



Parul University

FACULTY OF ENGINEERING AND TECHNOLOGY
BACHELOR OF TECHNOLOGY

OPERATING SYSTEM

(203105214)

4th SEMESTER

COMPUTER SCIENCE & ENGINEERING DEPARTMENT

Lab Manual

CERTIFICATE

This is to certify that **Jenish Ashokbhai Bambhroliya**
with Enrollment no. **2303031260020** has successfully
completed his laboratory experiments in **OPERATING**
SYSTEMS(203105214) from the department of **COMPUTER**
SCIENCE AND ENGINEERING during the academic year
2024-25



Date of Submission:

Staff In charge:

Head of Department:

PARUL UNIVERSITY
FACULTY OF ENGINEERING & TECHNOLOGY
Operating System (203105214) B. Tech. 2nd Year

S.N O	Name of Practical	Page No.	Perfor- mance date	Assess- ment date	Marks	Sign
1.	Study of Basic commands of Linux.					
2.	Study the basics of shell programming.					
3.	Write a Shell script to print given numbers sum of all digits					
4.	Write a shell script to validate the entered date. (eg. Date format is: dd-mm-yyyy).					
5.	Write a shell script to check entered string is palindrome or not.					
6.	Write a Shell script to say Good morning/Afternoon/Evening as you log in to system.					
7.	Write a C program to create a child process					

PARUL UNIVERSITY
FACULTY OF ENGINEERING & TECHNOLOGY
Operating System (203105214) B. Tech. 2nd Year

8.	Finding out biggest number from given three numbers supplied as command line arguments					
9.	Printing the patterns using for loop.					
10.	Shell script to determine whether given file exist or not.					
11.	Write a program for process creation using C. (Use of gcc compiler).					
12.	Implementation of FCFS & Round Robin Algorithm.					
13.	Implementation of Banker's Algorithm.					

S

List of Practicals:

1. Study of Basic commands of Linux.
2. Study the basics of shell programming.
3. Write a Shell script to print given numbers sum of all digits.
4. Write a shell script to validate the entered date. (eg. Date format is: dd-mm-yyyy).
5. Write a shell script to check entered string is palindrome or not.
6. Write a Shell script to say Good morning/Afternoon/Evening as you log in to system.
7. Write a C program to create a child process
8. Finding out biggest number from given three numbers supplied as command line arguments
9. Printing the patterns using for loop.
10. Shell script to determine whether given file exist or not.
11. Write a program for process creation using C. (Use of gcc compiler).
12. Implementation of FCFS & Round Robin Algorithm.
13. Implementation of Banker's Algorithm.

PRACTICAL NO: 1

Definition: Study of Basic commands of Linux/UNIX.

Command shell: A program that interprets commands is Command shell.

Shell Script: Allows a user to execute commands by typing them manually at a terminal, or automatically in programs called shell scripts.

A shell is not an operating system. It is a way to interface with the operating system and run Commands.

BASH (Bourne Again Shell)

- Bash is a shell written as a free replacement to the standard Bourne Shell (/bin/sh) originally written by Steve Bourne for UNIX systems.
- It has all of the features of the original Bourne Shell, plus additions that make it easier to program with and use from the command line.
- Since it is Free Software, it has been adopted as the default shell on most Linux systems.

BASIC LINUX COMMANDS:

1. pwd : Print Working Directory

DESCRIPTION:

pwd prints the full pathname of the current working directory.

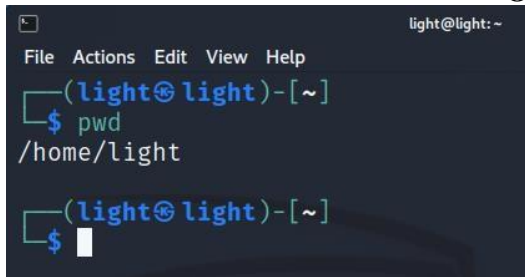
SYNTAX:

Pwd

EXAMPLE:

\$ pwd

/home/directory_name

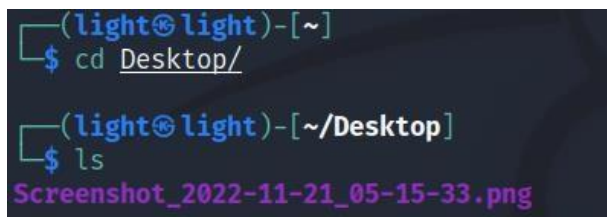
A terminal window with a dark background. The title bar shows a window icon and the text 'light@light: ~'. The menu bar contains 'File', 'Actions', 'Edit', 'View', and 'Help'. The prompt is '(light@light)-[~]'. The user enters '\$ pwd' and the output is '/home/light'. The prompt returns to '(light@light)-[~]' with a cursor on the next line.

```
light@light: ~  
File Actions Edit View Help  
(light@light)-[~]  
$ pwd  
/home/light  
(light@light)-[~]  
$
```

2. cd: Change Directory

DESCRIPTION:

It allows you to change your working directory. You use it to move around within the hierarchy of your file system.

A terminal window with a dark background. The title bar shows a window icon and the text 'light@light: ~'. The menu bar contains 'File', 'Actions', 'Edit', 'View', and 'Help'. The prompt is '(light@light)-[~]'. The user enters '\$ cd Desktop/' and the prompt changes to '(light@light)-[~/Desktop]'. The user then enters '\$ ls' and the output is 'Screenshot_2022-11-21_05-15-33.png'.

```
(light@light)-[~]  
$ cd Desktop/  
(light@light)-[~/Desktop]  
$ ls  
Screenshot_2022-11-21_05-15-33.png
```

SYNTAX:

`cd directory_name`

EXAMPLE:

To change into “work directory” in “documents” need to write as follows.

Input: `$ cd /documents/work`



```
(light@light)-[~]  
$ cd Desktop/  
  
(light@light)-[~/Desktop]  
$ ls  
Screenshot_2022-11-21_05-15-33.png
```

3. cd ..

DESCRIPTION:

Move up one directory.

SYNTAX:

`cd ..`

EXAMPLE:

If you are in work directory and want to go to documents then write

`cd ..`

You will end up in /documents.

4. ls : list all the files and directories

DESCRIPTION:

List all files and folders in the current directory in the column format.

SYNTAX:

`ls [options]`

EXAMPLE: Using various options

- Lists the total files in the directory and subdirectories, the names of the files in the current

5. cat

DESCRIPTION:

cat stands for "catenate". It reads data from files, and outputs their contents. It is the simplest way to display the contents of a file at the command line.

SYNTAX:

cat filename

EXAMPLES:

- Print the contents of files mytext.txt and yourtext.txt

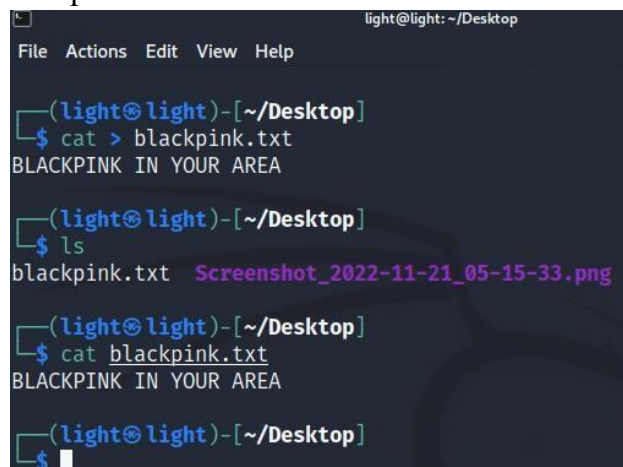
```
cat mytext.txt yourtext.txt
```

- Print the cpu information using cat command

```
cat /proc/cpuinfo
```

- Print the memory information using cat command

```
cat /proc/meminfo
```



```
light@light: ~/Desktop
File Actions Edit View Help

(light@light)-[~/Desktop]
$ cat > blackpink.txt
BLACKPINK IN YOUR AREA

(light@light)-[~/Desktop]
$ ls
blackpink.txt  Screenshot_2022-11-21_05-15-33.png

(light@light)-[~/Desktop]
$ cat blackpink.txt
BLACKPINK IN YOUR AREA

(light@light)-[~/Desktop]
$
```

6. head

DESCRIPTION:

head, by default, prints the first 10 lines of each FILE to standard output. With more than one FILE, it precedes each set of output with a header identifying the file name.

If no FILE is specified, or when FILE is specified as a dash ("-"), head reads from standard input.

SYNTAX:

head [option]...[file/directory]

EXAMPLE:

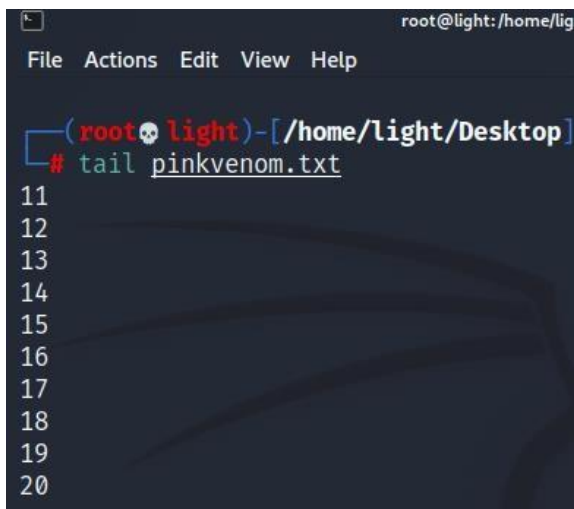
Display the first ten lines of myfile.txt.

head myfile.txt



```
(root@light)-[/home/light/Desktop]
# head pinkvenom.txt
1
2
3
4
5
6
7
8
9
```

7. tail



```
root@light:/home/light/Desktop
File Actions Edit View Help
(root@light)-[/home/light/Desktop]
# tail pinkvenom.txt
11
12
13
14
15
16
17
18
19
20
```

DESCRIPTION:

`tail` is a command which prints the last few number of lines (10 lines by default) of a certain file, then terminates.

SYNTAX:

`tail [option]...[file/directory]`

EXAMPLE:

Output the last 100 lines of the file `myfile.txt`.

`tail myfile.txt -n 100`

8. `mv` : Moving (and Renaming) Files

DESCRIPTION:

The `mv` command lets you move a file from one directory location to another. It also lets you rename a file (there is no separate *rename* command).

SYNTAX:

`mv [option] source directory`

EXAMPLE:

- Moves the file `myfile.txt` to the directory `destination-directory`.

`mv myfile.txt destination_directory`

- Move the file `myfile.txt` into the parent directory.

`mv myfile.txt ../`

- In this case, if `JOE1_expenses` does not exist, it will be created with the exact content of `joe_expenses`, and `joe_expenses` will disappear.

If `JOE1_expenses` already exists, its content will be replaced with that of `joe_expenses` (and `joe_expenses` will still disappear).

`mv joe_expenses JOE1_expenses`

```
(root@light)-[/home/light/Desktop]
# mv lightyagami.png /home

(root@light)-[/home/light/Desktop]
# ls

(root@light)-[/home/light/Desktop]
#
```

9. mkdir : Make Directory

DESCRIPTION:

```
light@light: ~
File Actions Edit View Help

(light@light)-[~]
$ mkdir light
mkdir: cannot create directory 'light': File exists

(light@light)-[~]
$ mkdir blackpink

(light@light)-[~]
$
```

If the specified directory does not already exist, mkdir creates it. More than one directory may be specified when calling mkdir.

SYNTAX:

```
mkdir [option] directory
```

EXAMPLE:

Create a directory named work.

```
mkdir work
```

10. cp : Copy Files

DESCRIPTION:

The cp command is used to make copy of files and directories.

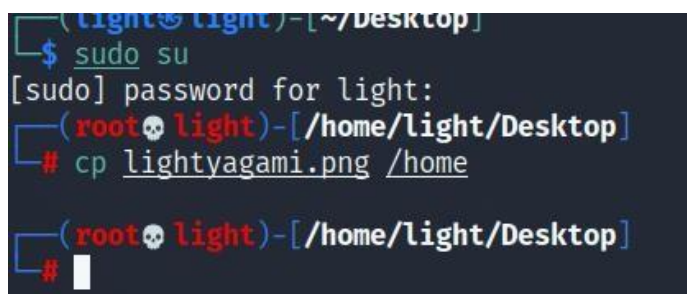
SYNTAX:

```
cp [option] source directory
```

EXAMPLE:

Creates a copy of the file in the currently working directory named origfile. The copy will be named newfile, and will be located in the working directory.

```
cp origfile newfile
```



```
(light@light)-[~/Desktop]
$ sudo su
[sudo] password for light:
(root@light)-[/home/light/Desktop]
# cp lightvagami.png /home
(root@light)-[/home/light/Desktop]
#
```

11. rmdir : Remove Directory

DESCRIPTION:

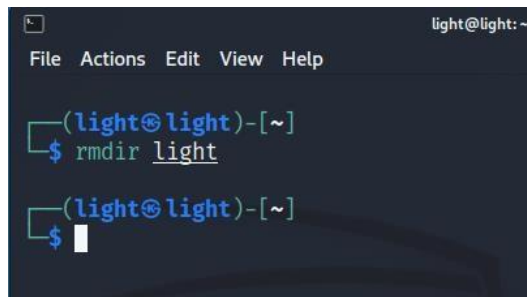
The rmdir command is used to remove a directory that contains other files or directories.

SYNTAX:

`rm directory_name`

EXAMPLE:

Delete mydir directory along with all files and directories within that directory. Here, -r is for recursive and -f is for forcefully.

A screenshot of a terminal window with a dark background. The window title is 'light@light: ~'. The menu bar shows 'File', 'Actions', 'Edit', 'View', and 'Help'. The prompt is '(light@light)-[~]'. The user enters '\$ rmdir light'. The prompt changes to '(light@light)-[~]' again, and the cursor is at the end of the line, indicating the command has been executed.

```
light@light: ~  
File Actions Edit View Help  
(light@light)-[~]  
$ rmdir light  
(light@light)-[~]  
$
```

Rmdir -rf my dir

12. gedit

DESCRIPTION:

The gedit command is used to create and open a file.

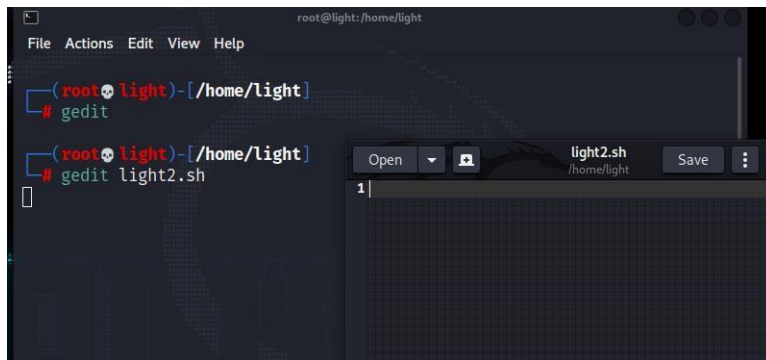
SYNTAX:

gedit filename.txt

EXAMPLE:

To create a file named abc.sh

gedit light2.sh



MAN

DESCRIPTION:

Displays on online manual page or manpage.

SYNTAX:

man command

EXAMPLE:

To learn about listing files

man ls

```
(root👤light)-[/home/light/Desktop]
# nikto man
- Nikto v2.1.6

--
+ ERROR: No host or URL specified

      -config+           Use this conf
      -Display+         Turn on/off d
      -dbcheck          check databas
errors
```


13. clear

DESCRIPTION:

Used to clear the screen

SYNTAX:

clear

EXAMPLE:

Clear the entire screen

clear

14. whoami

DESCRIPTION:

whoami prints the effective user ID. This command prints the username associated with the current effective user ID.

SYNTAX:

whoami [option]

EXAMPLE:

Display the name of the user who runs the command.

Whoami



```
(root skull light)-[/home]# whoami
root
(root skull light)-[/home]#
```

15. wc

DESCRIPTION:

wc (word count) command, can return the number of lines, words, and characters in a file.

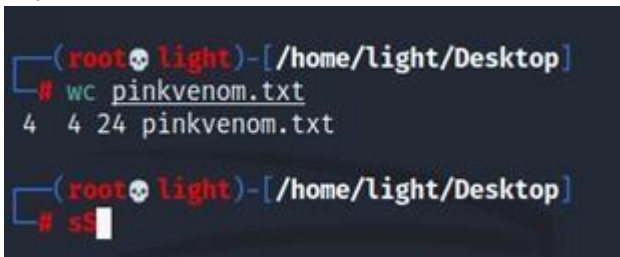
SYNTAX:

wc [option]... [file]...

EXAMPLE:

- Print the byte counts of file myfile.txt wc -c myfile.txt
- Print the line counts of file myfile.txt

wc -l myfile.txt



```
(root@light)-[/home/light/Desktop]
# wc pinkvenom.txt
4 4 24 pinkvenom.txt

(root@light)-[/home/light/Desktop]
# ss
```


- Print the word counts of file myfile.txt wc -w myfile.txt
- Print the byte counts of file myfile.txt wc -c myfile.txt
- Print the line counts of file myfile.txt

wc -l myfile.txt

- Print the word counts of file myfile.txt wc -w myfile.txt

16. grep

DESCRIPTION:



```
(root@light)-[/home/light/Desktop]
# cat pinkvenom.txt
AREA
YOUR
IN
BLACKPINK

(root@light)-[/home/light/Desktop]
# grep A pinkvenom.txt
AREA
BLACKPINK
```

grep command uses a search term to look through a file.

SYNTAX:

grep [option]... Pattern [file]...

EXAMPLE:

Search the word Hello in file named myfile.txt `grep "Hello" myfile.txt`

18. FREE

DESCRIPTION:

Display RAM details in Linux machine.

SYNTAX:

`free`

EXAMPLE:

To display the RAM details in Linux machine need to write following command.

`free`

1. pipe (|)

DESCRIPTION:

Pipe command is used to send output of one program as a input to another. Pipes “|” help combine 2 or more commands.

SYNTAX:

`Command 1 | command 2`

EXAMPLE:

Display lines of input files containing “Aug” and send to standard output `ls -l | grep "Aug"`

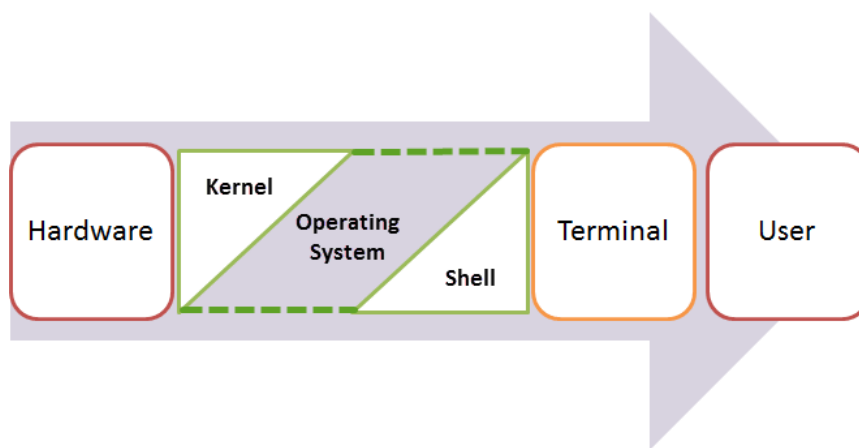
PRACTICAL NO: 2

Aim: Study the basics of shell programming.

What is a Shell?

An Operating is made of many components, but its two prime components are -

- Kernel
- Shell



A Kernel is at the nucleus of a computer. It makes the communication between the hardware and software possible. While the Kernel is the innermost part of an operating system, a shell is the outermost one.

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 known as sh
- **B**ourne **A**gain **S**hell also known 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

What is Shell Scripting?

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

Let us understand the steps in creating a Shell Script

1. **Create a file using a vi editor**(or any other editor). Name script file with **extension .sh**
2. **Start** the script with **#!/bin/sh**
3. Write some code.
4. Save the script file as filename.sh
5. For **executing** the script type **bash filename.sh**

"#!" is an operator called shebang which directs the script to the interpreter location. So, if we use "#!/bin/sh" the script gets directed to the bourne-shell.

Let's create a small script -

```
#!/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     Temp  
abc.bash  examples.desktop  Public      test  
abcd.sh   help         sample      test
```

Adding shell comments

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

```
#comment
```

Let understand this with an example.



What are Shell Variables?

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

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

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

Below is a small script which will use a variable.

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

Let's understand, the steps to create and execute the script

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!
```

script repeats the remark

As you see, the program picked the value of the variable 'name' as Joy and 'remark' as excellent.

This is a simple script. You can develop advanced scripts which contain conditional statements, loops, and functions. Shell scripting will make your life easy and Linux administration a breeze.

Summary:

- Kernel is the nucleus of the operating systems, and it communicates between hardware and software
- Shell is a program which interprets user commands through CLI like Terminal
- The Bourne shell and the C shell are the most used shells in Linux
- Shell scripting is writing a series of command for the shell to execute
- Shell variables store the value of a string or a number for the shell to read
- Shell scripting can help you create complex programs containing conditional statements, loops, and functions

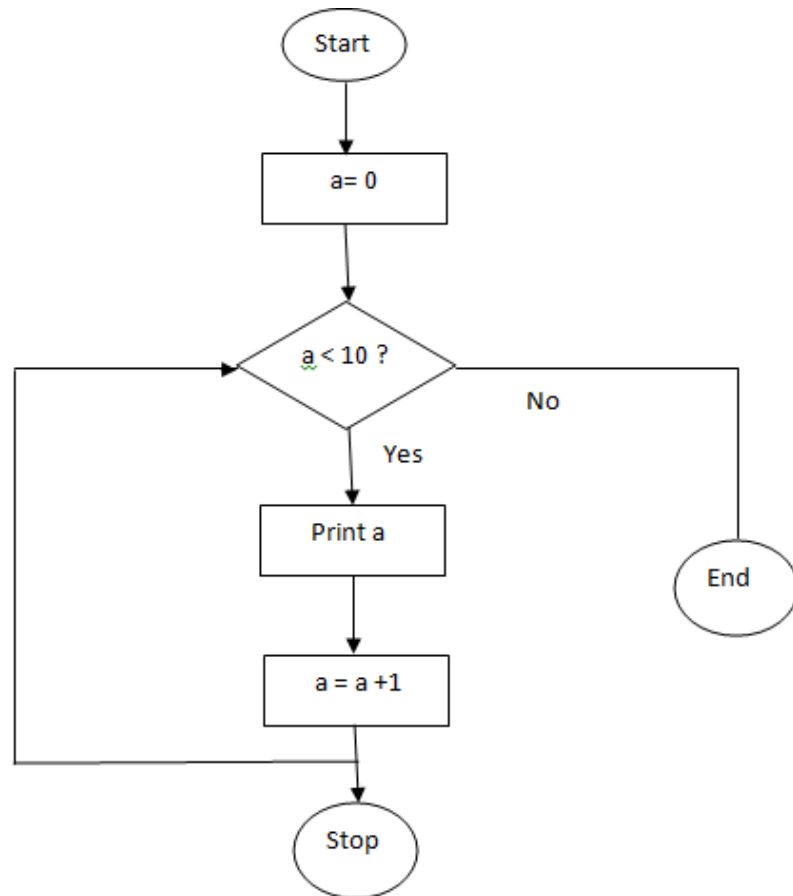
For more detail reference, follow this link: <https://www.shellscript.sh/>

PRACTICAL NO: 3

Definition: Take any number from the user. Get each and every digit one by one and make addition of that to find the sum of digits of a given number. E.g. if user has entered 342 then the sum of digits is $3+4+2 = 9$.

Set 1: Printing numbers from 0 to 9 using while loop.

Flowchart:



Sample Code:

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

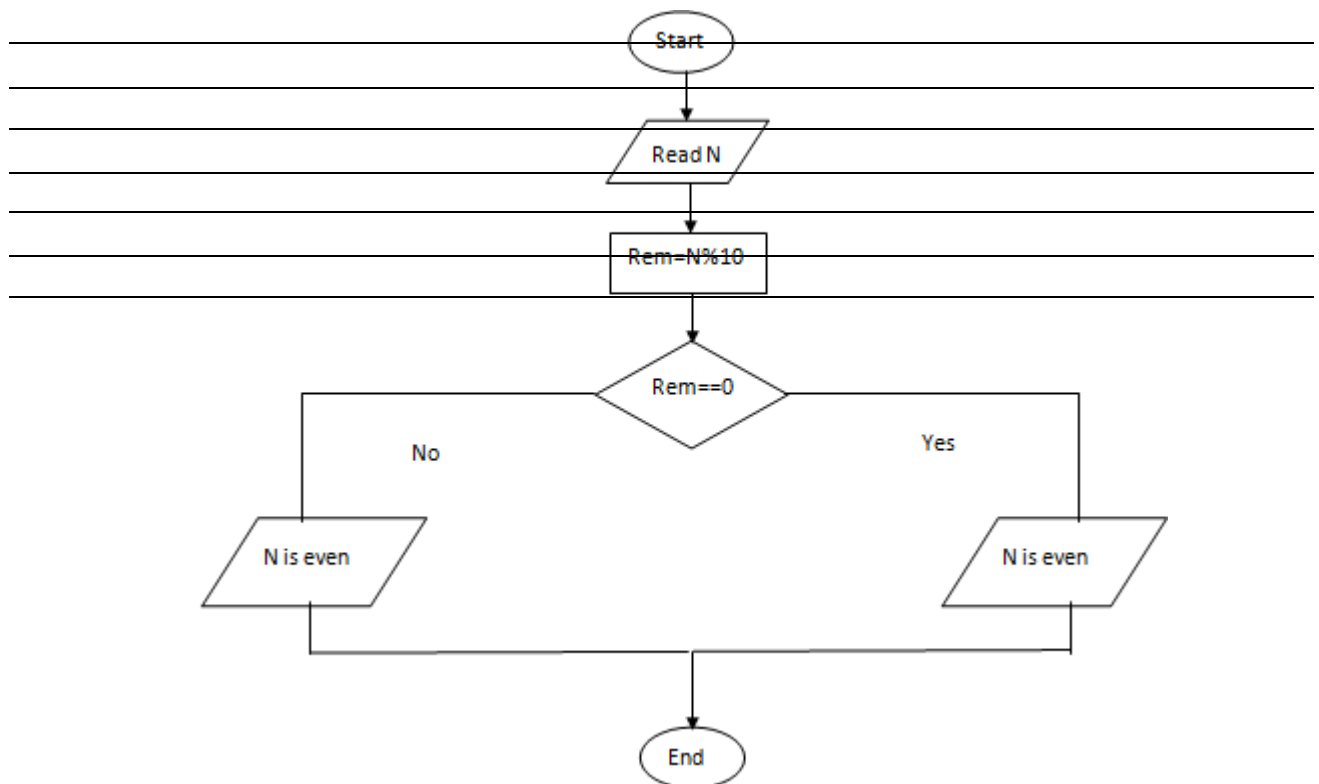
Input: -

Output: To execute shell script need to write: **sh filename**

0
1
2
3
4
5
6
7
8
9

Set 2: To find whether a number is even or odd.

Flowchart:



Sample Code:

```

echo "Enter a number"
read num

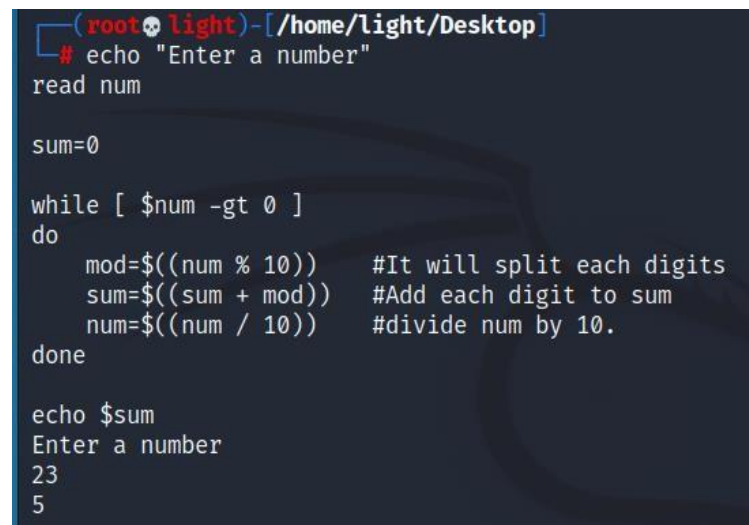
sum=0
  
```

```
while [ $num -gt 0 ]
do
    mod=$((num % 10)) #It will split each digits
    sum=$((sum + mod)) #Add each digit to sum
    num=$((num / 10)) #divide num by 10.
done
```

```
echo $sum
```

Note: -gt=greater than, -lt=less than

OUTPUT:



```
(root@light)-[/home/light/Desktop]
# echo "Enter a number"
read num

sum=0

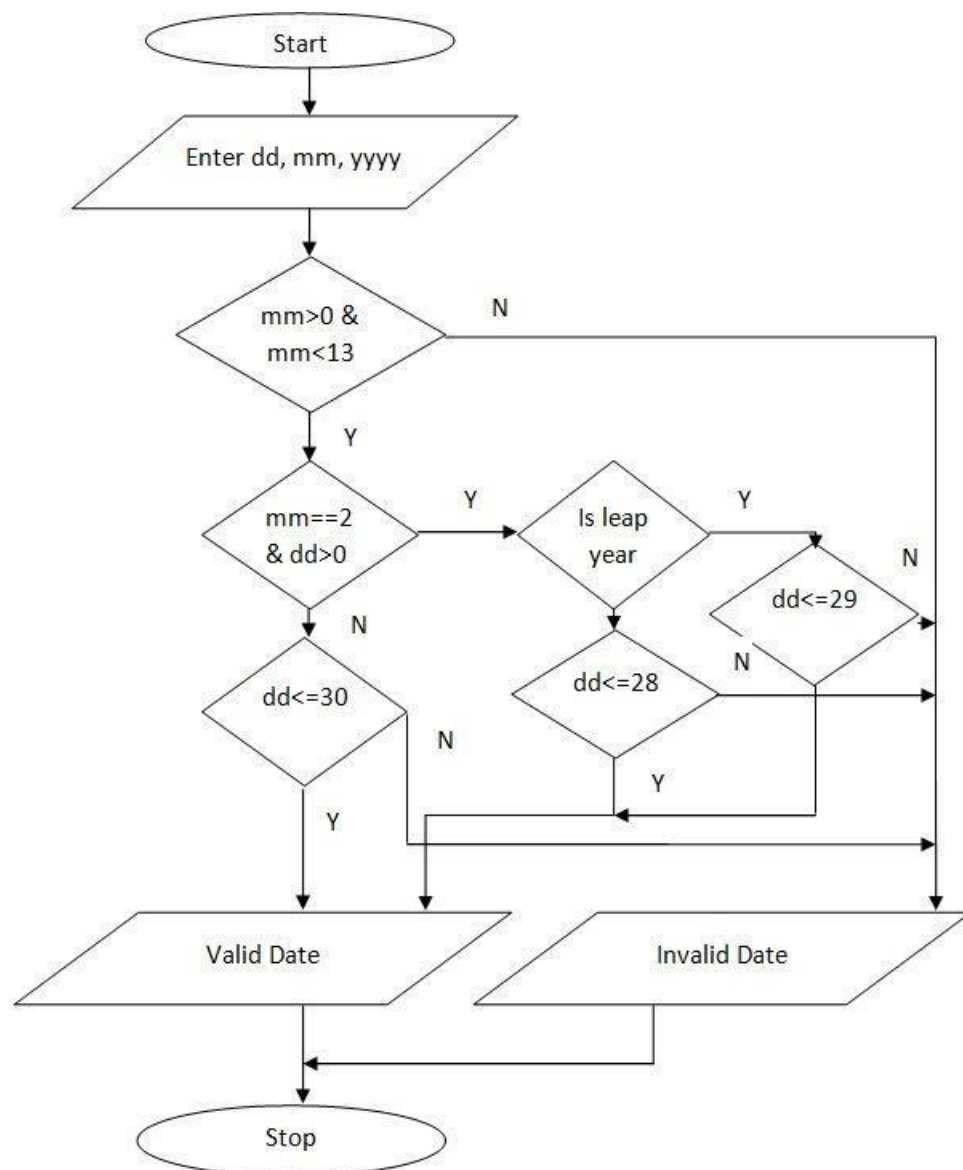
while [ $num -gt 0 ]
do
    mod=$((num % 10))    #It will split each digits
    sum=$((sum + mod))  #Add each digit to sum
    num=$((num / 10))   #divide num by 10.
done

echo $sum
Enter a number
23
5
```

PRACTICAL NO: 4

Definition: In a year there are 12 months and in each month the no. of days are 30 or 31. For February month, the no. of days is 28 and if it is a leap year then the no. days will be 29. Bychecking these conditions for days, month and year using various control statements of Linux can validate the date entered by the user. (eg. Date format is: dd-mm-yyyy)

Flowchart:



Set 1: Read a string from the user and if it is hello then print “Hello yourself”, if it is bye then print “See you again” otherwise print “Sorry, I don't understand" using case statement.

Syntax: bash case statement.

```
case expression
in
pattern1 )
statements ;;
pattern2 )
statements ;;
...
esac
```

Following are the key points of bash case statements:

- Case statement first expands the expression and tries to match it against each pattern.
- When a match is found all of the associated statements until the double semicolon (;;) are executed.
- After the first match, case terminates with the exit status of the last command that was executed.
- If there is no match, exit status of case is zero.

Sample Code:

```
read str
case $str in
    hello)
        echo "Hello yourself!"
        ;;
    bye)
        echo "See you again!"
        ;;
    *)
        echo "Sorry, I don't understand"
        ;;
esac
```

Input:

Hi

Output:

Sorry, I don't understand

Code:

```
dd=0
mm=0
yy=0

days=0
echo -n "Enter day (dd) : "
read dd

echo -n "Enter month (mm) : "
read mm

echo -n "Enter year (yyyy) : "
read yy

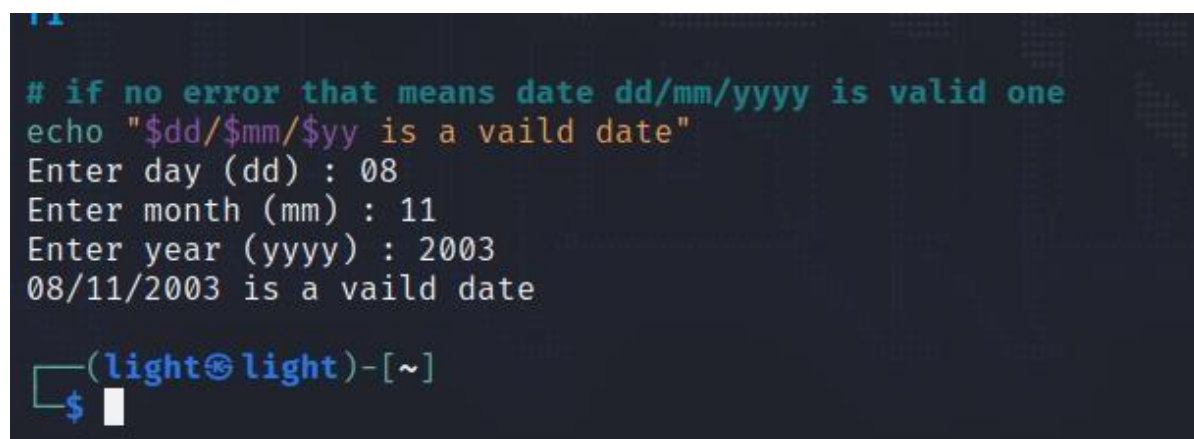
if [ $mm -le 0 -o $mm -gt 12 ];
then
    echo "$mm is invalid month."
    exit 1
fi
case $mm in
    1) days=31;;
    2) days=28 ;;
    3) days=31 ;;
    4) days=30 ;;
    5) days=31 ;;
    6) days=30 ;;
    7) days=31 ;;
    8) days=31 ;;
    9) days=30 ;;
    10) days=31 ;;
    11) days=30 ;;
    12) days=31 ;;
    *) days=-1;;
esac
if [ $mm -eq 2 ]; # if it is feb month then only check of leap year
then
    if [ $((yy % 4)) -ne 0 ] ; then
        : # not a leap year : means do nothing and use old value of days
    elif [ $((yy % 400)) -eq 0 ] ; then
        # yes, it's a leap year
        days=29
    elif [ $((yy % 100)) -eq 0 ] ; then
        : # not a leap year do nothing and use old value of days
    else
        # it is a leap year
        days=29
    fi
fi

if [ $dd -le 0 -o $dd -gt $days ];
then
    echo "$dd day is invalid"
    exit 3
```

fi

```
echo "$dd/$mm/$yy is a vaild date"
```

OUTPUT



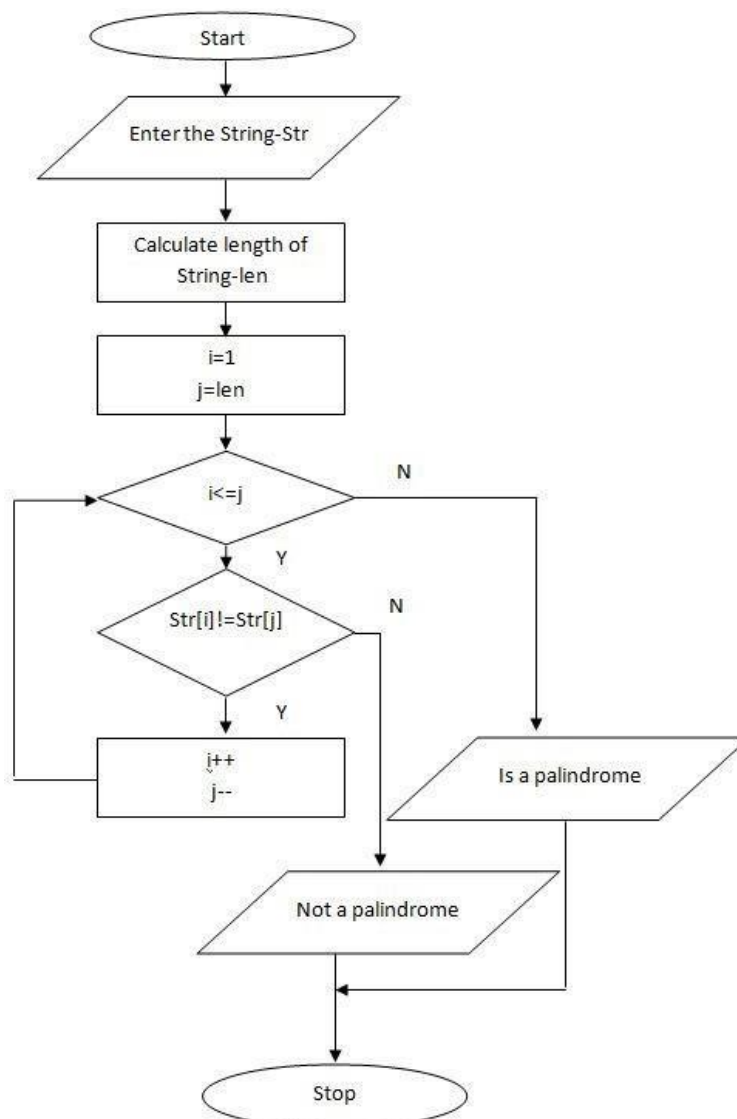
```
11
# if no error that means date dd/mm/yyyy is valid one
echo "$dd/$mm/$yy is a vaild date"
Enter day (dd) : 08
Enter month (mm) : 11
Enter year (yyyy) : 2003
08/11/2003 is a vaild date

(light@light)-[~]
$
```

PRACTICAL NO: **5**

Definition: A string is said to be palindrome if reverse of the string is same as string. For example, “abba” is palindrome, but “abbc” is not palindrome. We can use Linux command to find the length of the string and then by comparing first character and last character of the string, second character and second last character of the string and so on., we can identify that whether the string is palindrome or not.

Flowchart:



Set 1: Extract the second character of the entered string.

Sample Code:

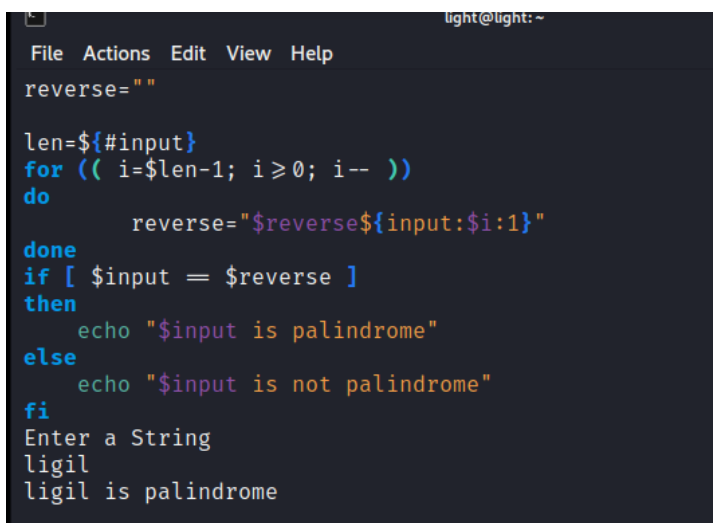
```
echo "Enter a String"
```

```
read str  
k=`echo $str | cut -c 2` echo "$k"
```

Input:

```
#!/bin/bash  
echo "Enter a String"  
read input  
reverse=""  
len=${#input}  
for (( i=$len-1; i>=0; i-- ))  
do  
    reverse="$reverse${input:$i:1}"  
done  
if [ $input == $reverse ]  
then  
    echo "$input is palindrome"  
else  
    echo "$input is not palindrome"  
fi
```

OUTPUT

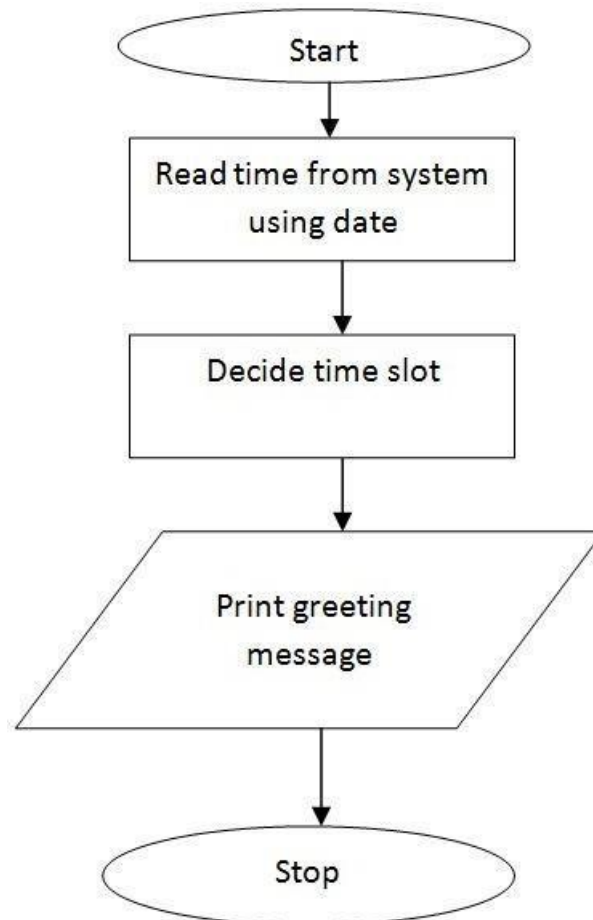


```
light@light: ~  
File Actions Edit View Help  
reverse=""  
  
len=${#input}  
for (( i=$len-1; i>=0; i-- ))  
do  
    reverse="$reverse${input:$i:1}"  
done  
if [ $input = $reverse ]  
then  
    echo "$input is palindrome"  
else  
    echo "$input is not palindrome"  
fi  
Enter a String  
ligil  
ligil is palindrome
```


PRACTICAL NO: 6

Definition: The date command is used to print out the system's time and date information. By extracting hours from the date using Linux 'cut' command and by using if else ladder can wish user an appropriate message like Good morning/Afternoon/Evening.

Flowchart:



Set 1: Read the minutes from system.

Sample Code:

```
minutes=`date + M%`  
echo $minutes
```

Set 2: Read the hours from system using cut command of unix.

Sample Code:

```
clear
hours=`date | cut -c12-13`
echo "the value of hour is $hours"
```

Input: -

Output:

the value of hour is 4

Set 3: Write a Shell script to say Good morning/Afternoon/Evening as you log in to system.

Syntax

```
h=$(date +"%H")
if [ $h -gt 6 -a $h -le 12 ]
then
echo good morning
elif [ $h -gt 12 -a $h -le 16 ]
then
echo good afternoon
elif [ $h -gt 16 -a $h -le 20 ]
then
echo good evening
else
echo good night
fi
```

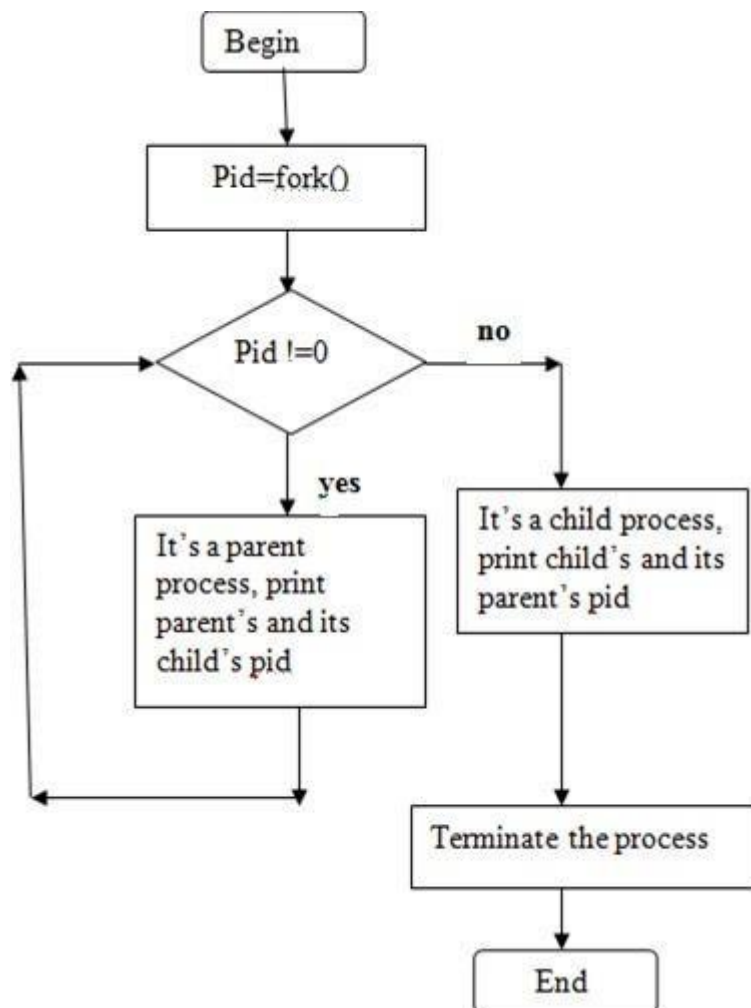
```
(light@light)-[~]  
$ h=$(date +%H)  
if [ $h -gt 6 -a $h -le 12 ]  
then  
echo good morning  
elif [ $h -gt 12 -a $h -le 4 ]  
then  
echo good afternoon  
elif [ $h -gt 4 -a $h -le 7 ]  
then  
echo good evening  
else  
echo good night  
fi  
good night  
  
(light@light)-[~]  
$
```

PRACTICAL NO: 7

Definition: Create child process using `fork()`, which is a system call to create a child process. Use `getpid()` for getting the id of the process, `getppid()` for getting the parent process id.

Fork system call is used for creating a new process, which is called *child process*, which runs concurrently with the process that makes the `fork()` call (parent process). After a new child process is created, both processes will execute the next instruction following the `fork()` system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.

Flowchart:



Set 1: Create a child process using one `fork()`, `getpid()`, `getppid()`.

*Note: fork() is threading based function, to get the correct output run the program on a local system. **Please note that the above program do not compile in Windows Environment.***

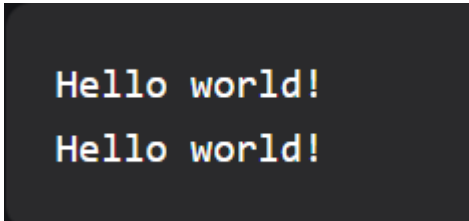
INPUT:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{

    // make two process which run same
    // program after this instruction
    fork();

    printf("Hello world!\n");
    return 0;
}
```

OUTPUT:



```
Hello world!
Hello world!
```

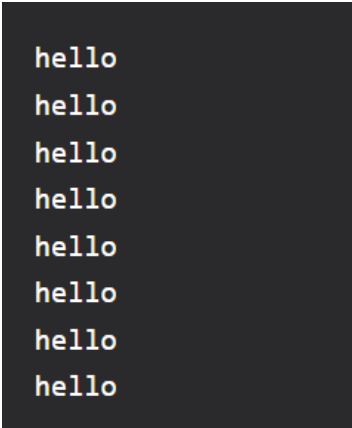
Set 2.

INPUT:

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
}
```

```
printf("hello\n");  
return 0;  
}
```

OUTPUT:



```
hello  
hello  
hello  
hello  
hello  
hello  
hello  
hello
```

****The number of times ‘hello’ is printed is equal to number of process created. Total Number of Processes = 2^n , where n is number of fork system calls. So here $n = 3$, $2^3 = 8$.**

Set 3

INPUT:

```
#include <stdio.h>  
#include <sys/types.h>  
#include <unistd.h>  
void forkexample()  
{  
    // child process because return value zero  
    if (fork() == 0)  
        printf("Hello from Child!\n");  
  
    // parent process because return value non-zero.  
    else  
        printf("Hello from Parent!\n");  
}  
int main()  
{  
    forkexample();  
    return 0;  
}
```

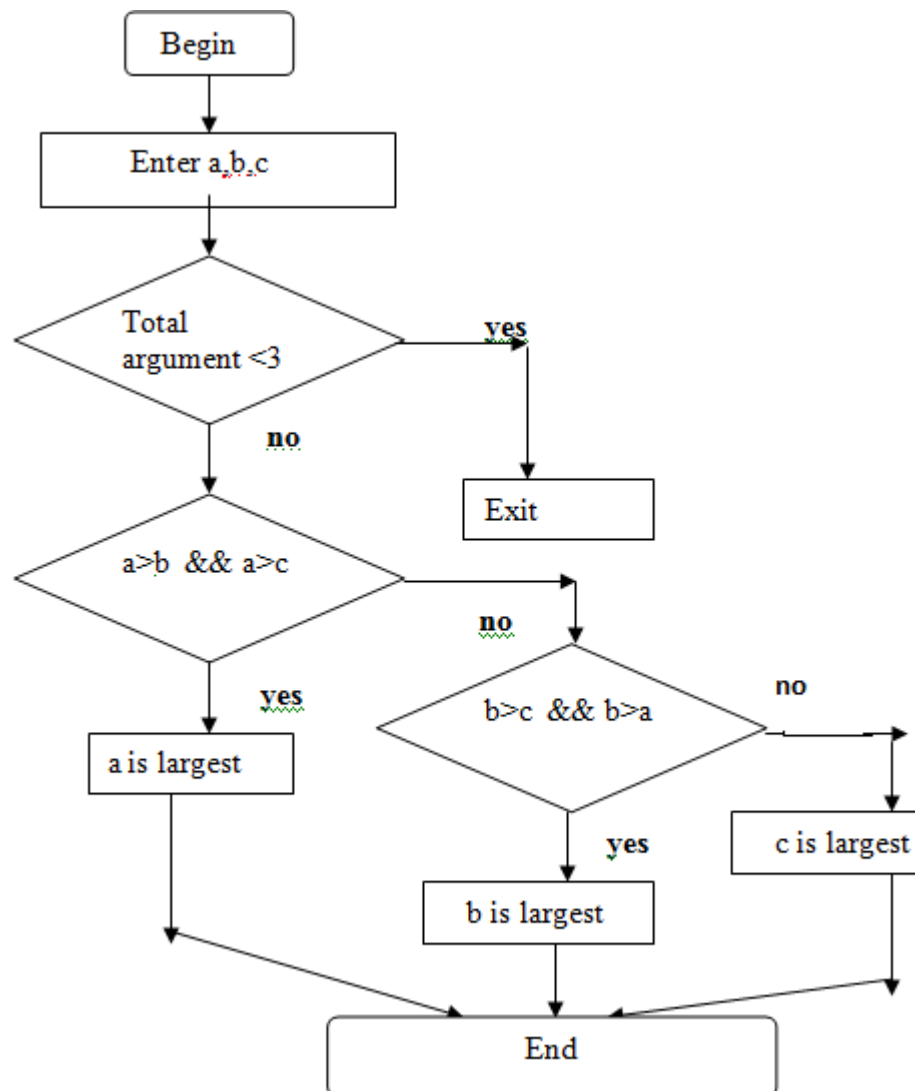
OUTPUT:

```
1.  
Hello from Child!  
Hello from Parent!  
    (or)  
2.  
Hello from Parent!  
Hello from Child!
```

PRACTICAL NO: 8

Definition: Finding out biggest number from given three numbers supplied as command line argument. Command argument means at run time, the user will enter the input. First input will be considered as number 1, second input will be consider as number 2 and so on. It depends on the no of command line argument.

Flowchart:

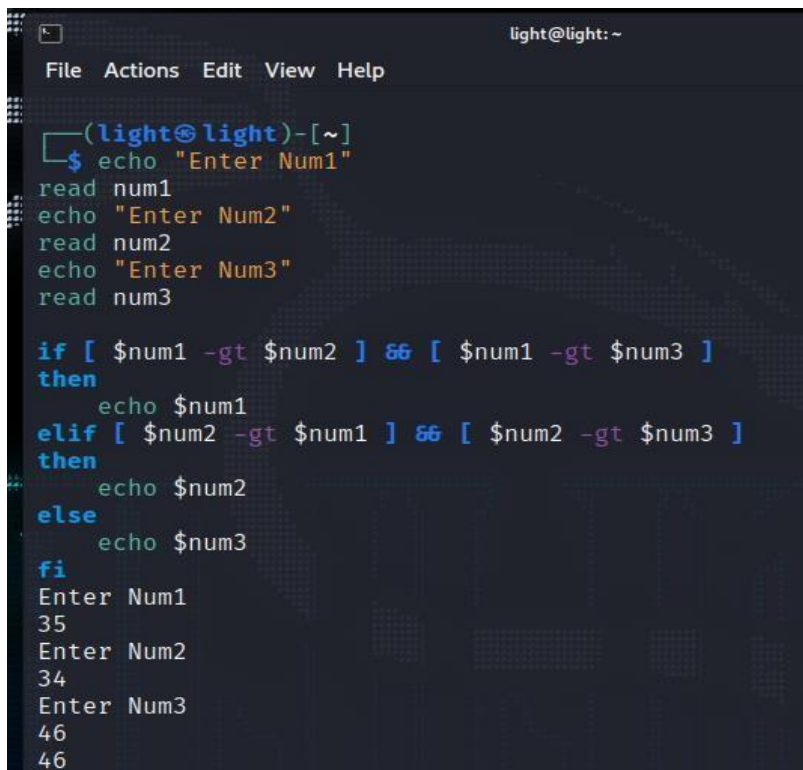


Show the use of command line argument for finding the biggest number among three numbers using if else ladder.

INPUT:

```
echo "Enter Num1"
read num1
echo "Enter Num2"
read num2
echo "Enter Num3"
read num3
if [ $num1 -gt $num2 ] && [ $num1 -gt $num3 ]
then
    echo $num1
elif [ $num2 -gt $num1 ] && [ $num2 -gt $num3 ]
then
    echo $num2
else
    echo $num3
fi
```

OUTPUT:



```
light@light: ~
File Actions Edit View Help

(light@light)-[~]
$ echo "Enter Num1"
read num1
echo "Enter Num2"
read num2
echo "Enter Num3"
read num3

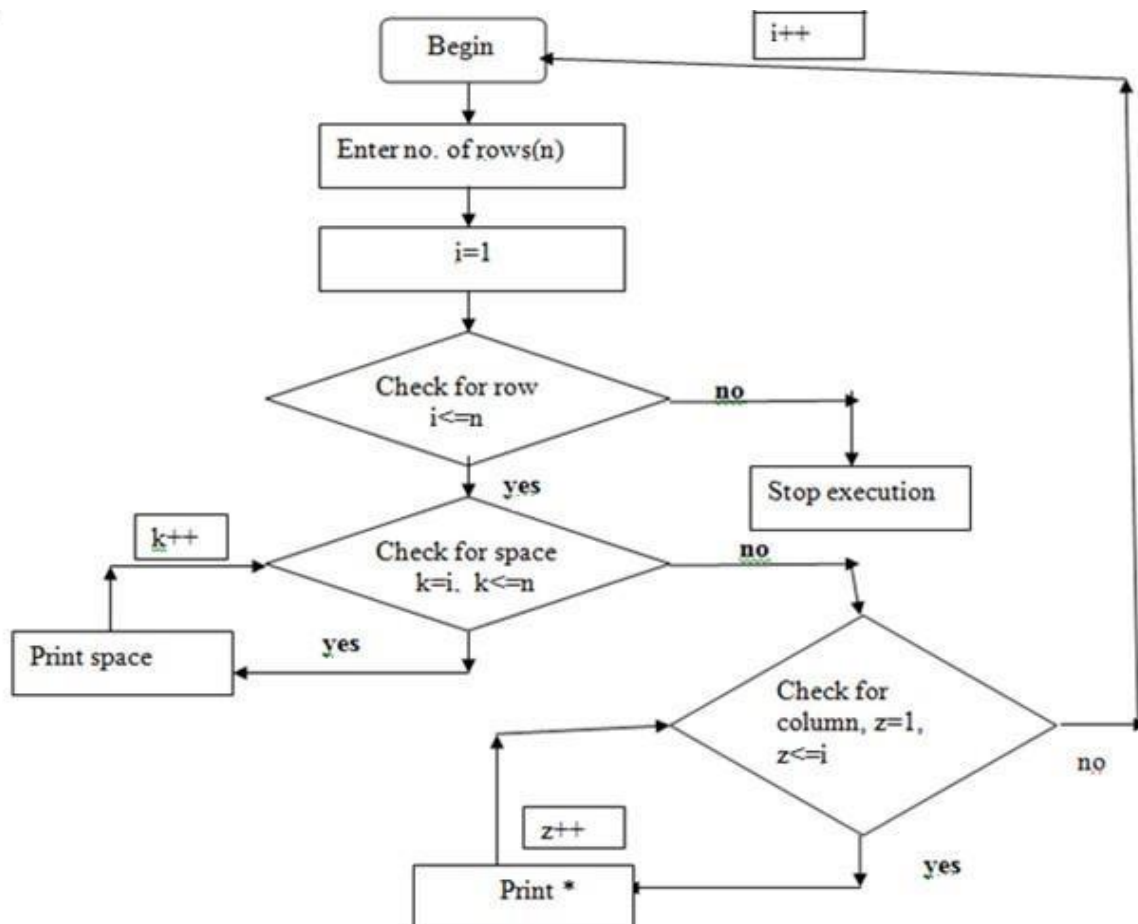
if [ $num1 -gt $num2 ] && [ $num1 -gt $num3 ]
then
    echo $num1
elif [ $num2 -gt $num1 ] && [ $num2 -gt $num3 ]
then
    echo $num2
else
    echo $num3
fi
Enter Num1
35
Enter Num2
34
Enter Num3
46
46
```

PRACTICAL NO: 9

Definition: Print the pattern using for loop which is used to do the same thing again and again until some condition is satisfied.

For loop is used to do the same thing until some condition is there.

Flowchart:



Code:

#Here \$1 is the parameter you passed. It specifies the no. of rows i,e “ / ”

```
# Static input for N
N=5
i=0
j=0
while [ $i -le `expr $N - 1` ]
do
    j=0
    while [
$ j -le `expr $N - 1` ]
    do
        if [ `expr $N - 1` -le `expr $i + $j` ]
        then
            # Print the pattern
            echo -ne "/"
        else
            # Print the spaces required
            echo -ne " "
        fi
        j=`expr $j + 1`
    done
    # For next line
    echo
    i=`expr $i + 1`
done
```

Input:

/

Output:

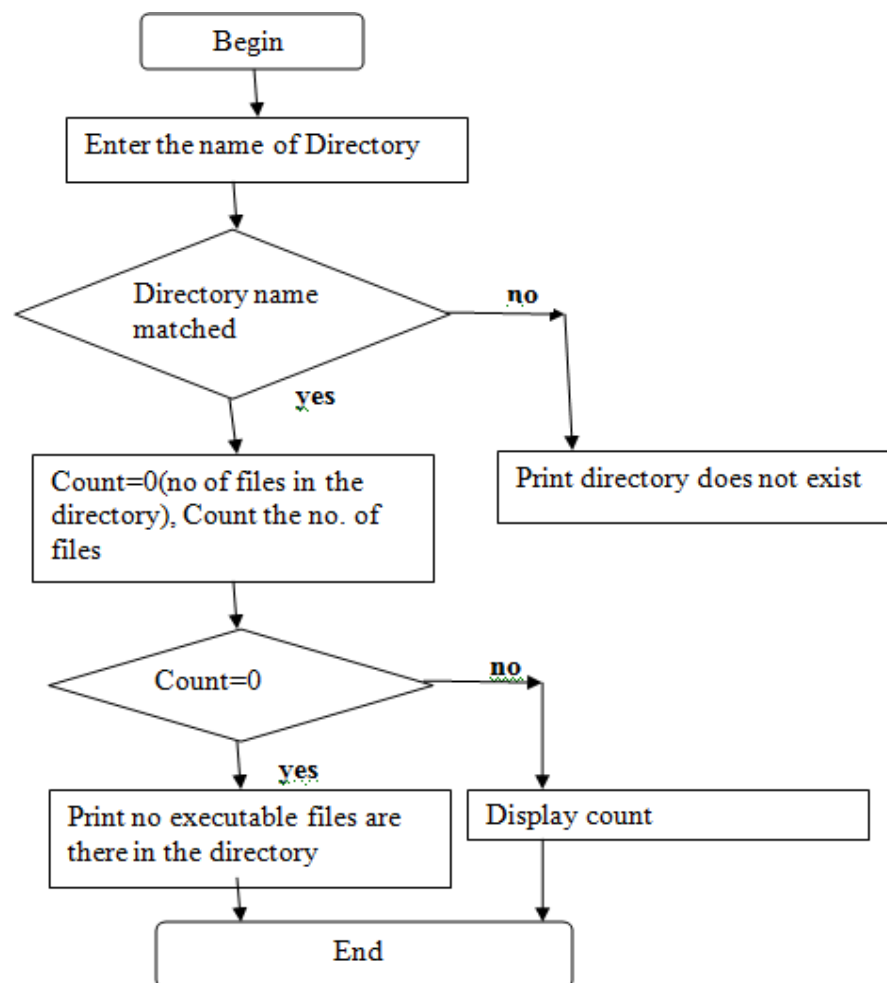
```
if [ `expr $N - 1` -le `expr $i + $j`  
then  
# Print the pattern  
echo -ne "/"  
else  
# Print the spaces required  
echo -ne " "  
fi  
j=`expr $j + 1`  
  
done  
# For next line  
echo  
  
i=`expr $i + 1`  
  
done  
/  
//  
///  
////  
/////
```

(light@light)-[~]
\$

PRACTICAL NO: 10

Definition: Various commands are available in Linux to check whether the given directory or file are exist or not in the system. Several options are also available to check special conditions like file is empty or not. We can use these commands in shell script as well.

Flowchart:

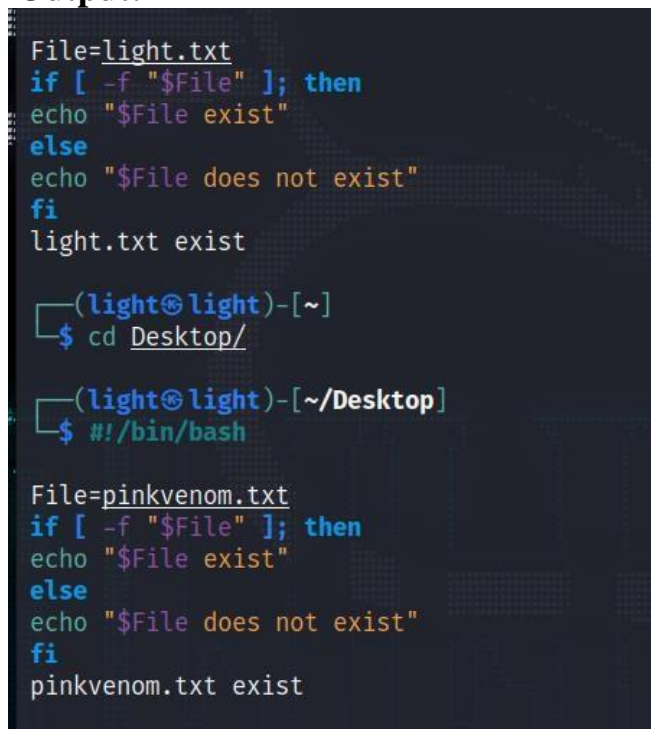


Code:

Shell script to determine whether given directory, exist or not.

```
#!/bin/bash
File=read_file.txt
if [ -f "$File" ]; then
echo "$File exist"
else
echo "$File does not exist"
fi
```

Output:



```
File=light.txt
if [ -f "$File" ]; then
echo "$File exist"
else
echo "$File does not exist"
fi
light.txt exist

(light@light)-[~]
$ cd Desktop/

(light@light)-[~/Desktop]
$ #!/bin/bash

File=pinkvenom.txt
if [ -f "$File" ]; then
echo "$File exist"
else
echo "$File does not exist"
fi
pinkvenom.txt exist
```

PRACTICAL :11

Write a program for process creation using C. (Use of gcc compiler).

Let us consider a simple program.

File name: basicfork.c

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

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

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

```
int main() {
```

```
    fork();
```

```
    printf("Called fork() system call\n");
```

```
    return 0;
```

```
}
```

Execution Steps

Compilation

```
gcc basicfork.c -o basicfork
```

Execution/Output

Called fork() system call

Called fork() system call

Note – Usually after fork() call, the child process and the parent process would perform different tasks. If the same task needs to be run, then for each fork() call it would run 2^n times, where n is the number of times fork() is invoked.

In the above case, fork() is called once, hence the output is printed twice (2 power 1). If fork() is called, say 3 times, then the output would be printed 8 times (2 power 3). If it is called 5 times, then it prints 32 times and so on and so forth.

Having seen fork() create the child process, it is time to see the details of the parent and the child processes.

File name: pids_after_fork.c

```
#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main() {

    pid_t pid, mypid, myppid;

    pid = getpid();

    printf("Before fork: Process id is %d\n", pid);

    pid = fork();

    if (pid < 0) {

        perror("fork() failure\n");

        return 1;

    }

    // Child process

    if (pid == 0) {

        printf("This is child process\n");

        mypid = getpid();

        myppid = getppid();

        printf("Process id is %d and PPID is %d\n", mypid, myppid);

    } else { // Parent process
```



```
sleep(2);

printf("This is parent process\n");

mypid = getpid();

myppid = getppid();

printf("Process id is %d and PPID is %d\n", mypid, myppid);

printf("Newly created process id or child pid is %d\n", pid);

}

return 0;

}
```

OUTPUT

```
Before fork: Process id is 166629
This is child process
Process id is 166630 and PPID is 166629
Before fork: Process id is 166629
This is parent process
Process id is 166629 and PPID is 166628
Newly created process id or child pid is 166630
```

A process can terminate in either of the two ways –

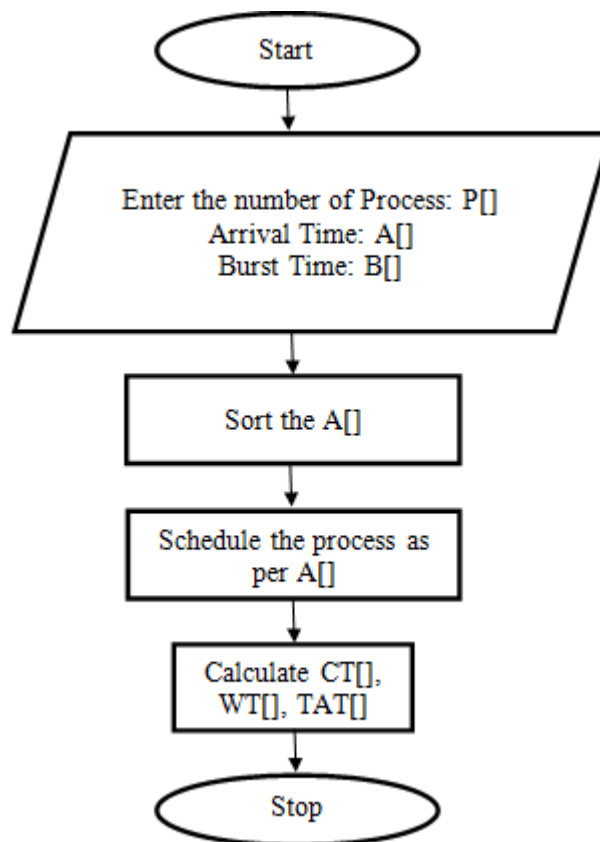
- Abnormally, occurs on delivery of certain signals, say terminate signal.
- Normally, using `_exit()` system call (or `_Exit()` system call) or `exit()` library function.

The difference between `_exit()` and `exit()` is mainly the cleanup activity. The `exit()` does some cleanup before returning the control back to the kernel, while the `_exit()` (or `_Exit()`) would return the control back to the kernel immediately.

PARUL UNIVERSITY
FACULTY OF ENGINEERING & TECHNOLOGY
Operating System (203105214) B. Tech. 2nd Year
PRACTICAL NO: 12

Definition: First come, first served (FCFS) is an operating system process scheduling algorithm and a network routing management mechanism that automatically executes queued requests and processes by the order of their arrival.

Flowchart:



Set 1: How to take Arrival Time and Burst Time of the processes.

Sample Code:

```
#include<stdio.h>
int main()
{
    int n,count;
    int at[10],bt[10];
    printf("Enter Total Process:\t ");
    scanf("%d",&n);
    for(count=0;count<n;count++)
```

```
printf("Enter Arrival Time and Burst Time for Process Number %d :",count+1);
scanf("%d",&at[count]);
scanf("%d",&bt[count]);
}
return 0;
}
```

Input:

Enter Total Process: 3

```
Enter the number of processes: 4
Enter arrival time and burst time for process 1: 2 02
Enter arrival time and burst time for process 2: 1 05
Enter arrival time and burst time for process 3: 2 04
Enter arrival time and burst time for process 4: 3 20
```

Set 2: How to sort processes based on their Arrival Time for 5 processes.

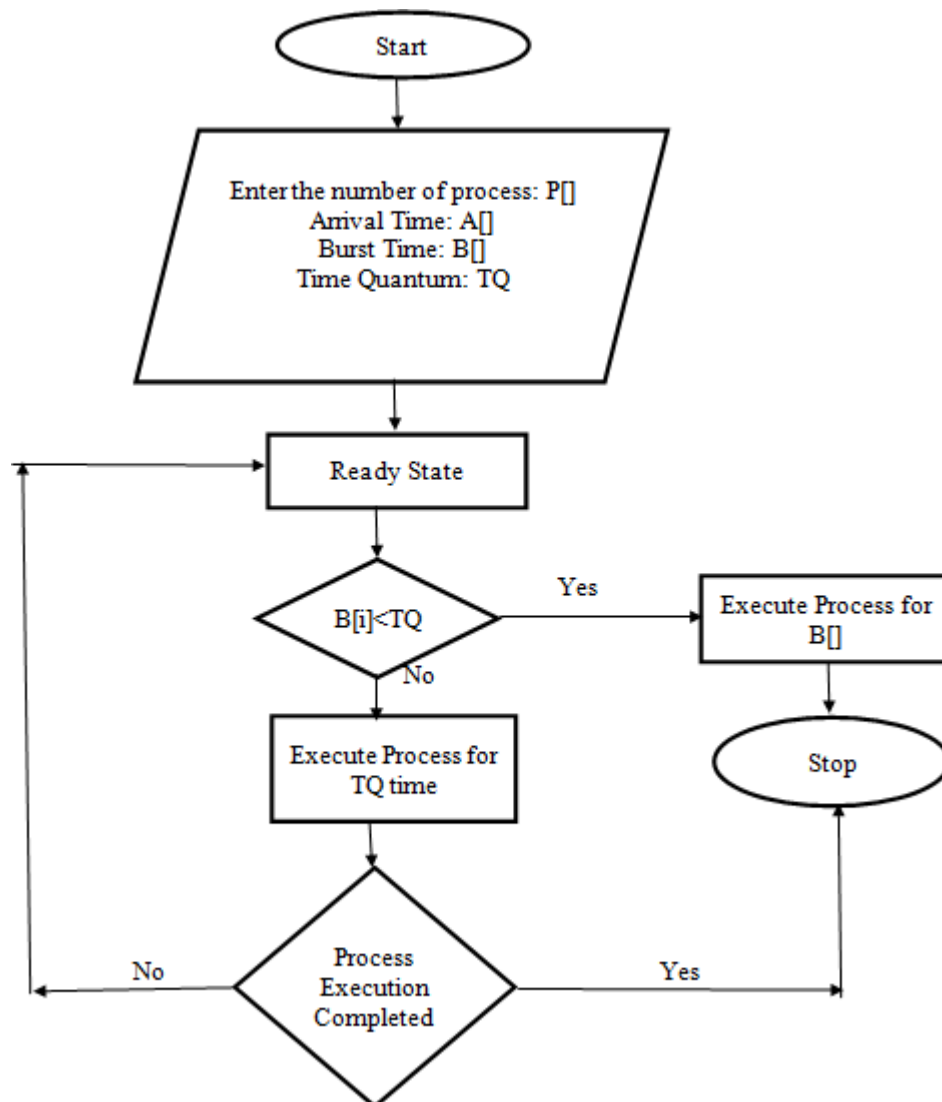
Sample Code:

```
#include<stdio.h>
void main()
{
    int i,temp,j,f,array[10];
    printf("Enter 5 Processes");
    //read array
    for(i=0;i<5;i++)
    {
        scanf("%d",&array[i]);
    }
    //bubble sort
    for(i=0;i<10;i++)
    {
        f=1;
        for(j=0;j<9;j++)
        {
            if(array[j]>array[j+1])
            {
                temp=array[j];
                array[j]=array[j+1];
                array[j+1]=temp;
                f=0;
            }
        }
        if(f==1)
    }
```

```
        break;
    }
    printf("after swaping");
    for(i=0;i<5;i++)
        printf(" %d",array[i])
}
```

Definition: Round robin (RR) scheduling is a job-scheduling algorithm that is considered to be very fair, as it uses time slices that are assigned to each process in the queue or line. Each process is then allowed to use the CPU for a given amount of time, and if it does not finish within the allotted time, it is preempted and then moved at the back of the line so that the next process in line is able to use the CPU for the same amount of time.

Flowchart:



Set 1: How to take Arrival Time, Burst Time and Time Quantum of the processes.

Sample Code:

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

```
int main()
{

    int n,count, tq;
    int at[10],bt[10];
    printf("Enter Total Process:\t ");
    scanf("%d",&n);
    for(count=0;count<n;count++)
    {
        printf("Enter Arrival Time and Burst Time for Process Number %d :",count+1);
        scanf("%d",&at[count]);
        scanf("%d",&bt[count]);
    }
    printf("Enter Time Quantum:\t ");
    scanf("%d",&tq);
    return 0;
}
```

Input:

Enter Total Process: 3

Output:

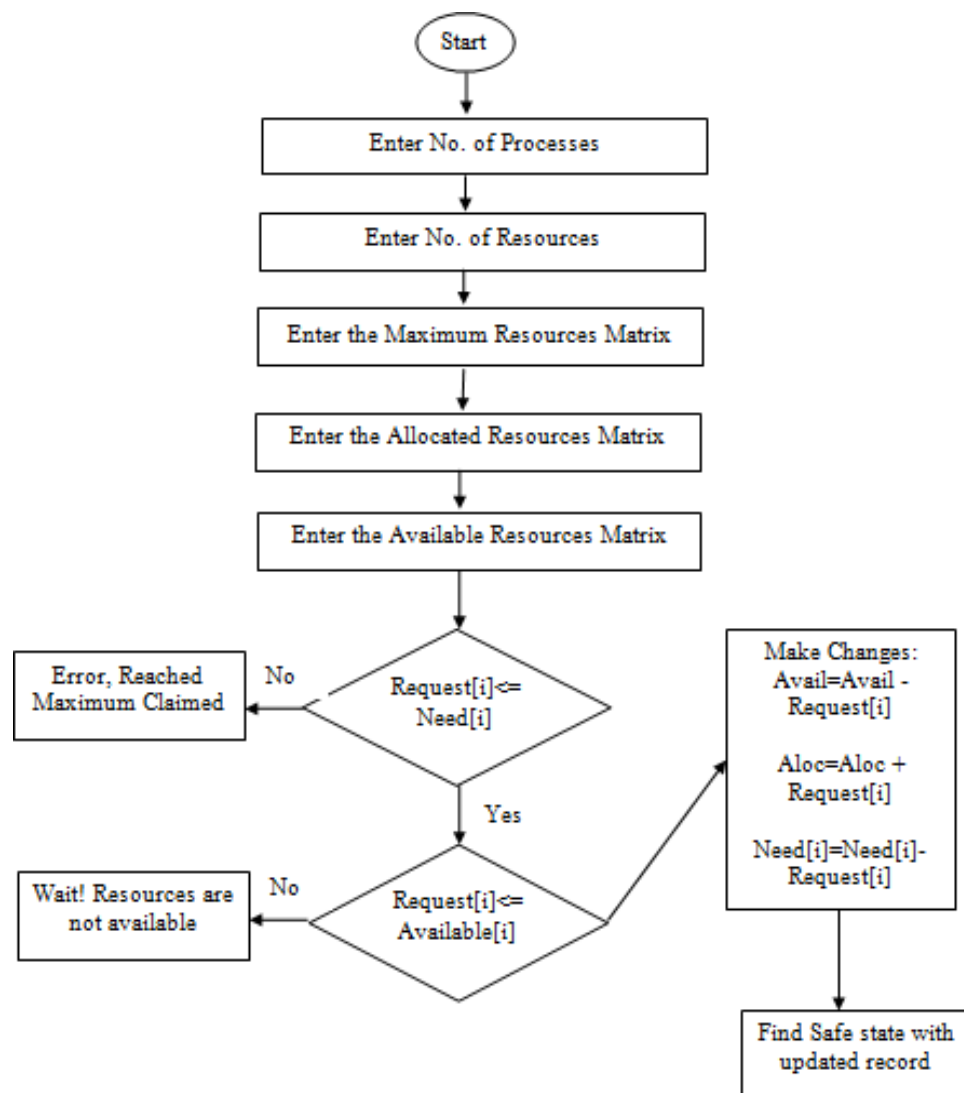
```
Enter Total Process:      3
Enter Arrival Time and Burst Time for Process Number 1 :1
3
Enter Arrival Time and Burst Time for Process Number 2 :2
4
Enter Arrival Time and Burst Time for Process Number 3 :1
2
Enter Time Quantum:      23
```

PRACTICAL NO: 13

Definition: The Banker's algorithm, sometimes referred to as the detection algorithm, is a resource allocation and deadlock avoidance algorithm. It tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources. When a new process enters a system, it must declare the maximum number of instances of each resource type that it may ever claim; clearly, that number may not exceed the total number of resources in the system. Also, when a process gets all its requested resources it must return them in a finite amount of time.

Flow Chart:

Resource-Request Algorithm



Set 1: How to take no. of processes, no. of resources, maximum resource matrix, allocated resource

matrix and available resources for each process.

Code:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int Max[10][10], need[10][10], alloc[10][10], avail[10], completed[10];
    int p, r, i, j, process, count;
    count = 0;
    printf("Enter the no of processes : ");
    scanf("%d", &p);
    for(i = 0; i < p; i++)
        completed[i] = 0;
    printf("\n\nEnter the no of resources : ");
    scanf("%d", &r);
    printf("\n\nEnter the Max Matrix for each process : ");
    for(i = 0; i < p; i++)
    {
        printf("\nFor process %d : ", i + 1);
        for(j = 0; j < r; j++)
            scanf("%d", &Max[i][j]);
    }
    printf("\n\nEnter the allocation for each process : ");
    for(i = 0; i < p; i++)
    {
        printf("\nFor process %d : ", i + 1);
        for(j = 0; j < r; j++)
            scanf("%d", &alloc[i][j]);
    }
    printf("\n\nEnter the Available Resources : ");
    for(i = 0; i < r; i++)
        scanf("%d", &avail[i]);

    for(i = 0; i < p; i++)
    {
        for( j = 0; j < r; j++)
            printf("%d ", Max[i][j]);
        printf("\t\t");
        for( j = 0; j < r; j++)
            printf("%d ", alloc[i][j]);
        printf("\n");
    }
}
```


Set 3: Perform the Banker's Algorithm and find out that whether the System is Safe or not and also print the Safe Sequence.

Safety Algorithm:

1. Let Work and Finish be vectors of length m and n, respectively. Initially,

Work = Available

Finish[i] = false for $i = 0, 1, \dots, n - 1$.

This means, initially, no process has finished and the number of available resources is represented by the Available array.

2. Find an index i such that both

Finish[i] == false

Needi \leq Work

If there is no such i present, then proceed to step 4.

It means, we need to find an unfinished process whose need can be satisfied by the available resources. If no such process exists, just go to step 4.

3. Perform the following:

Work = Work + Allocation;

Finish[i] = true;

Go to step 2.

When an unfinished process is found, then the resources are allocated and the process is marked finished. And then, the loop is repeated to check the same for all other processes.

4. If Finish[i] == true for all i, then the system is in a safe state.

That means if all processes are finished, then the system is in safe state.

```
Enter the number of resources : 3

Enter the max instances of each resource
a= 10
b= 5
c= 7

Enter the number of processes: 5

Enter the allocation matrix
      a b c
P[0]  0 1 0
P[1]  2 0 0
P[2]  3 0 2
P[3]  2 1 1
P[4]  0 0 2

Enter the MAX matrix
      a b c
P[0]  7 5 3
P[1]  3 2 2
P[2]  9 0 2
P[3]  4 2 2
P[4]  5 3 3

      < P[1]  P[3]  P[4]  P[0]  P[2] >
```