors and print an error message.

After saving the changes, we use the cc command to compile fork_test.c:

```
$ cc fork_test.cCopy
```

This creates an executable file called a.out in the working directory.

Finally, we can execute the a.out file:

```
$ ./a.out
```

```
This is the Parent process and pid is: 69032

This is the Child process and pid is: 69033Copy
```

(ii) vfork()

Similar to the fork() system call, vfork() also creates a child process that's identical to its parent process. However, the child process temporarily suspends the parent process until it terminates. This is because both processes use the same address space, which contains the stack, stack pointer, and instruction pointer.

vfork() acts as a special case of the clone() system call. It creates new processes without copying the address space of the parent process. This is useful in performance-oriented applications.

The parent process is always suspended once the child process is created. It remains suspended until the child process terminates normally, abnormally, or until it executes the exec system call starting a new process.

The child process created by the vfork() system call inherits its parent's attributes. These include file descriptors, current working directory, signal dispositions, and more.

Executing vfork()   ----- (1)

Create a file named vfork_test.c:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
        pid_t pid = vfork();  //creating the child process
        printf("parent process pid before if...else block: %d\n", getpid());
        if (pid == 0)
        {  //checking if this is the a child process
                printf("This is the child process and pid is: %d\n\n", getpid());
                exit(0);
        }
        else if (pid > 0)
        {  //parent process execution
                printf("This is the parent process and pid is: %d\n", getpid());
        }
        else
        {
                printf("Error while forking\n");
                exit(EXIT_FAILURE);
        }
         return 0;
}
```

Copy ----- (2)

Here, we’re using variable pid to store the PID of the child process created by the vfork() call. We then check to see the value of the parent’s PID before the if…else block.

After saving the changes, let’s compile vfork_test.c:

```
        $ cc vfork_test.cCopy
```

Finally, we can execute the created a.out file:

```
        $ ./a.out
parent process pid before if...else block: 117117
This is the child process and pid is: 117117
parent process pid before if...else block: 117116
This is the parent process and pid is: 117116Copy
```

The vfork() system call returns the output twice, first in the child process and then in the parent process.

Since both processes share the same address space, we have matching PID values in the first output. In the if else block, the child process is run first because it blocks the parent process while executing.

(iii) exec()

The exec() system function runs a new process in the context of an existing process and replaces it. This is also referred to as an overlay.

The function doesn’t create a new process, so the PID doesn’t change. However, the new process replaces the data, heap, stack, and machine code of the current process. It loads the new process into the current process space and executes it from the entry point. Control never returns to the original process unless there’s an exec() error.

This system function belongs to a family functions that
includes execl(), execlp(), execv(), execvp(), execle(), and execve().

Executing exec() ------ (1)

Create the first program named exec_test1.c:

```
#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

int main(int argc, char *argv[])

{

        printf("PID of exec_test1.c = %d\n", getpid());

        char *args[] = {"Hello", "From", "Parent", NULL};

        execv("./exec_test2", args);

        printf("Back to exec_test1.c");

        return 0;

}
```

Copy ----- (2)

Create the second program called exec_test2.c:

```
#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

int main(int argc, char *argv[])

{

printf("Hello from exec_test2.c\n");

printf("PID of exec_test2.c process is: %d\n", getpid());

return 0;
}
```

Copy

Here, we’re printing a message and the PID of the process launched by exec() from the first program.

We use the cc command to compile exec_test1.c to an executable:

```
$ cc exec_test1.c -o exec_test1Copy
```

This creates an executable file called exec_test1 in the working directory.

Then, we compile the second program:

```
$ cc exec_test2.c -o exec_test2Copy
```

This creates an executable file called exec_test1 in the working directory.

Finally, we can execute the exec_test1 file:

```
$ ./exec_test1

PID of exec_test1.c = 171939

Hello from exec_test2.c

PID of exec_test2.c process is: 171939
```

From the output above, we can notice that the PID didn’t change in the second program’s process. Furthermore, the last print statement from exec_test1.c file wasn’t printed. This is because executing the execv() system call replaced the currently running process, and we haven’t included a way of returning back to the first process.

(iv) clone()

The clone() system call is an upgraded version of the fork call. It’s powerful since it creates a child process and provides more precise control over the data shared between the parent and child processes. The caller of this system call can control the table of file descriptors, the table of signal handlers, and whether the two processes share the same address space.

clone() system call allows the child process to be placed in different namespaces. With the flexibility that comes with using the clone() system call, we can choose to share an address space with the parent process, emulating the vfork() system call. We can also choose to share file system information, open files, and signal handlers using different flags available.

This is the signature of the clone() system call:

```
int clone(int (*fn)(void *), void *stack, int flags, void *arg, ...
/* pid_t *parent_tid, void *tls, pid_t *child_tid */ );Copy
```

Details:

- *fn: pointer that points to a function
- *stack: points to the smallest byte of a stack
- pid_t: process identifier (PID)
- *parent_tid: points to the storage location of child process thread identifier (TID) in parent process memory
- *child_tid: points to the storage location of the child process thread identifier (TID) in the child process memory

Executing clone() ------ (1)

Create a file named clone_test.c:

```
// We have to define the _GNU_SOURCE to get access to clone(2) and the CLONE_*
#define _GNU_SOURCE
#include <sched.h>
#include <sys/syscall.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
static int child_func(void* arg)
{
        char* buffer = (char*)arg;

        printf("Child sees buffer = \"%s\"\n", buffer);

        strcpy(buffer, "hello from child");

        return 0;

}

int main(int argc, char** argv)

{

        // Allocate stack for child task.

        const int STACK_SIZE = 65536;

        char* stack = malloc(STACK_SIZE);

        if (!stack)

        {

                perror("malloc");

        exit(1);

        }

        // When called with the command-line argument "vm", set the CLONE_VM flag on.

        unsigned long flags = 0;

        if (argc > 1 && !strcmp(argv[1], "vm"))

        {

                flags |= CLONE_VM;

        }

        char buffer[100];

        strcpy(buffer, "hello from parent");

        if (clone(child_func, stack + STACK_SIZE, flags | SIGCHLD, buffer) == -1)

        {

                perror("clone");
```

```
            exit(1);
        }
        int status;
        if (wait(&status) == -1)
        {
            perror("wait");
            exit(1);
        }
        printf("Child exited with status %d. buffer = \"%s\"\n", status, buffer);
return 0;
}
```

Copy -----(2)

clone() is used in two ways, once with the CLONE_VM flag and once without. We're passing a buffer into the child process, and the child process writes a string to it. We then allocate a stack size for the child process and create a function that checks whether we're executing the file using the CLONE_VM (vm) option. Furthermore, we're creating a buffer of 100 bytes in the parent process and copying a string to it, then executing the clone() system call and checking for errors.

We use the cc command to compile exec_test.c to an executable:

```
$ cc clone_test.cCopy
```

This creates an executable file called a.out in the working directory.

Finally, we can execute the a.out file:

```
./a.out
```

```
Child sees buffer = "hello from parent"

Child exited with status 0. buffer = "hello from parent"
```

Copy -------(2.1)

When we execute it without the vm argument, the CLONE_VM flag isn’t active, and the parent process virtual memory is cloned into the child process. The child process can access the message passed by the parent process in buffer, but anything written into buffer by the child isn’t accessible to the parent process.

But, when we pass in the vm argument, CLONE_VM is active and the child process shares the parent’s process memory. We can see it writing into buffer:

```
$ ./a.out vm
```

```
Child sees buf = "hello from parent"

Child exited with status 0. buf = "hello from child"Copy
```

This time our message is different, and we can see the message passed from the child process.

Schedule:

Day/ Time	8:45 am to 9:35 am	9:35 am to 10:25 am	10:25 am to 10:40 am	10:40 am to 11:30 am	11:30 am to 12:20 pm	12:20 pm to 1:30 pm	1:30 pm to 2:20 pm	2:20 pm to 3:10 pm	3:10 pm to 3:25 pm	3:25 pm to 4:15 pm
05	Linux Signals and Creation of Signals		**Tea Break**	Task 1 and Task 2		**Lunch Break**	Linux Process creation		**Tea Break**	Task 3 & 4 and Assessment

Description of the task:

Task 1: Write a C program to handle and catch Signals in the Linux environment.

Task 2: Write a C program to handle user defined signal.

Task 3: Managing Linux process

Task 4: How to create child process using fork(), vfork(), exec(), clone() system calls.

Assessment Pattern:

Sl. No.	**Aspect Type (M or J)**	**Aspect of description**	**Extra aspect of description**	**Requirement (only for M)**	**Max. marks**
1.	J	Implementation of SIGINT signal	Yes **- 1** No – **0**	-	1
2.	J	Execution and display of signal commands	Yes – **1** No – **0**	-	1
3.	J	Creation of C program to demonstrate the signal handling	Yes **- 1** No – **0**	-	1
4.	J	Creation of C program to demonstrate User defined signal handling	Yes – **1** No – **0**	-	1
5.	J	Foreground process using sleep()	Yes **- 1** No **- 0**	-	1
6.	J	To get details of a process running in background	Yes **- 1** No **- 0**	-	1
7.	J	To run processes with priority	Yes **- 1** No **- 0**	-	1
8.	J	Creation of child process using fork()	Yes **- 1** No **- 0**	-	1
9.	J	Time Management	**0 -** Exceeded 45 mins **0.5 -** Completed within 30 to 45 mins **1 -** Completed within 30 min	-	1
10.	J	**Coding Ethics** Proper Indentation Overall design look	**0 -** If none of the aspect is found **0.5 -** If exhibiting any one aspect **1 -** If exhibiting all two aspects	-	1
Total Marks					**10**

Day 06	Linux Scheduler and Memory Management

Objective:

To understand the Linux scheduler and memory management in Linux.

Outcomes:

To implement scheduling algorithms and memory management techniques using C programming.

Resources Required : Linux Virtual environment

Safety Precautions : Nil

Prerequisites : C programming

Theory:

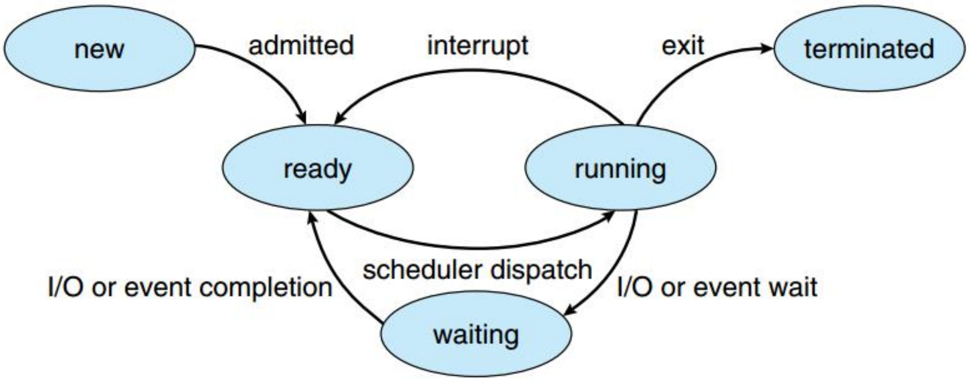
I. LINUX SCHEDULER

Reference Link:

https://kuleuven-diepenbeek.github.io/osc-course/exercises/

Scheduler has two main responsibilities:

1. Choose the next task that is allowed to run on a given processor
2. Dispatching: switching tasks that are running on a given processor



Types of Scheduling:

Scheduler algorithms

1. FCFS

2. SJF

Pre-emptive scheduling

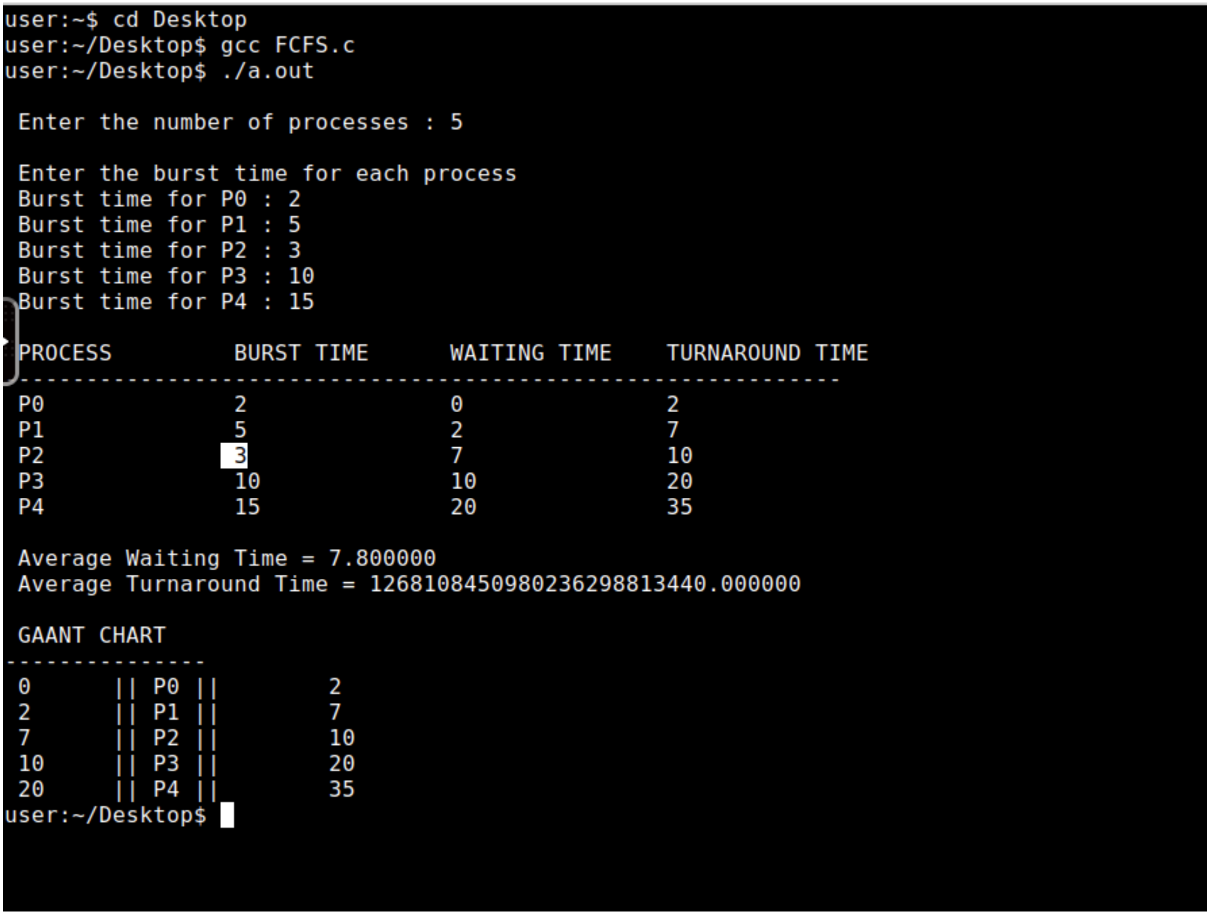
1. Priority-based scheduling
2. Round-Robin scheduling

Program 1: Scheduling algorithms - FCFS

```
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>
int main()
{
	int i, n, *bt, *wt, *tat;
	float avgtat, avgwt;
	printf("\n Enter the number of processes : ");
	scanf("%d", &n);
	bt = (int*)malloc(n*sizeof(int));
	wt = (int*)malloc(n*sizeof(int));
	tat = (int*)malloc(n*sizeof(int));
	printf("\n Enter the burst time for each process \n");
	for(i=0; i<n; i++)
	{
		printf(" Burst time for P%d : ", i);
		scanf("%d", &bt[i]);
	}
	wt[0] = 0;
	tat[0] = bt[0];
	for(i=1; i<n; i++)
	{
	wt[i] = wt[i-1] + bt[i-1];  //waiting time[p] = waiting time[p-1] + Burst Time[p-1]
	tat[i] = wt[i] + bt[i];     //Turnaround Time = Waiting Time + Burst Time
```

```
    }
    for(i=0; i<n; i++)
    {
        avgwt += wt[i];
        avgtat += tat[i];
    }
    avgwt = avgwt/n;
    avgtat = avgtat/n;
    printf("\n PROCESS \t BURST TIME \t WAITING TIME \t TURNAROUND TIME
    \n");
    printf("--------------------------------------------------------------\n");
    for(i=0; i<n; i++)
    {
        printf(" P%d \t\t %d \t\t %d \t\t %d \n", i, bt[i], wt[i], tat[i]);
    }
    printf("\n Average Waiting Time = %f \n Average Turnaround Time = %f \n", avgwt,
    avgtat);
    printf("\n GAANT CHART \n");
    printf("---------------\n");
    for(i=0; i<n; i++)
    {
        printf(" %d\t|| P%d ||\t%d\n", wt[i], i, tat[i]);
    }
}
```

Output:



```
user:~$ cd Desktop
user:~/Desktop$ gcc FCFS.c
user:~/Desktop$ ./a.out

 Enter the number of processes : 5

 Enter the burst time for each process
 Burst time for P0 : 2
 Burst time for P1 : 5
 Burst time for P2 : 3
 Burst time for P3 : 10
 Burst time for P4 : 15

 PROCESS         BURST TIME      WAITING TIME    TURNAROUND TIME
 -------------------------------------------------------------
 P0              2               0               2
 P1              5               2               7
 P2              3               7               10
 P3              10              10              20
 P4              15              20              35

 Average Waiting Time = 7.800000
 Average Turnaround Time = 1268108450980236298813440.000000

 GAANT CHART
---------------
 0      || P0 ||        2
 2      || P1 ||        7
 7      || P2 ||        10
 10     || P3 ||        20
 20     || P4 ||        35
user:~/Desktop$
```

II. MEMORY MANAGEMENT:

In modern computers and embedded systems there is typically a large amount of memory available. To be able to read and write to this memory, we need a way to access individual units of storage. Memory addresses is used that uniquely identify memory locations.

Reference Link:

https://kuleuven-diepenbeek.github.io/osc-course/exercises/

Program:

#include<stdio.h>

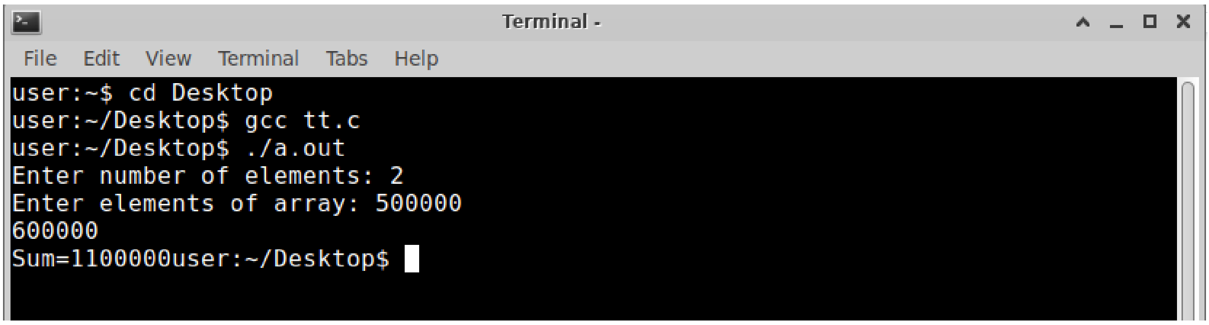
#include<stdlib.h>

int main()

{

```
        int n,i,*ptr,sum=0;

        printf("Enter number of elements: ");

        scanf("%d",&n);

        ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc

        if(ptr==NULL)

        {

                printf("Sorry! unable to allocate memory");

                exit(0);

        }

        printf("Enter elements of array: ");

        for(i=0;i<n;++i)

        {

                scanf("%d",ptr+i);

                sum+=*(ptr+i);

        }

        printf("Sum=%d",sum);

        free(ptr);

        return 0;

}
```

Output:



```
user:~$ cd Desktop
user:~/Desktop$ gcc tt.c
user:~/Desktop$ ./a.out
Enter number of elements: 2
Enter elements of array: 500000
600000
Sum=1100000user:~/Desktop$ 
```

Schedule:

Day/ Time	8:45 am to 9:35 am	9:35 am to 10:25 am	10:25 am to 10:40 am	10:40 am to 11:30 am	11:30 am to 12:20 pm	12:20 pm to 1:30 pm	1:30 pm to 2:20 pm	2:20 pm to 3:10 pm	3:10 pm to 3:25 pm	3:25 pm to 4:15 pm
06	Implementation of Scheduling algorithm		**Tea Break**	Implementation of Dynamic Memory		**Lunch Break**	Task		**Tea Break**	Assess -ment

Description of the task:

Write a C Program to implement priority scheduling using dynamic memory allocation.

Assessment Pattern:

Sl. No.	**Aspect Type (M or J)**	**Aspect of description**	**Extra aspect of description**	**Requirement (only for M)**	**Max. marks**
1.	J	Memory allocation for each process	Yes – **1** No – **0**	-	1
2.	J	Calculation of Burst time	Yes – **1** No – **0**	-	1
3.	J	Calculation of waiting time	Yes **- 1** No – **0**	-	1
4.	J	Calculation of Turnaround time	Yes – **1** No – **0**	-	1
5.	J	Displaying average waiting and turnaround time	Yes **- 1** No **- 0**	-	1
6.	J	Implementation of dynamic memory allocation	Yes **- 1** No **- 0**	-	1
7.	J	Displaying the output in table format	Yes **- 1** No **- 0**	-	1
8.	J	Displaying the output using Gantt chart	Yes **- 1** No **- 0**	-	1
9.	J	Time Management	**0 -** Exceeded 45 mins	-	1

			0.5 - Completed within 30 to 45 mins **1 -** Completed within 30 min		
10.	J	**Coding Ethics** Proper Indentation Overall design look	**0 -** If none of the aspect is found **0.5 -** If exhibiting any one aspect **1 -** If exhibiting all two aspects	-	1
Total Marks					**10**

Day 07	Multithreading and Interthread communication between threads

Objective:

To understand the basic concepts of multithreading, creating multi-thread applications and managing communication between the threads in Linux.

Outcomes:

To create an application for multiple threads and make interthread communication between threads using C programming.

Resources Required	**:**	Linux Virtual environment
Safety Precautions	**:**	Nil
Prerequisites	**:**	C programming

Theory:

I. THREAD

Reference Link:

https://kuleuven-diepenbeek.github.io/osc-course/exercises/

Pthread library and thread Function

1. Header file

Include the header file pthread.h.

```
#include <pthread.h>
```

2. The ID of a thread

Each thread has an object of type pthread_t associated with it that tells its ID. The same pthread_t object cannot be used by multiple threads simultaneously. For multiple threads, an array can be created where **pthread_t id[2];**each element is an ID for a separate thread:

```
pthread_t id[2];
```

3. Creating Pthread

A thread is created and starts using the function pthread_create(). It takes four parameters:

Name	Type	Description
ID	pthread_t *	Reference (or pointer) to the ID of the thread.
Attributes	pthread_attr_t *	Used to set the attributes of a thread(e.g., the stack size, scheduling policy, etc.) Passing NULL suffices for most applications.
Starting routine	void *	The name of the function that the thread starts to execute. If the function's return type is void *, then its name is simply written; otherwise, it has to be type-cast to void *.
Arguments	void *	This is the argument that the starting routine takes. If it takes multiple arguments, a struct is used.

The return type of a starting routine and its argument is usually set to void *.

```
        pthread_create(&id[0], NULL, printNumber, &arg);
```

4. Exiting a thread

pthread_exit() is used to exit a thread. This function is usually written at the end of the starting routine. If a value is returned by a thread upon ending, its reference is passed as an argument. Since a thread's local variables are destroyed when they exit, only references to global or dynamic variables are returned.

Program:

```
// Global variable:
int i = 1;
// Starting routine:
void* foo(void* p)
{
        int i = *(int*) p;
        printf("Received value: %i", i);
        pthread_exit(&i);    // Return reference to global variable
}
```

5. Waiting for a thread

A parent thread is made to wait for a child thread using pthread_join(). The two parameters of this function are:

Name	Type	Description
Thread ID	pthread_t	The ID of the thread that the parent thread waits for.
Reference to return value	void **	The value returned by the exiting thread is caught by this pointer.

```
        int* ptr;
        pthread_join(id, &ptr);
```

6. pthread Attributes

Threads can be assigned various thread attributes at the time of thread creation. This is controlled through the second argument to pthread_create(). First pass the pthread_attr_t variable through the following code.

```
        int pthread_attr_init(pthread_attr_t *attr);
```

7. Compiling Multithreaded Programs

To compile a multithreaded program using gcc, we need to link it with the pthreads library. Following is the command used to compile the program.

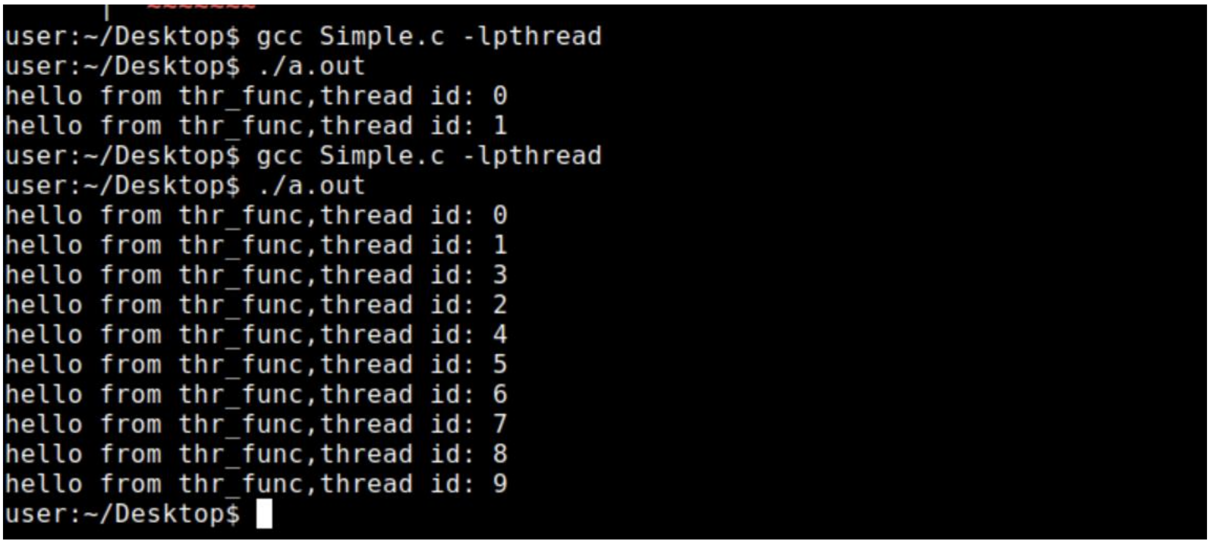
```
        gcc filename.c –lpthread  // gcc myProgram.o -o myProgram -lpthread
        ./a.out
```

Program 1: Simple Mutltithread Program

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
 #define NUM_THREADS 2
 typedef struct _thread_data_t
{
	int tid;
	double stuff;
} thread_data_t;
 void *thr_func(void *arg)
{
	thread_data_t *data = (thread_data_t *)arg;
	printf("hello from thr_func, thread id: %d\n", data->tid);
	pthread_exit(NULL);
}
 int main(int argc, char **argv)
{
	pthread_t thr[NUM_THREADS];
	int i, rc;
	thread_data_t thr_data[NUM_THREADS];
	for (i = 0; i < NUM_THREADS; ++i)
	{
		thr_data[i].tid = i;
		if ((rc = pthread_create(&thr[i], NULL, thr_func, &thr_data[i])))
		{	fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
```

```
                        return EXIT_FAILURE;                    }
              }
              for (i = 0; i < NUM_THREADS; ++i)
              {
                        pthread_join(thr[i], NULL);
              }
     return EXIT_SUCCESS;
}
```

Output:



```
user:~/Desktop$ gcc Simple.c -lpthread
user:~/Desktop$ ./a.out
hello from thr_func,thread id: 0
hello from thr_func,thread id: 1
user:~/Desktop$ gcc Simple.c -lpthread
user:~/Desktop$ ./a.out
hello from thr_func,thread id: 0
hello from thr_func,thread id: 1
hello from thr_func,thread id: 3
hello from thr_func,thread id: 2
hello from thr_func,thread id: 4
hello from thr_func,thread id: 5
hello from thr_func,thread id: 6
hello from thr_func,thread id: 7
hello from thr_func,thread id: 8
hello from thr_func,thread id: 9
user:~/Desktop$ █
```

Program description:

C-program that has 4 threads (in addition to the main thread). Each thread calculates all prime numbers between a lower and larger limit. Given a maximum number N which is divisible by 4 (e.g., 100000):

- Thread 1 computes all primes between 2 and N/4
- Thread 2 computes all primes between N/4 + 1 and N/2
- Thread 3 computes all primes between N/2 + 1 and 3N/4
- Thread 4 computes all primes between 3N/4 + 1 and N

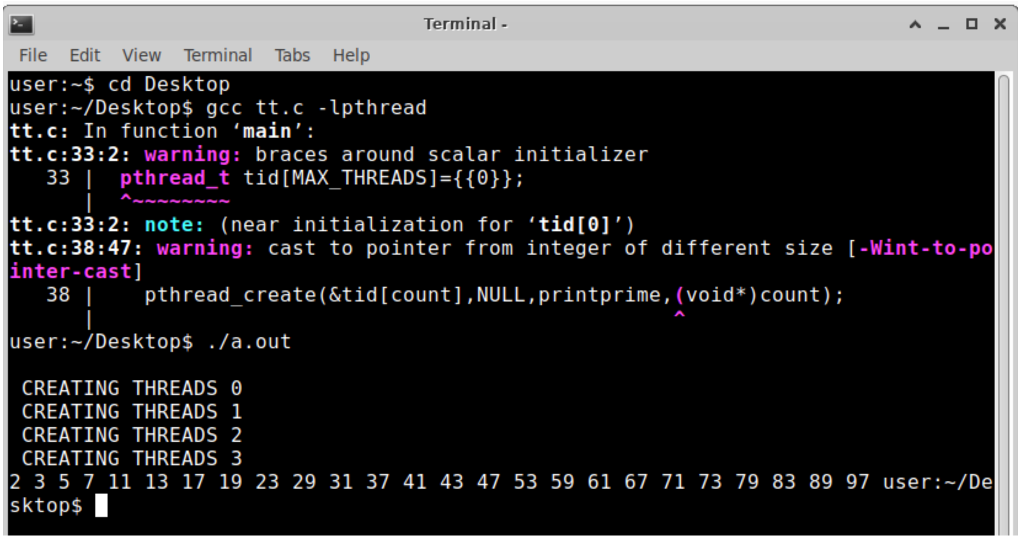
- Make sure you pass these limits properly to the threads as function parameters. Think about how you can pass more than 1 parameter!

Sample Code:

```
#include<stdio.h>
#include<pthread.h>
#define N 1000
#define MAX_THREADS 4
int prime_arr[N]={0};
void *printprime(void *ptr)
{
        int  j,flag;
        int i=(int)(long long int)ptr;
        for(i=2;i<N;i++)
        {
                flag=0;
                for(j=2;j<=i/2;j++)
                {
                        if(i%j==0)
                        {       flag=1;
                                break;          }
                }
                if(flag==0)
                {       prime_arr[i]=1;         }
        }
}
```

```
int main()
{
        pthread_t tid[MAX_THREADS]={{0}};
        int count=0;
        for(count=0;count<MAX_THREADS;count++)
        {       printf("\r\n CREATING THREADS %d",count);
                pthread_create(&tid[count],NULL,printprime,(void*)count);
        }
        printf("\n");
        for(count=0;count<MAX_THREADS;count++)
        {       pthread_join(tid[count],NULL);       }
        int c=0;
        for(count=0;count<N;count++)
        if(prime_arr[count]==1)
        printf("%d ",count);
        return 0;
}
```

Output:



```
user:~$ cd Desktop
user:~/Desktop$ gcc tt.c -lpthread
tt.c: In function ‘main’:
tt.c:33:2: warning: braces around scalar initializer
   33 |  pthread_t tid[MAX_THREADS]={{0}};
      |  ^~~~~~~~~
tt.c:33:2: note: (near initialization for ‘tid[0]’)
tt.c:38:47: warning: cast to pointer from integer of different size [-Wint-to-po
inter-cast]
   38 |    pthread_create(&tid[count],NULL,printprime,(void*)count);
      |                                               ^
user:~/Desktop$ ./a.out

 CREATING THREADS 0
 CREATING THREADS 1
 CREATING THREADS 2
 CREATING THREADS 3
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 user:~/De
sktop$ 
```

II. INTERTHREAD COMMUNICATION:

pthread Mutexes

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
```

The pthread_mutex_init() function requires a pthread_mutex_t variable to operate on as the first argument. Attributes for the mutex can be given through the second parameter. To specify default attributes, pass NULL as the second parameter. Alternatively, mutexes can be initialized to default values through a convenient macro rather than a function call:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

To perform mutex locking and unlocking, the pthreads provides the following functions:

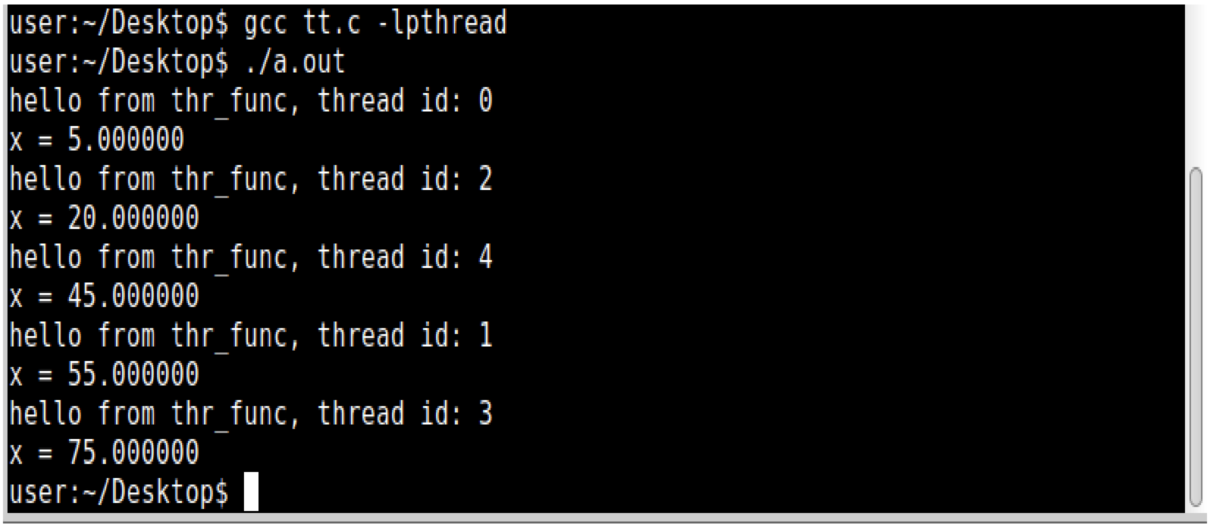
```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Program 1:

```
 #include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
 #define NUM_THREADS 5
 typedef struct _thread_data_t {
        int tid;
        double stuff;
} thread_data_t;
 double shared_x;
pthread_mutex_t lock_x;
```

```
 void *thr_func(void *arg)
{
        thread_data_t *data = (thread_data_t *)arg;
        printf("hello from thr_func, thread id: %d\n", data->tid);
        pthread_mutex_lock(&lock_x);
        shared_x += data->stuff;
        printf("x = %f\n", shared_x);
        pthread_mutex_unlock(&lock_x);
        pthread_exit(NULL);
}
 int main(int argc, char **argv)
{
        pthread_t thr[NUM_THREADS];
        int i, rc;
        thread_data_t thr_data[NUM_THREADS];
        shared_x = 0;
        pthread_mutex_init(&lock_x, NULL);
        for (i = 0; i < NUM_THREADS; ++i)
        {
                thr_data[i].tid = i;
                thr_data[i].stuff = (i + 1) * NUM_THREADS;
                if ((rc = pthread_create(&thr[i], NULL, thr_func, &thr_data[i])))
                {
                        fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
                        return EXIT_FAILURE;
                }
        }
        for (i = 0; i < NUM_THREADS; ++i)
        {
                pthread_join(thr[i], NULL);
        }
        return EXIT_SUCCESS;
}
```

Output:



```
user:~/Desktop$ gcc tt.c -lpthread
user:~/Desktop$ ./a.out
hello from thr_func, thread id: 0
x = 5.000000
hello from thr_func, thread id: 2
x = 20.000000
hello from thr_func, thread id: 4
x = 45.000000
hello from thr_func, thread id: 1
x = 55.000000
hello from thr_func, thread id: 3
x = 75.000000
user:~/Desktop$ 
```

Schedule:

Day/Time	8:45 am to 9:35 am	9:35 am to 10:25 am	10:25 am to 10:40 am	10:40 am to 11:30 am	11:30 am to 12:20 pm	12:20 pm to 1:30 pm	1:30 pm to 2:20 pm	2:20 pm to 3:10 pm	3:10 pm to 3:25 pm	3:25 pm to 4:15 pm
07	Implementation of Multithread		**Tea Break**	Implementation of Multithread & Task		**Lunch Break**	Communication between thread & Task		**Tea Break**	Assessment

Description of the task:

Task Name: Producer Consumer problem

We have two processes, producer and consumer, who share a fixed size buffer. Producer work is to produce data or items and put them in a buffer. Consumer work is to remove data from a buffer and consume it. We have to make sure that producers do not produce data when the buffer is full and consumers do not remove data when the buffer is empty.

Assessment Pattern:

Sl. No.	**Aspect Type (M or J)**	**Aspect of description**	**Extra aspect of description**	**Requirement (only for M)**	**Max. marks**
1.	J	Creation of new script file	Yes **- 1** No **- 0**	-	1
2.	J	Implementation of Thread	Yes **- 1** No – **0**	-	1
3.	J	Creating Pthread()	Yes – **1** No – **0**	-	1
4.	J	Create Thread ID	Yes – **1** No – **0**	-	1
5.	J	Create Producer function	Yes – **1** No – **0**	-	1
6.	J	Create Consumer Function	Yes – **1** No – **0**	-	1

7.	J	Join the Threads both producer and Consumer	Yes – **1** No – **0**	-	1
8.	J	Communication between Producer and Consumer	Yes – **1** No – **0**	-	1
9.	J	Show case output in Command Line	Yes – **1** No – **0**	-	1
10.	J	Time Management	Yes – **1** No – **0**	-	1
Total Marks					**10**

DAY 08	Data sharing between multiple processes using IPC mechanism

Objective:

The aim of the task is to provide adequate knowledge in communication of one process with another process using different mechanisms like PIPE, FIFO, Message Queue and Shared Memory.

Outcome:

At the end of the task, students can understand the fundamental concepts of Inter process Communication and can implement a C program for sharing messages between two processes.

Resources required:

- Linux OS
- C Compiler

Safety Precautions	**:**	Nil
Prerequisites	**:**	Nil

Theory:

INTER PROCESS COMMUNICATION

Inter Process Communication (IPC) is a mechanism that involves communication of one process with another process. This usually occurs only in one system.

1. Pipes

Communication between two related processes. The mechanism is half duplex meaning the first process communicates with the second process. To achieve a full duplex i.e., for the second process to communicate with the first process another pipe is required.

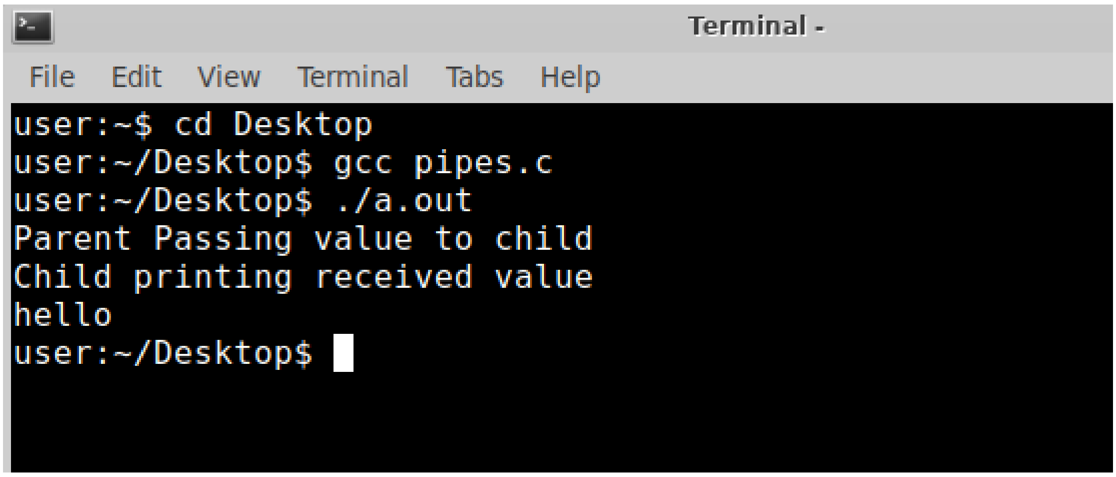
Reference Link:

https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_pipes.htm

Program:

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
int main()
{
        int fd[2],n;
        char buffer[100];
        pid_t p;
        pipe(fd); //creates a unidirectional pipe with two end fd[0] and fd[1]
        p=fork();
        if(p>0) //parent
        {
                printf("Parent Passing value to child\n");
                write(fd[1],"hello\n",6); //fd[1] is the write end of the pipe
                wait(NULL);
        }
        else // child
        {
                printf("Child printing received value\n");
                n=read(fd[0],buffer,100); //fd[0] is the read end of the pipe
                write(1,buffer,n);
        }
}
```

Output:



```
user:~$ cd Desktop
user:~/Desktop$ gcc pipes.c
user:~/Desktop$ ./a.out
Parent Passing value to child
Child printing received value
hello
user:~/Desktop$ 
```

(i) Two-way Communication Using Pipes:

Pipe communication is viewed as only one-way communication i.e., either the parent process writes and the child process reads or vice-versa but not both. However, what if both the parent and the child needs to write and read from the pipes simultaneously, the solution is a two-way communication using pipes. Two pipes are required to establish two-way communication.

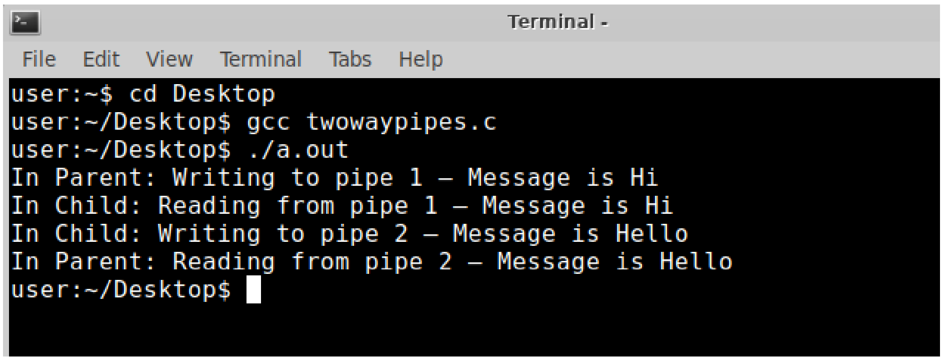
Program:

```
#include<stdio.h>

#include<unistd.h>

int main()

{

        int pipefds1[2], pipefds2[2];

        int returnstatus1, returnstatus2;

        int pid;

        char pipe1writemessage[20] = "Hi";

        char pipe2writemessage[20] = "Hello";

        char readmessage[20];

        returnstatus1 = pipe(pipefds1);
```

```
    if (returnstatus1 == -1)
    {
             printf("Unable to create pipe 1 \n");
             return 1;
     }
     returnstatus2 = pipe(pipefds2);
     if (returnstatus2 == -1) {
             printf("Unable to create pipe 2 \n");
             return 1;
     }
     pid = fork();
     if (pid != 0)
     { // Parent process
             close(pipefds1[0]);    // Close the unwanted pipe1 read side
             close(pipefds2[1]);    // Close the unwanted pipe2 write side
             printf("In Parent: Writing to pipe 1 – Message is %s\n",
             pipe1writemessage);
             write(pipefds1[1], pipe1writemessage, sizeof(pipe1writemessage));
             read(pipefds2[0], readmessage, sizeof(readmessage));
             printf("In Parent: Reading from pipe 2 – Message is %s\n",
             readmessage);
      }
else
{ //child process
       close(pipefds1[1]); // Close the unwanted pipe1 write side
       close(pipefds2[0]); // Close the unwanted pipe2 read side
       read(pipefds1[0], readmessage, sizeof(readmessage));
```

```
                printf("In Child: Reading from pipe 1 – Message is %s\n", readmessage);

                printf("In Child: Writing to pipe 2 – Message is %s\n", pipe2writemessage);

                write(pipefds2[1], pipe2writemessage, sizeof(pipe2writemessage));

        }

return 0;

}
```

Output:



```
user:~$ cd Desktop
user:~/Desktop$ gcc twowaypipes.c
user:~/Desktop$ ./a.out
In Parent: Writing to pipe 1 – Message is Hi
In Child: Reading from pipe 1 – Message is Hi
In Child: Writing to pipe 2 – Message is Hello
In Parent: Reading from pipe 2 – Message is Hello
user:~/Desktop$ 
```

2. FIFO (Named Pipes):

Communication between two unrelated processes. FIFO is a full duplex, meaning the first process can communicate with the second process and vice versa at the same time.

Link for FIFO :

https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_named_pipes.htm

Program: FIFO Server Code

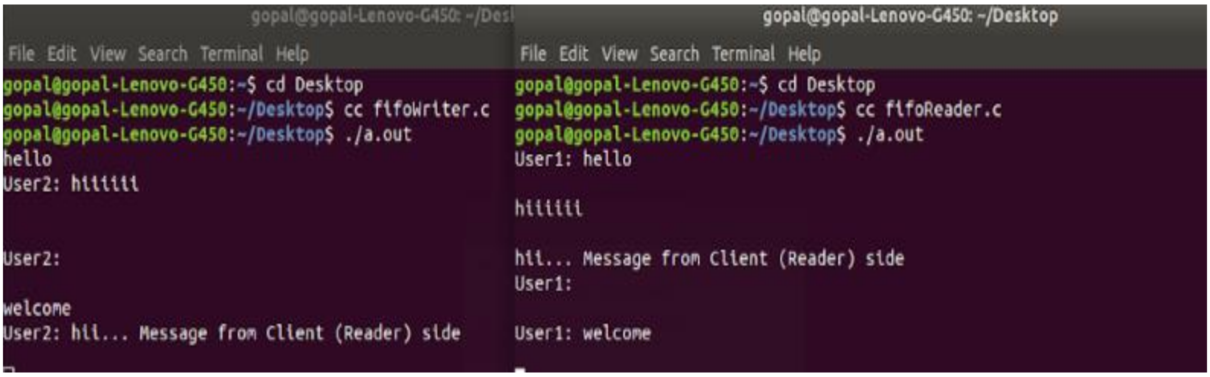
```
#include <stdio.h>

#include <string.h>

#include <fcntl.h>

#include <sys/stat.h>

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

```
#include <unistd.h>
int main()
{
        int fd;
        // FIFO file path
        char * myfifo = "/tmp/myfifo";
        // Creating the named file(FIFO)
        // mkfifo(<pathname>, <permission>)
        mkfifo(myfifo, 0666);
        char arr1[80], arr2[80];
        while (1)
        {
                // Open FIFO for write only
                fd = open(myfifo, O_WRONLY);
                // Take an input arr2ing from user.
                // 80 is maximum length
                fgets(arr2, 80, stdin);
                // Write the input arr2ing on FIFO and close it
                write(fd, arr2, strlen(arr2)+1);
                close(fd);
                 // Open FIFO for Read only
                fd = open(myfifo, O_RDONLY);
                // Read from FIFO
                read(fd, arr1, sizeof(arr1));
                // Print the read message
                printf("User2: %s\n", arr1);
                close(fd);
        }
return 0;
}
```

Program: FIFO Client Code

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
        int fd1;
        // FIFO file path
        char * myfifo = "/tmp/myfifo";
        // Creating the named file(FIFO)
        // mkfifo(<pathname>,<permission>)
        mkfifo(myfifo, 0666);
        char str1[80], str2[80];
         while (1)
        {
                // First open in read only and read
                fd1 = open(myfifo,O_RDONLY);
                read(fd1, str1, 80);
                // Print the read string and close
                printf("User1: %s\n", str1);
                close(fd1);
                // Now open in write mode and write
                // string taken from user.
                fd1 = open(myfifo,O_WRONLY);
                fgets(str2, 80, stdin);
                write(fd1, str2, strlen(str2)+1);
                close(fd1);
        }
return 0;
}
```

Output:


gopal@gopal-Lenovo-G450: ~/Desl
File Edit View Search Terminal Help
gopal@gopal-Lenovo-G450: ~/Desktop
File Edit View Search Terminal Help

```
gopal@gopal-Lenovo-G450:~$ cd Desktop
gopal@gopal-Lenovo-G450:~/Desktop$ cc fifoWriter.c
gopal@gopal-Lenovo-G450:~/Desktop$ ./a.out
hello
User2: hiiiiii


User2:

welcome
User2: hii... Message from Client (Reader) side
```

```
gopal@gopal-Lenovo-G450:~$ cd Desktop
gopal@gopal-Lenovo-G450:~/Desktop$ cc fifoReader.c
gopal@gopal-Lenovo-G450:~/Desktop$ ./a.out
User1: hello

hiiiiii

hii... Message from Client (Reader) side
User1:

User1: welcome
```

3. Message Queues

Communication between two or more processes with full duplex capacity. The processes will communicate with each other by posting a message and retrieving it out of the queue. Once retrieved, the message is no longer available in the queue.

Reference Link:

https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_message_queues.htm

Program: Sender code

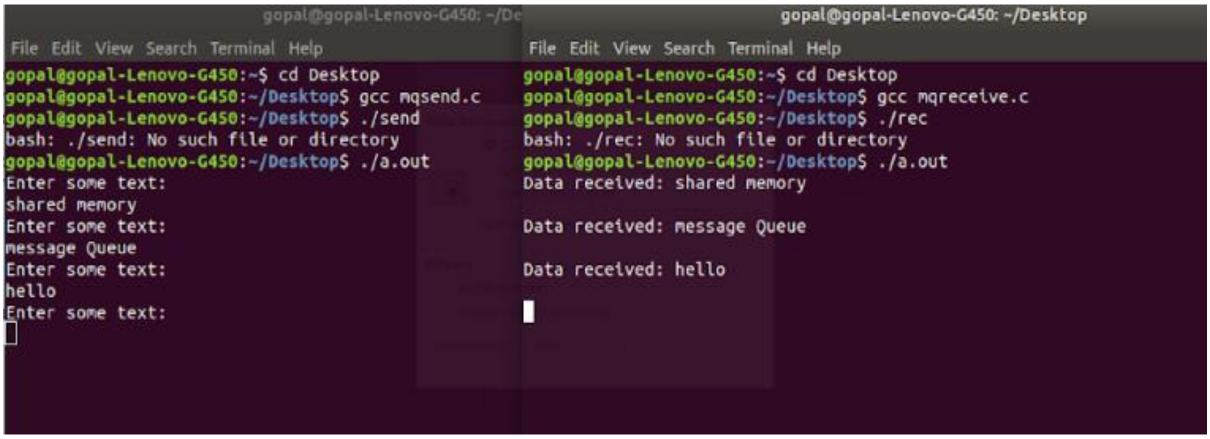
```
#include<stdlib.h>
#include<stdio.h>
 #include<string.h>
 #include<unistd.h>
 #include<sys/types.h>
 #include<sys/ipc.h>
 #include<sys/msg.h>
 #define MAX_TEXT 512   //maximum length of the message that can be sent allowed
 struct my_msg{
        long int msg_type;
        char some_text[MAX_TEXT];
 };
 int main()
 {
        int running=1;
        int msgid;
        struct my_msg some_data;
        char buffer[50]; //array to store user input
        msgid=msgget((key_t)14534,0666|IPC_CREAT);
        if (msgid == -1) // -1 means the message queue is not created
        {
                printf("Error in creating queue\n");
                exit(0);
      }
```

```
        while(running)
        {
                printf("Enter some text:\n");
                fgets(buffer,50,stdin);
                some_data.msg_type=1;
                strcpy(some_data.some_text,buffer);
                if(msgsnd(msgid,(void *)&some_data, MAX_TEXT,0)==-1)
                // msgsnd returns -1 if the message is not sent
                {
                        printf("Msg not sent\n");
                }
         if(strncmp(buffer,"end",3)==0)
         {      running=0;              }
        }
 }
```

Program: Receiver code

```
#include<stdlib.h>
 #include<stdio.h>
 #include<string.h>
 #include<unistd.h>
 #include<sys/types.h>
 #include<sys/ipc.h>
 #include<sys/msg.h>
 struct my_msg{
        long int msg_type;
        char some_text[BUFSIZ];
 };
int main()
{
        int running=1;
        int msgid;
        struct my_msg some_data;
        long int msg_to_rec=0;
        msgid=msgget((key_t)14534,0666|IPC_CREAT);
        while(running)
        {
                msgrcv(msgid,(void *)&some_data,BUFSIZ,msg_to_rec,0);
                printf("Data received: %s\n",some_data.some_text);
                if(strncmp(some_data.some_text,"end",3)==0)
                {               running=0;                              }
        }
 msgctl(msgid,IPC_RMID,0);
 }
```

Output:


gopal@gopal-Lenovo-G450: ~/De
File Edit View Search Terminal Help
gopal@gopal-Lenovo-G450: ~/Desktop
File Edit View Search Terminal Help

```
gopal@gopal-Lenovo-G450:~$ cd Desktop
gopal@gopal-Lenovo-G450:~/Desktop$ gcc mqsend.c
gopal@gopal-Lenovo-G450:~/Desktop$ ./send
bash: ./send: No such file or directory
gopal@gopal-Lenovo-G450:~/Desktop$ ./a.out
Enter some text:
shared memory
Enter some text:
message Queue
Enter some text:
hello
Enter some text:
```

```
gopal@gopal-Lenovo-G450:~$ cd Desktop
gopal@gopal-Lenovo-G450:~/Desktop$ gcc mqreceive.c
gopal@gopal-Lenovo-G450:~/Desktop$ ./rec
bash: ./rec: No such file or directory
gopal@gopal-Lenovo-G450:~/Desktop$ ./a.out
Data received: shared memory

Data received: message Queue

Data received: hello
```

4. Shared Memory

Communication between two or more processes is achieved through a shared piece of memory among all processes. The shared memory needs to be protected from each other by synchronizing access to all the processes.

Resource Link :

https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_shared_memory.htm

Program: Writer code

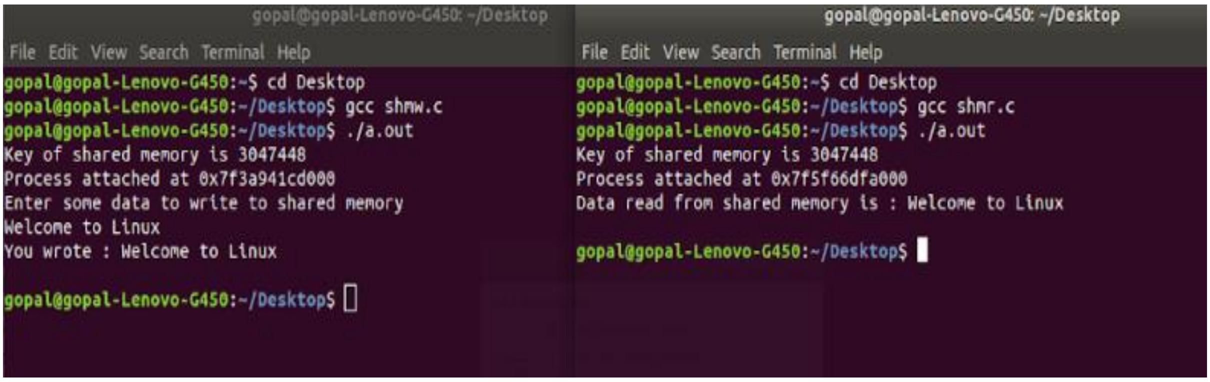
```
#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/shm.h>

#include<string.h>

int main()

{

        int i;

        void *shared_memory;

        char buff[100];

        int shmid;

        shmid=shmget((key_t)2345, 1024, 0666|IPC_CREAT);

        /* creates shared memory segment with key 2345, having size 1024 bytes. IPC_CREAT
        is used to create the shared segment if it does not exist. 0666 are the permisions on the
        shared segment */

        printf("Key of shared memory is %d\n",shmid);

        shared_memory=shmat(shmid,NULL,0); //process attached to shared memory segment

        printf("Process attached at %p\n",shared_memory);
```

```
    //this prints the address where the segment is attached with this process
    printf("Enter some data to write to shared memory\n");
    read(0,buff,100); //get some input from user
    strcpy(shared_memory,buff); //data written to shared memory
    printf("You wrote : %s\n",(char *)shared_memory);
}
```

Program: Reader code

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
    int i;
    void *shared_memory;
    char buff[100];
    int shmid;
    shmid=shmget((key_t)2345, 1024, 0666);
    printf("Key of shared memory is %d\n",shmid);
    shared_memory=shmat(shmid,NULL,0); //process attached to shared memory segment
    printf("Process attached at %p\n",shared_memory);
    printf("Data read from shared memory is : %s\n",(char *)shared_memory);
}
```

Output:



gopal@gopal-Lenovo-G450: ~/Desktop
File Edit View Search Terminal Help

```
gopal@gopal-Lenovo-G450:~$ cd Desktop
gopal@gopal-Lenovo-G450:~/Desktop$ gcc shmw.c
gopal@gopal-Lenovo-G450:~/Desktop$ ./a.out
Key of shared memory is 3047448
Process attached at 0x7f3a941cd000
Enter some data to write to shared memory
Welcome to Linux
You wrote : Welcome to Linux

gopal@gopal-Lenovo-G450:~/Desktop$
```

gopal@gopal-Lenovo-G450: ~/Desktop
File Edit View Search Terminal Help

```
gopal@gopal-Lenovo-G450:~$ cd Desktop
gopal@gopal-Lenovo-G450:~/Desktop$ gcc shmr.c
gopal@gopal-Lenovo-G450:~/Desktop$ ./a.out
Key of shared memory is 3047448
Process attached at 0x7f5f66dfa000
Data read from shared memory is : Welcome to Linux

gopal@gopal-Lenovo-G450:~/Desktop$
```

Schedule

Day/ Time	8:45 am to 9:35 am	9:35 am to 10:25 am	10:25 am to 10:40 am	10:40 am to 11:30 am	11:30 am to 12:20 pm	12:20 pm to 1:30 pm	1:30 pm to 2:20 pm	2:20 pm to 3:10 pm	3:10 pm to 3:25 pm	3:25 pm to 4:15 pm
08	Understand the basic IPC and Implementation of PIPE Concept		**Tea Break**	Implementation of FIFO and Message Queue		**Lunch Break**	Implementation of Shared Memory		**Tea Break**	Assessment

Description of the task:

Task 1: Implement a program to PIPE for one-way and two-way communication

Task 2: Implementation of IPC using FIFO (Named PIPE'S) between two processes

Task 3: Implementation of IPC using Message queue

Task 4: Implementation of IPC using Shared Memory

Assessment Pattern:

Sl. No.	**Aspect Type (M or J)**	**Aspect of description**	**Extra aspect of description**	**Requirement (only for M)**	**Max. marks**
1.	J	**Declaration of file path**	Yes **- 1** No **- 0**	-	1
2.	J	**Implementation of PIPE** **a. One-way** **a. Two-way**	One-way Implemented- 0.5 Two-way Implemented- 0.5 No - 0	-	1
3.	J	**Implementation of FIFO** **a. Client program** **a. Server program**	Client only - 0.5 Server only - 0.5 No - 0	-	1
4.	J	**Implementation of MessageQueue** **a. Client program** **a. Server program**	Client only - 0.5 Server only - 0.5 No - 0	-	1

5.	J	**Implementation of Shared Memory** **a. Client program** **a. Server program**	Client only - 0.5 Server only - 0.5 No - 0	-	1
6.	J	**Display the output of PIPE**	Yes - 1 No - 0	-	1
7.	J	**Display the output of FIFO**	Yes - 1 No - 0	-	1
8.	J	**Display the output MessageQueue**	Yes - 1 No - 0	-	1
9.	J	**Display the output of Shared Memory**	Yes - 1 No - 0	-	1
10.	J	**Time Management and Code Ethics**	Time management – 0.5 Code Ethics – 0.5 No – 0	-	1
Total Marks					**10**

Day 09	Linux Networking commands – Developing Client server-based networking application

Objective:

To understand the concepts of networking in Linux environment.

Outcomes:

To maintain and troubleshoot the network(s) connected to the system and develop client server-based application.

Resources Required	**:**	Computer system
Safety Precautions	**:**	Nil
Prerequisites	**:**	

- Basic shell commands
- Dir & File Commands
- System Commands
- Misc Commands

Theory:

- Linux is used both in software development and in servers. Most of the devices and embedded systems in the world implement one or the other distribution of Linux.
- Networking in computers speaks about networking both within the network and across the internet. A network can be as small and simple as a home network or as complex as a network for a space station.
- Networking includes network configuration and troubleshooting.

I. LINUX NETWORKING COMMANDS

Linux networking commands are used extensively to inspect, analyze, maintain, and troubleshoot the network connected to the system.

(1) ifconfig

Linux ifconfig stands for interface configurator. It is one of the most basic commands used in network inspection.

ifconfig is used to initialize an interface, configure it with an IP address, and enable or disable it. It is also used to display the route and the network interface.

Basic information displayed upon using ifconfig are:

1. IP address
2. MAC address
3. MTU (Maximum Transmission Unit)

Syntax : Ifconfig

Example : ifconfig lo

(2) ip

This is the latest and updated version of the ifconfig command.

Syntax

1. ip a
2. ip addr

(3) raceroute

Linux traceroute is one of the most useful commands in networking. It is used to troubleshoot the network. It detects the delay and determines the pathway to your target.

It basically helps in the following ways:

1. It provides the names and identifies every device on the path.
2. It follows the route to the destination

3. It determines where the network latency comes from and reports it

Syntax:

```
traceroute <destination>
```

Example:

```
$ traceroute google.com
```

(4) tracepath

- Linux tracepath is similar to traceroute command. It is used to detect network delays. However, it doesn't require root privileges.
- It is installed in Ubuntu by default.
- It traces the route to the specified destination and identifies each hop in it. If your network is weak, it recognizes the point where the network is weak.

Syntax:

```
tracepath <destination>
```

Example:

```
tracepath mindmajix.com
```

(5) ping

Linux ping is one of the most used network troubleshooting commands. It basically checks for the network connectivity between two nodes.

ping stands for Packet INternet Groper.

The ping command sends the ICMP echo request to check the network connectivity.

It keeps executing until it is interrupted.

Syntax:

```
ping <destination>
```

Example:

```
$ ping google.comnetstat
```

(6) netstat

Linux netstat command refers to the network statistics.

It provides statistical figures about different interfaces which include open sockets, routing tables, and connection information.

Syntax:

```
netstat
```

Example:

```
netstat -p
netstat -r
```

(7) SS

Linux ss command is the replacement for netstat command. It is regarded as a much faster and more informative command than netstat.

The faster response of ss is possible as it fetches all the information from within the kernel userspace.

Syntax:

```
ss
```

This command gives information about all TCP, UDP, and UNIX socket connections.

You can use -t, -u, -x in the command respectively to show TCP/UDP or UNIX sockets.

(8) nslookup

Linux nslookup is also a command used for DNS related queries. It is the older version of dig.

Syntax:

```
nslookup <domainName>
```

Example:

```
nslookup mindmajix.com route
```

(9) route

Linux route command displays and manipulates the routing table existing for your system.

A router is basically used to find the best way to send the packets across to a destination.

Syntax:

```
route host
```

Linux host command displays the domain name for a given IP address and IP address for a given hostname. It is also used to fetch DNS lookup for DNS related query.

Example:

```
host mindmajix.com

host 149.77.21.18
```

(10) arp

Linux arp command stands for Address Resolution Protocol. It is used to view and add content to the kernel's ARP table.

Syntax:

```
arp
```

(11) hostname

Linux hostname is the simple command used to view and set the hostname of a system.

Syntax:

```
hostname
```

(12) curl and wget

Linux curl and wget commands are used in downloading files from the internet through CLI. The curl command has to be used with the option "O" to fetch the file, while the wget command is used directly.

Below are the syntax and the example for the two commands.

Example:

```
curl -O google.com/doodles/childrens-day-2014-multiple-countries

wget google.com/doodles/new-years-day-2012
```

(13) whois

Linux whois command is used to fetch all the information related to a website. You can get all the information about a website including the registration and the owner information.

Example:

```
whois mindmajix.com
```

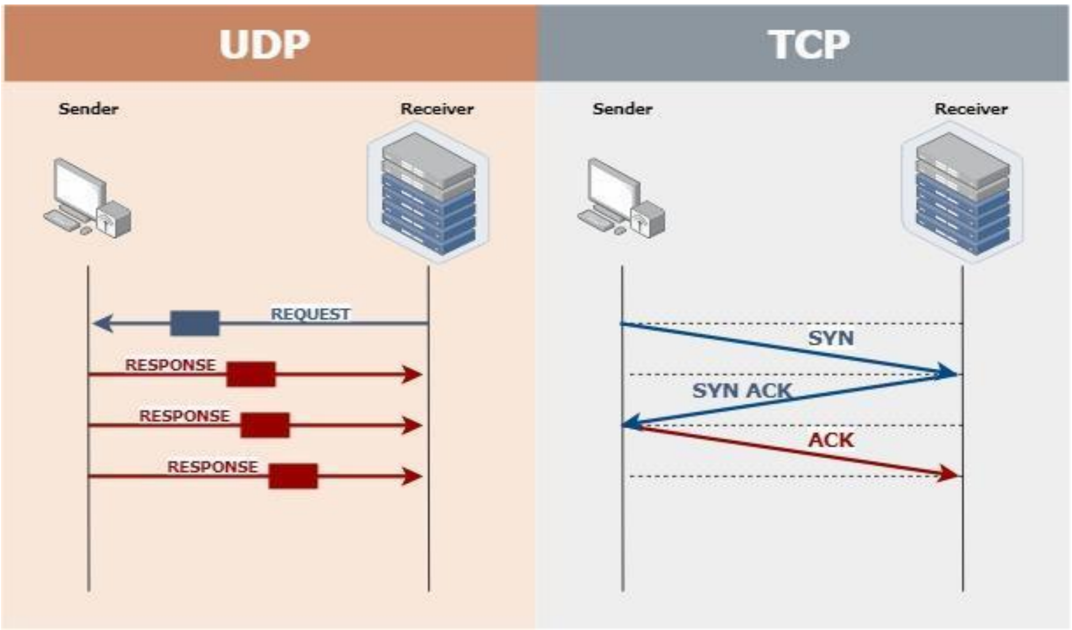
II. TCP AND UDP PROTOCOLS

TCP

The TCP stands for **Transmission Control Protocol**. TCP/IP is a commonly used standard for transmitting data over networks.

UDP

The UDP stands for **User Datagram Protocol**. Its working is similar to the TCP as it is also used for sending and receiving the message. The main difference is that UDP is a connectionless protocol.



1. Developing Client Server Application in Linux

Most of the Net Applications use the Client-Server architecture, which refers to two processes or two applications that communicate with each other to exchange some information. One of the two processes acts as a client process, and another process acts as a server.

Client Process

This is the process, which typically makes a request for information. After getting the response, this process may terminate or may do some other processing.

Server Process

This is the process which takes a request from the clients. After getting a request from the client, this process will perform the required processing, gather the requested information, and send it to the requestor client. Once done, it becomes ready to serve another client. Server processes are always alert and ready to serve incoming requests.

2. The entire process can be broken down into following steps:

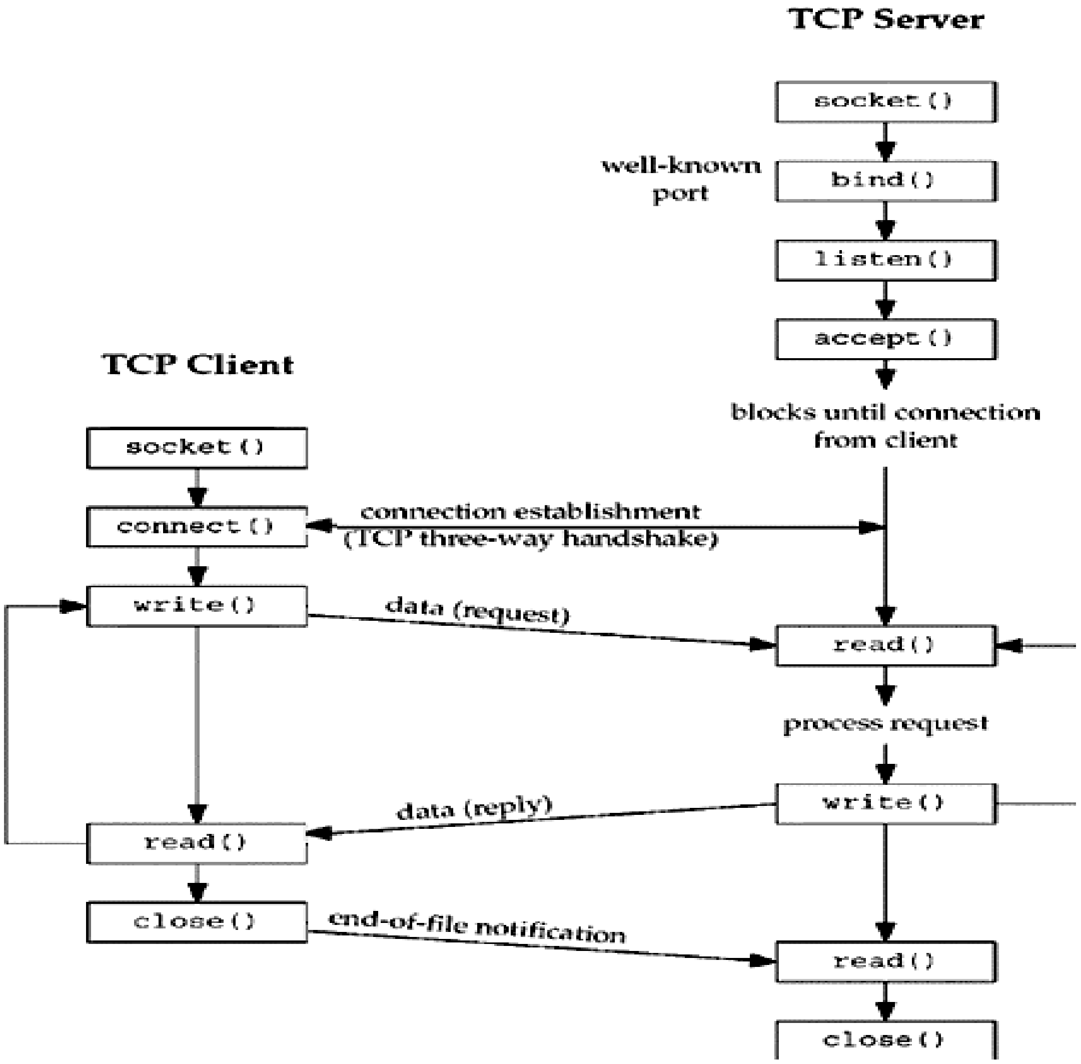
TCP Server:

1. using create(), Create TCP socket.
2. using bind(), Bind the socket to the server address.

3. using listen(), put the server socket in a passive mode, where it waits for the client to approach the server to make a connection
4. using accept(), At this point, connection is established between client and server, and they are ready to transfer data.
5. Go back to Step 3.

TCP Client:

1. Create a TCP socket.
2. Connect the newly created client socket to server.



TCP Server:

```
#include <stdio.h>

#include <netdb.h>

#include <netinet/in.h>

#include <stdlib.h>

#include <string.h>

#include <sys/socket.h>

#include <sys/types.h>

#define MAX 80

#define PORT 8080

#define SA struct sockaddr

// Function designed for chat between client and server.

void func(int connfd)

{

        char buff[MAX];

        int n;

        // infinite loop for chat

        for (;;) {

                bzero(buff, MAX);

                // read the message from client and copy it in buffer

                read(connfd, buff, sizeof(buff));

                // print buffer which contains the client contents

                printf("From client: %s\t To client : ", buff);

                bzero(buff, MAX);

                n = 0;

                // copy server message in the buffer
```

```
		while ((buff[n++] = getchar()) != '\n')

		;

		// and send that buffer to client

		write(connfd, buff, sizeof(buff));

		// if msg contains "Exit" then server exit and chat ended.

		if (strncmp("exit", buff, 4) == 0) {

			printf("Server Exit...\n");

			break;

		}

	}

}


// Driver function

int main()

{

	int sockfd, connfd, len;

	struct sockaddr_in servaddr, cli;

	// socket create and verification

	sockfd = socket(AF_INET, SOCK_STREAM, 0);

	if (sockfd == -1) {

		printf("socket creation failed...\n");

		exit(0);

	}

	else

		printf("Socket successfully created..\n");

	bzero(&servaddr, sizeof(servaddr));
```

```
// assign IP, PORT
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(PORT);
// Binding newly created socket to given IP and verification
if ((bind(sockfd, (SA*)&servaddr, sizeof(servaddr))) != 0) {
        printf("socket bind failed...\n");
        exit(0);
}
else
        printf("Socket successfully binded..\n");
// Now server is ready to listen and verification
if ((listen(sockfd, 5)) != 0) {
        printf("Listen failed...\n");
        exit(0);
}
else
          printf("Server listening..\n");
len = sizeof(cli);
// Accept the data packet from client and verification
connfd = accept(sockfd, (SA*)&cli, &len);
if (connfd < 0) {
        printf("server accept failed...\n");
        exit(0);
}
else
```

```
        printf("server accept the client...\n");
    // Function for chatting between client and server
    func(connfd);
    // After chatting close the socket
    close(sockfd);
}
```

TCP Client:

```
#include <arpa/inet.h> // inet_addr()
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h> // bzero()
#include <sys/socket.h>
#include <unistd.h> // read(), write(), close()
#define MAX 80
#define PORT 8080
#define SA struct sockaddr
void func(int sockfd)
{
    char buff[MAX];
    int n;
    for (;;) {
        bzero(buff, sizeof(buff));
        printf("Enter the string : ");
```

```
                n = 0;
                 while ((buff[n++] = getchar()) != '\n')
                ;
                 write(sockfd, buff, sizeof(buff));
                bzero(buff, sizeof(buff));
                read(sockfd, buff, sizeof(buff));
                printf("From Server : %s", buff);
                if ((strncmp(buff, "exit", 4)) == 0) {
                        printf("Client Exit...\n");
                        break;
                 }
        }
}

int main()
{
         int sockfd, connfd;
         struct sockaddr_in servaddr, cli;
         // socket create and verification
         sockfd = socket(AF_INET, SOCK_STREAM, 0);
         if (sockfd == -1) {
                 printf("socket creation failed...\n");
                 exit(0);
        }
        else
                 printf("Socket successfully created..\n");
        bzero(&servaddr, sizeof(servaddr));
         // assign IP, PORT
         servaddr.sin_family = AF_INET;
        servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
        servaddr.sin_port = htons(PORT);
         // connect the client socket to server socket
         if (connect(sockfd, (SA*)&servaddr, sizeof(servaddr)) != 0) {
                printf("connection with the server failed...\n");
                exit(0);
        }
        else
                printf("connected to the server..\n");
        // function for chat
        func(sockfd);
        // close the socket
        close(sockfd);
}
```

Compilation:

Server side:

```
gcc server.c -o server
./server
```

Client side:

```
gcc client.c -o client
./client
```

Output :

```
Server side:

Socket successfully created..

Socket successfully binded..

Server listening..

server accept the client...

From client: hi

     To client : hello

From client: exit

     To client : exit

Server Exit...

Client side:

 Socket successfully created..

connected to the server..

Enter the string : hi

From Server : hello

Enter the string : exit

From Server : exit

Client Exit...
```

Schedule:

Day/ Time	8:45 am to 9:35 am	9:35 am to 10:25 am	10:25 am to 10:40 am	10:40 am to 11:30 am	11:30 am to 12:20 pm	12:20 pm to 1:30 pm	1:30 pm to 2:20 pm	2:20 pm to 3:10 pm	3:10 pm to 3:25 pm	3:25 pm to 4:15 pm
09	Basic Linux Networking commands		**Tea Break**	Establishing Socket connection		**Lunch Break**	Developing Client server applications		**Tea Break**	Task Assessment

Description of the task:

Task 1: Write a program to perform basic networking operations such as finding the IP address, network delay etc.

Task 2: Write a Shell script to establish socket connection.

Task 3: Write a Shell script to demonstrate Chat application

Assessment Pattern:

Sl. No.	**Aspect Type (M or J)**	**Aspect of description**	**Extra aspect of description**	**Requirement (only for M)**	**Max. marks**
1.	J	Creation of new script file	Yes – **0.5** No **- 0**	-	0.5
2.	J	Find IP Address	Yes **- 1** No **- 0**	-	1
3.	J	Demonstrate Ping and traceroute	Yes – **0.5** No **- 0**	-	0.5
4.	J	Display domain name for given IP address	Yes **- 1** No **- 0**	-	1
5.	J	Shows the working of TCP UDP protocols	Yes **- 1** No **- 0**	-	1
6.	J	Implementation of Socket	Yes **- 1** No **- 0**	-	1
7.	J	Connect newly created client socket to server	Yes **- 1** No **- 0**	-	1

8.	J	Shell script to Implement TCP client	Yes **- 2** No **- 0**	-	2
9.	J	Shell script to Implement TCP server	Yes **- 2** No **- 0**	-	2
Total Marks					**10**

TEST PROJECT

Chat Application Using C Program

Description of Task:

Creating a chat application with the implementation of Multithread, signals and dynamic memory allocation and make a communication between the n number of clients using C Programming.

Assessment Pattern:

S.No	Aspect Type (M or J)	Aspect Description	Extra Aspect of Description	Requirement M only	Maximum score (50)
Work organization and management					
1	J	Logic of the program	0.5 marks - If imported less than the required 1 mark If imported all required libraries	-	2
2	M	Time management	**1 mark -** In time **0 mark -** Out of time	-	2
3	J	Code ethics (i)Indentation (ii)Overall design Look	**0 mark -** not suited **1 mark -** Well suited	-	1
Sub-Total					**5**
Create Server					
6	J	Implement necessary header files	**0 marks -** If not added **2 mark-** If added	-	2
7	M	Add Client	**0 marks -** If not Created	-	2

			2 mark- If created		
8	J	Delete Client	**0 marks -** If not deleted **2 mark-** If deleted	-	2
9	J	Delete Client List	**0 marks -** If not deleted **2 mark-** If deleted	-	2
10	J	Send message to all client	**0 marks -** Message not reached to client **2 mark-** Message reached to client	-	2
11	J	Display the messages send by client	**2 mark**– Messages Displayed **0 mark-** Message not Displayed	-	2
12	J	Create Server Socket	**2 mark**– If Socket Created **0 mark-** If Socket Not Created	-	2

13	J	Create Port Number	**0 marks**
			if not implemented
			2 mark
			if implemented
				-	2

14	J	Display the Client list added in the client	**0 marks**
			if port not Enabled
			2 mark
			if port number Enabled
				-	2

15	J	Quit all the Client using Single Command	**0 marks**
			if not implemented
			2 marks
			if implemented
				-	2

16	J	Server execute Successfully	If yes 3 Execute success but not wait for the client 2 Not execute 0	-	3
Sub-Total					**23**

Create Client					
16	M	Dynamic Memory	**0 marks-** if not Created **2 marks**- if Created	-	2
17	J	Create Multi Thread	**marks-** if not Thread implemented **2 marks**- if Thread Implemented	-	2
18	J	Join Threads	**0 marks** if not joined **2 marks** if not joined	-	2
19	J	Create Client Socket	**0 marks-** if not Created **2 marks**- if Created	-	2
20	J	Chat Read from Server	**0 marks-** if not Created **2 marks**- if Created	-	2
21	J	Write Chat to the Server	**marks-** if not implemented **2 marks**- if Implemented	-	2
22	J	Unexpected Server Error Handling	**0 marks** if error not handled **2 marks** if error handled	-	2
23	J	Execute the Client Program	**2 marks** If correctly executed and connected with server **0 mark** if not executed and connected with server	-	2
24	M	Create Multiple Clients	**0 marks-** if not Created **2 marks**- if Created multiple clients	-	2
25	J	Communicate Between Client Through Server	**0 marks-** if not Created **2 marks**- if Created and communicated with different	-	2

			clients		
26	J	Client quit from server	**0 marks-** if not quit properly **2 marks-** if client quits from server		2
Sub – Total					**22**
TOTAL MARKS					**50**