Parul University

FACULTY OF ENGINEERING AND TECHNOLOGY
BACHELOR OF TECHNOLOGY

OPERATING SYSTEM LABORATORY
(303105252)

IV SEMESTER

Computer Science & Engineering Department

Parul® University | NAAC A++

Laboratory Manual

Session 2023-24

Parul® University NAAC A++

Faculty of Engineering & Technology
Subject Name : OS Laboratory
Subject Code : 303105252
B.Tech CSE Year 2nd Semester 4th

CERTIFICATE

This is to Certify that

Mr./Ms. **Balar om sanjaybhai** With enrolment no.**2203051050389**

has successfully completed his/her Laboratory experiments in

Operating System laboratory (303105252) From the department of

COMPUTER SCIENCE AND ENGINEERING during the

academic year 2023 – 2024.

Date of Submission : …….

Staff In Charge: …….

Head of department: …….

ENGINEER

TABLE OF CONTENT

PRACTICAL – 1

AIM: 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

   OUTPUT:

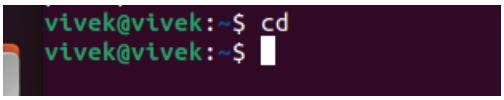
```
vivek@vivek:~$ pwd
/home/vivek
vivek@vivek:~$
```

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.

   SYNTAX:
   cd directory_name

   EXAMPLE: To change into “work directory” in “KALI”

OUTPUT:

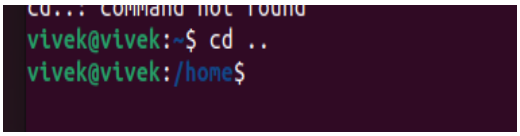


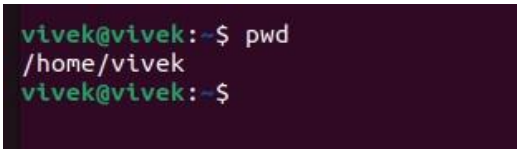
3. **cd ..**

DESCRIPTION:
Move up one directory.

SYNTAX:
cd ..

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





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 directory, their permissions, the number of subdirectories in directories listed, the size of the file, and the date of last modification. ls -l
• List all files including hidden files ls -a

OUTPUT:

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

OUTPUT:

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:

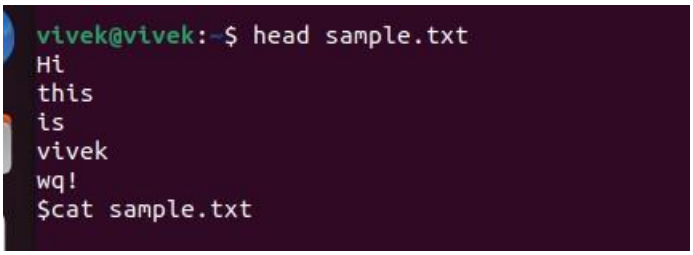
```
[root@localhost ~]# #!/bin/bash
[root@localhost ~]#
[root@localhost ~]# echo "Enter a Number:"
Enter a Number:
[root@localhost ~]#
[root@localhost ~]# read n
40
[root@localhost ~]# temp=$n
[root@localhost ~]# sd=0
[root@localhost ~]# sum=0
[root@localhost ~]# while [ $n -gt 0 ]
> do
>      sd=$(($n % 10))
>      n=$(($n/10))
>      sum=$(($sum + $sd))
> done
[root@localhost ~]# echo "Sum is $sum"
Sum is 4
[root@localhost ~]#
```

head [option]…[file/directory]

EXAMPLE:

Display the first ten lines of myfile.txt. head myfile.txt

OUTPUT:



7. **tail**

DESCRIPTION:

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

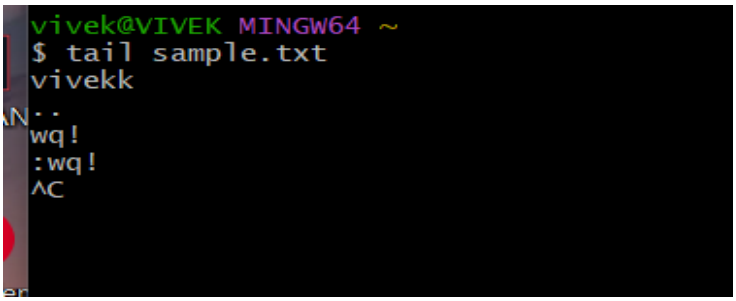
VSYNTAX:

tail [option]…[file/directory]

EXAMPLE:

Output the last 100 lines of the file myfile.txt. tail myfile.txt -n 100

OUTPUT:



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).

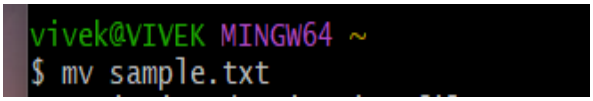
SYNTAX: mv

[option] source directory

EXAMPLE:

• Moves the file myfile.txt to the directory destination-directory. mv myfile.txt destination_directory

OUTPUT:



9. **mkdir :**

Make Directory

DESCRIPTION:

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 210304124456. mkdir work

OUTPUT:

```
vivek@VIVEK MINGW64 ~
$ mkdir d1
```

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

OUTPUT:

```
vivek@VIVEK MINGW64 ~
$ cp sample.txt sample1.txt
```

11. **rmdir :**
Romove 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. rmdir -rf mydir

OUTPUT:

```
vivek@VIVEK MINGW64 ~
$ rmdir d1
```

12. **echo**
DESCRIPTION:
Display text on the screen.

SYNTAX: echo yourtext

EXAMPLE: Print Hello World on the screen echo "Hello World"

OUTPUT:

```
vivek@vivek:~$ echo "Hello vivekk"
Hello vivekk
vivek@vivek:~$
```

13. **clear**

DESCRIPTION:

Used to clear the screen

SYNTAX: clear

EXAMPLE: Clear the entire screen

OUTPUT:

```
vivek@vivek:~$
```

14. **date :**

DESCRIPTION:

display current date and time.

OUTPUT:

```
[root@localhost ~]# date
Mon Sep 11 03:18:28 PM UTC 2023
[root@localhost ~]#
```

15. **cal:**

DESCRIPTION:

display current month calendar.

OUTPUT:

```
[root@localhost ~]# cal
   September 2023
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

16. **touch:**

DESCRIPTION:

create a new file

OUTPUT:

```
[root@localhost ~]# touch t1.txt
[root@localhost ~]# ls
bench.py  hello.c  t1.txt
[root@localhost ~]#
```

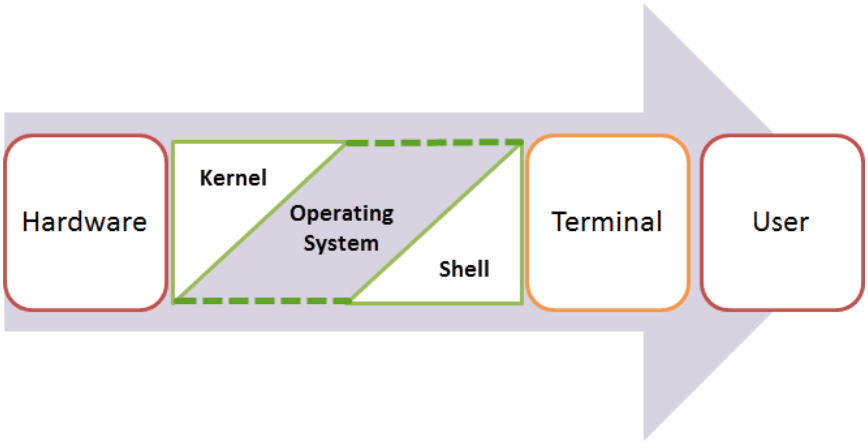
PRACTICAL – 2

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**.

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.

PROGRAM TO UNDERSTAND BASIC OF SHELL PROGRAMING:

```
#!/bin/sh

x=10

y=11

if [ $x-ne $y ]

then

echo "Not equal"

fi
```

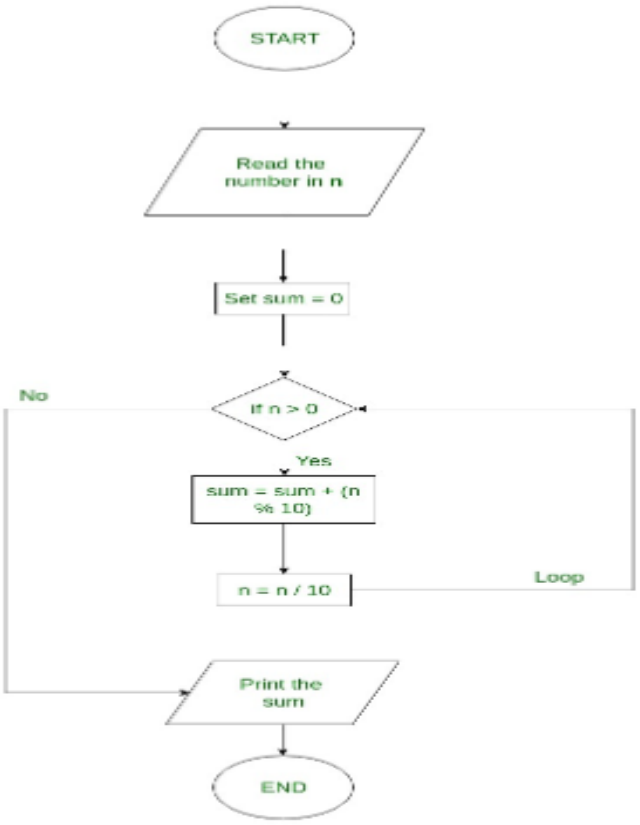
OUTPUT:

```
[root@localhost ~]# #!/bin/sh
[root@localhost ~]# x=10
[root@localhost ~]# y=11
[root@localhost ~]# if [ $x -ne $y ]
> then
> echo "Not equal"
> fi
Not equal
[root@localhost ~]#
```

PRACTICAL – 3

AIM: Write a shell script to print given numbers sum of all digits

Flow chart :



Flowchart to find sum of digits of a number

CODE:

```
#!/bin/bash

echo "Enter a Number:"
read n
temp=$n
sd=0
sum=0
while [ $n-gt 0 ]
do
sd=$(($n % 10))
n=$(($n/10))
sum=$(($sum + $sd))
done
echo "Sum is $sum"
```

PRACTICAL – 3

OUTPUT:

```
Enter a number
100
1
```

```
Enter a number
786
21
```

PRACTICAL – 4

AIM: Write a shell script to validate the entered date. (eg. Date format is: dd-mm-yyyy).

Flow chart :

CODE:

```
echo "Enter Valid Date"
read date
echo "You have entered $date"
date -d $date
if [ $? -eq 0]
then
echo "Enter Date is Valid"
else
echo "Enter Date is Invalid"
fi
```

PRACTICAL – 4

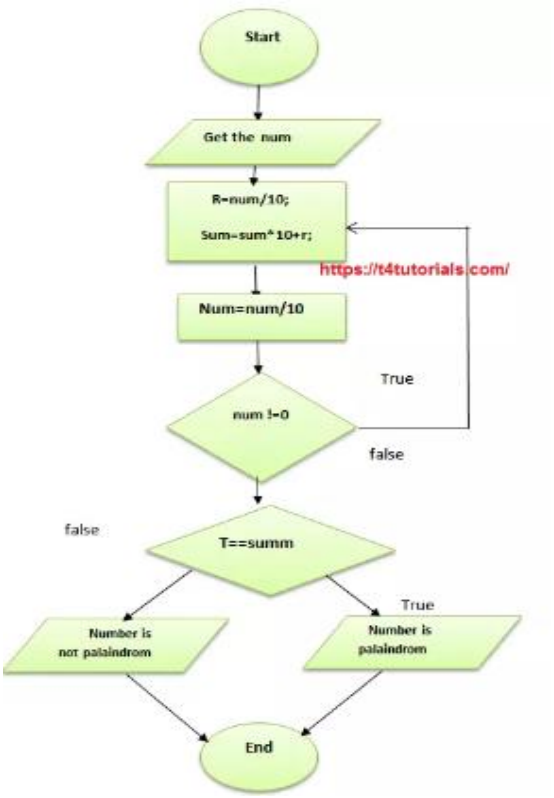
OUTPUT:

```
[root@localhost ~]# echo "enter date"
enter date
[root@localhost ~]# read date
01/01/2024
[root@localhost ~]# echo "you have entered date $date"
you have entered date 01/01/2024
[root@localhost ~]# date -d $date
Mon Jan  1 12:00:00 AM UTC 2024
[root@localhost ~]# if [ $? -eq 0 ]
> then
> echo "valid date"
> else
> echo "invalid date"
> fi
valid date
[root@localhost ~]#
```

PRACTICAL – 5

AIM: Write a shell script to check entered string is palindrome or not**.**

Flow chart :



CODE:

```
#!/bin/bash read

name echo ""

name1=`echo $name | rev

if [ "$name" == "$name1" ]

then

echo "**$name is palindrome**"

else

echo "**$name is not a palindrome**"
fi
```

PRACTICAL – 5

OUTPUT:

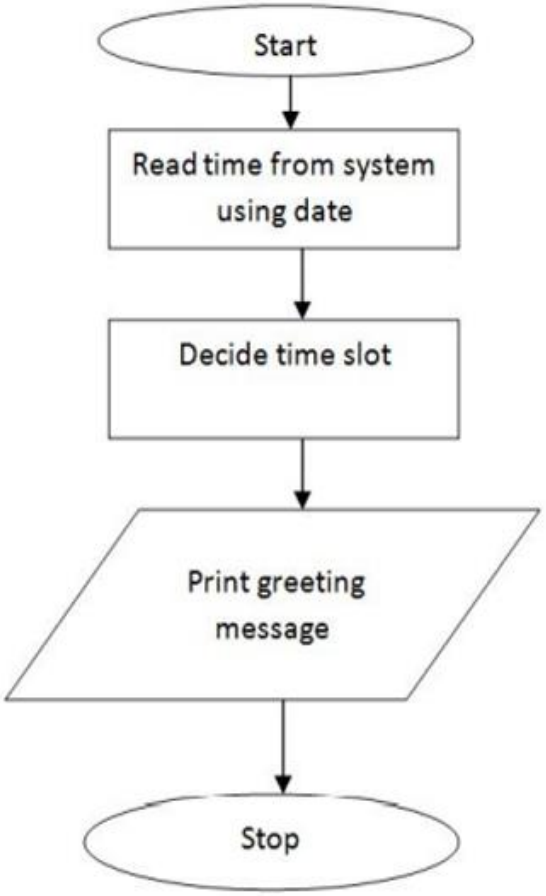
```
[root@localhost ~]# #!/bin/bash
[root@localhost ~]# read name
madam
[root@localhost ~]# echo " "

[root@localhost ~]# name1=`echo $name | rev`
[root@localhost ~]# if [ "$name" == "$name1" ]
> then
> echo "**$name is palindrome**"
> else
> echo "**$name is not a palindrome**"
> fi
**madam is palindrome**
[root@localhost ~]#
```

PRACTICAL – 6

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

Flow chart :



CODE:

```
#!/bin/bash
hour=`date +%H`
if [ $hour -lt 12 ] # if hour is less than 12
then
echo "GOOD MORNING WORLD"
elif [ $hour -le 16 ] # if hour is less than equal to 16
then
echo "GOOD AFTERNOON WORLD"
elif [ $hour -le 20 ] # if hour is less than equal to 20
then
echo "GOOD EVENING WORLD"
else
echo "GOOD NIGHT WORLD"
fi
```

PRACTICAL – 6

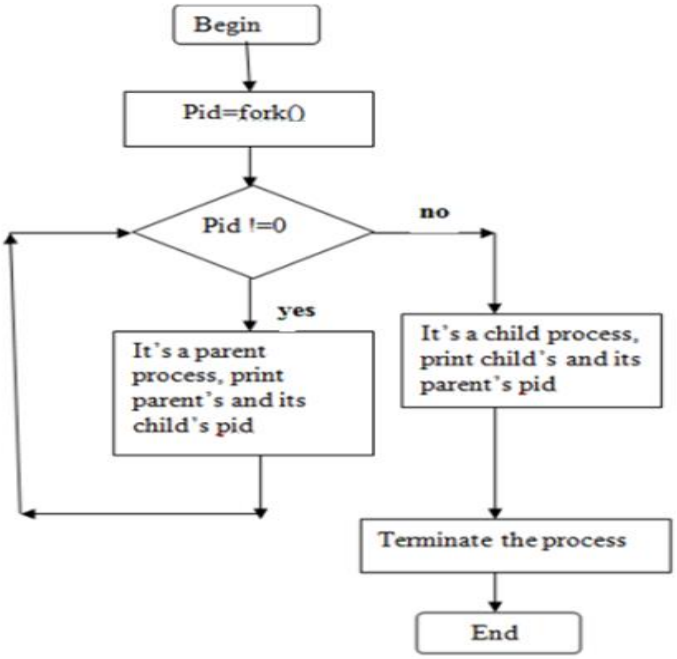
OUTPUT:

```
[root@localhost ~]# #!/bin/bash
[root@localhost ~]# hour=`date +%H`
[root@localhost ~]# if [ $hour -lt 12 ] # if hour is less than 12
> then
> echo "GOOD MORNING WORLD"
> elif [ $hour -le 16 ] # if hour is less than equal to 16
> then
> echo "GOOD AFTERNOON WORLD"
> elif [ $hour -le 20 ] # if hour is less than equal to 20
> then
> echo "GOOD EVENING WORLD"
> else
> echo "GOOD NIGHT WORLD"
> fi
GOOD AFTERNOON WORLD
[root@localhost ~]#
```

PRACTICAL – 7

AIM: Write a shell script to check entered string is palindrome or not**.**

Flow chart :



CODE:

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

PRACTICAL – 7

OUTPUT:

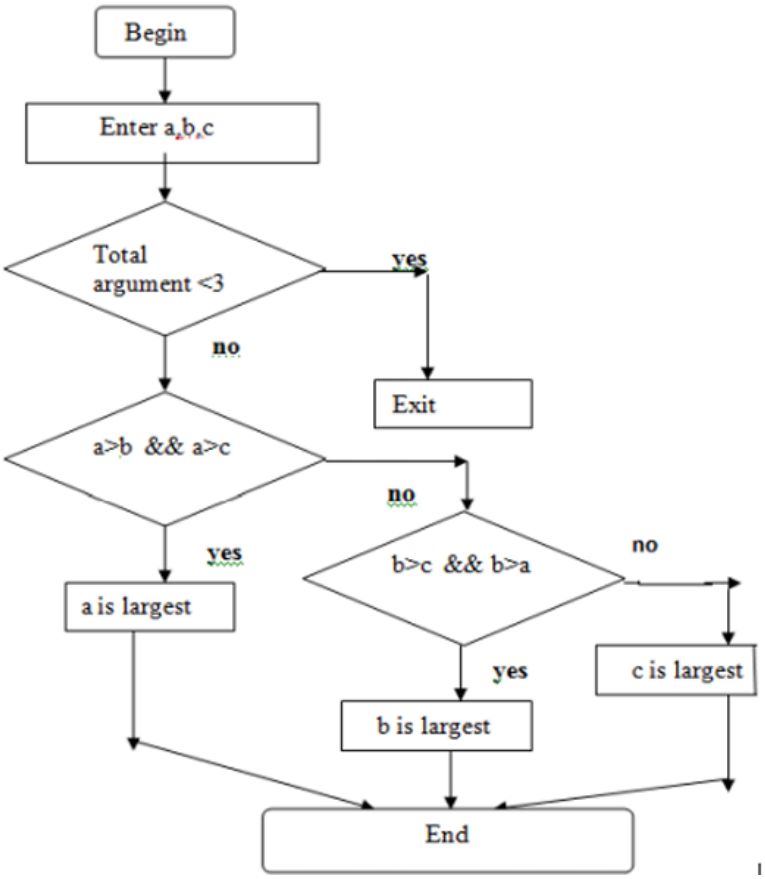
Output:

```
Hello World!
I am the child process.
Hello World!
I am the parent process.
```

PRACTICAL – 8

AIM: Find out the biggest number from given three numbers supplied as command line arguments.

Flow chart :



CODE:

```
#!/bin/bash
echo "enter number 1"
read n1
echo "enter number 2"
read n2
echo "enter number 3"
read n3
   if [ $n1 -gt $n2 ] && [ $n1 -gt $n3 ]
   then
   echo "Number 1 is biggest: $n1"
   elif [ $n2 -gt $n1 ] && [ $n2 -gt $n3 ]
   then
   echo "Number 2 is biggest: $n2"
   else
      echo "Number 3 is biggest: $n3"
   fi
```

PRACTICAL – 8

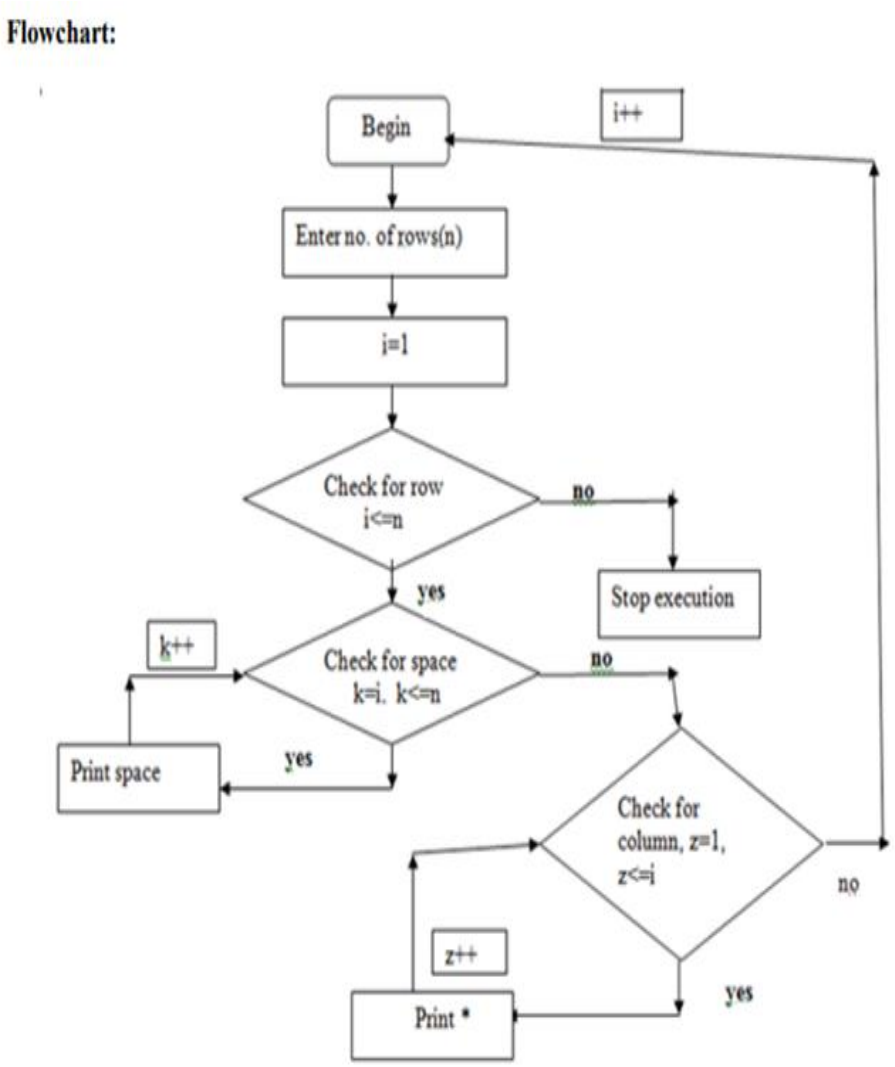
OUTPUT:

```
[root@localhost ~]# #!/bin/bash
[root@localhost ~]# echo "enter number 1"
enter number 1
[root@localhost ~]# read n1
45
[root@localhost ~]# echo "enter number 2"
enter number 2
[root@localhost ~]# read n2
65
[root@localhost ~]# echo "enter number 3"
enter number 3
[root@localhost ~]# read n3
85
[root@localhost ~]# if [ $n1 -gt $n2 ] && [ $n1 -gt $n3 ]
> then
> echo "Number 1 is biggest: $n1"
> elif [ $n2 -gt $n1 ] && [ $n2 -gt $n3 ]
> then
> echo "Number 2 is biggest: $n2"
> else
> echo "Number 3 is biggest: $n3"
> fi
Number 3 is biggest: 85
[root@localhost ~]#
```

PRACTICAL – 9

AIM: Printing patterns using loops.

Flow chart :



CODE:

```
#! /bin/sh
for ((i=1;i<=5;i++));          # Loop to create 5 lines of text
do
 for((k=1;k<=(5-i);k++));      # Loop to padd prepending spaces
 do
  printf "%s" " ";
 done;
 for ((j=1;j<=i;j++));         # Loop to create asterix
 do
  printf "%s" "*";
 done;
 printf "\n";                  # Print the carriage return
done
```

PRACTICAL – 9

OUTPUT:

```
[root@localhost ~]# #! /bin/sh
[root@localhost ~]# for ((i=1;i<=5;i++));
> do
> for((k=1;k<=(5-i);k++));
> do
> printf "%s" " ";
> done;
> for ((j=1;j<=i;j++));
> do
> printf "%s" "*";
> done;
> printf "\n";
> done
    *
   **
  ***
 ****
*****
[root@localhost ~]#
```

PRACTICAL – 10

AIM: Shell script to determine whether given file exist or not.

CODE:

```
#!/bin/bash

if [ -f "File.txt" ];
then

# if file exist the it will be printed
echo "File is exist"
else

# is it is not exist then it will be printed
echo "File is not exist"
fi
```

OUTPUT:

```
[root@localhost ~]# touch Engineer.txt
[root@localhost ~]# #!/bin/bash
[root@localhost ~]# ls
bench.py  Engineer.txt  hello.c
[root@localhost ~]# if [ -f "Engineer.txt" ];
> then
> echo "file exist"
> else
> echo "file not exist"
> fi
file exist
[root@localhost ~]#
```

```
[root@localhost ~]# #!/bin/bash
[root@localhost ~]# if [ -f "File.txt" ];
> then
> echo "File is exist"
> else
> echo "File is not exist"
> fi
File is not exist
[root@localhost ~]#
```

PRACTICAL – 11

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

CODE:

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

OUTPUT:

Output:

```
PROCESS CREATION
I am the child process.
PROCESS CREATION
I am the parent process.
```

PRACTICAL – 12

AIM: Implementation of FCFS & Round Robin Algorithm

CODE:

Practical 12.1: FCFS

```
#include<stdio.h>

int main(){

   int bt[10]={0},at[10]={0},tat[10]={0},wt[10]={0},ct[10]={0};
   int n,sum=0;
   float totalTAT=0,totalWT=0;

   printf("Enter number of processes");
   scanf("%d",&n);

   printf("Enter arrival time and burst time for each process\n\n");

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

      printf("Arrival time of process[%d]    ",i+1);
      scanf("%d",&at[i]);

      printf("Burst time of process[%d]    ",i+1);
      scanf("%d",&bt[i]);

      printf("\n");
   }

   //calculate completion time of processes

   for(int j=0;j<n;j++)
   {
      sum+=bt[j];
      ct[j]+=sum;
   }

   //calculate turnaround time and waiting times

   for(int k=0;k<n;k++)
   {
      tat[k]=ct[k]-at[k];
      totalTAT+=tat[k];
   }
```

```
    for(int k=0;k<n;k++)
    {
        wt[k]=tat[k]-bt[k];
        totalWT+=wt[k];
    }

    printf("Solution: \n\n");
    printf("P#\t AT\t BT\t CT\t TAT\t WT\t\n\n");

    for(int i=0;i<n;i++)
    {
        printf("P%d\t %d\t %d\t %d\t %d\t %d\n",i+1,at[i],bt[i],ct[i],tat[i],wt[i]);
    }

    printf("\n\nAverage Turnaround Time = %f\n",totalTAT/n);
    printf("Average WT = %f\n\n",totalWT/n);

    return 0;
}
```

OUTPUT:

```
Enter number of processes2
Enter arrival time and burst time for each process

Arrival time of process[1]    0
Burst time of process[1]    2

Arrival time of process[2]    4
Burst time of process[2]    2

Solution:

P#      AT      BT      CT      TAT     WT

P1      0       2       2       2       0
P2      4       2       4       0       -2


Average Turnaround Time = 1.000000
Average WT = -1.000000
```

Practical 12.2: Round Robin Algorithm

```
#include<stdio.h>

int main()
{
```

```
int count,j,n,time,remain,flag=0,time_quantum;
int wait_time=0,turnaround_time=0,at[10],bt[10],rt[10];
printf("Enter Total Process:\t ");
scanf("%d",&n);
remain=n;
for(count=0;count<n;count++)
{
 printf("Enter Arrival Time and Burst Time for Process Process Number %d :",count+1);
 scanf("%d",&at[count]);
 scanf("%d",&bt[count]);
 rt[count]=bt[count];
}
printf("Enter Time Quantum:\t");
scanf("%d",&time_quantum);
printf("\n\nProcess\t|Turnaround Time|Waiting Time\n\n");
for(time=0,count=0;remain!=0;)
{
 if(rt[count]<=time_quantum && rt[count]>0)
 {
  time+=rt[count];
  rt[count]=0;
  flag=1;
 }
 else if(rt[count]>0)
 {
  rt[count]-=time_quantum;
  time+=time_quantum;
 }
 if(rt[count]==0 && flag==1)
 {
  remain--;
  printf("P[%d]\t|\t%d\t|\t%d\n",count+1,time-at[count],time-at[count]-bt[count]);
  wait_time+=time-at[count]-bt[count];
  turnaround_time+=time-at[count];
  flag=0;
 }
 if(count==n-1)
  count=0;
 else if(at[count+1]<=time)
  count++;
 else
  count=0;
}
printf("\nAverage Waiting Time= %f\n",wait_time*1.0/n);
printf("Avg Turnaround Time = %f",turnaround_time*1.0/n);

return 0;
```

}

OUTPUT:

```
Enter number of processes3
Enter arrival time and burst time for each process

Arrival time of process[1]    0
Burst time of process[1]    2

Arrival time of process[2]    3
Burst time of process[2]    4

Arrival time of process[3]    6
Burst time of process[3]    6

Solution:

P#      AT      BT      CT      TAT     WT

P1      0       2       2       2       0
P2      3       4       6       3       -1
P3      6       6       12      6       0


Average Turnaround Time = 3.666667
Average WT = -0.333333
```

PRACTICAL – 13

AIM: Implementation of Banker's Algorithm.

CODE:

```
/*<--------------PREPROCESSING  STATEMENTS ---------------- >*/
#include <stdio.h>
#include <stdlib.h>

/*<--------------MAIN FUNCTION---------------->*/

int main()
{
int Max[10][10], need[10][10], alloc[10][10], avail[10], completed[10], safeSequence[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++)
        need[i][j] = Max[i][j] - alloc[i][j];

do
{
    printf("\n Max matrix:\tAllocation matrix:\n");
    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");
    }

    process = -1;

    for(i = 0; i < p; i++)
    {
        if(completed[i] == 0)//if not completed
        {
            process = i ;
            for(j = 0; j < r; j++)
            {
                if(avail[j] < need[i][j])
                {
                    process = -1;
                    break;
                }
            }
        }
        if(process != -1)
            break;
    }

    if(process != -1)
    {
        printf("\nProcess %d runs to completion!", process + 1);
        safeSequence[count] = process + 1;
        count++;
        for(j = 0; j < r; j++)
        {
            avail[j] += alloc[process][j];
            alloc[process][j] = 0;
            Max[process][j] = 0;
            completed[process] = 1;
        }
    }
```

```
}while(count != p && process != -1);

if(count == p)
{
   printf("\nThe system is in a safe state!!\n");
   printf("Safe Sequence : < ");
   for( i = 0; i < p; i++)
         printf("%d  ", safeSequence[i]);
   printf(">\n");
}
else
   printf("\nThe system is in an unsafe state!!");
}
```

OUTPUT:

```
Enter number of processes4
Enter arrival time and burst time for each process

Arrival time of process[1]    2
Burst time of process[1]    56

Arrival time of process[2]    3
Burst time of process[2]    56

Arrival time of process[3]    3
Burst time of process[3]    67

Arrival time of process[4]    4
Burst time of process[4]    56

Solution:

P#      AT      BT      CT      TAT     WT

P1      2       56      56      54      -2
P2      3       56      112     109     53
P3      3       67      179     176     109
P4      4       56      235     231     175


Average Turnaround Time = 142.500000
Average WT = 83.750000
```