2OEUCS015 - ARTHIKA G

Ex.No : 2 **PROGRAMS USING SHELL PROGRAMMING**

Date : 13/08/2021

AIM:

To write a programs using shell programming.

DESCRIPTION:

A Linux shell is a command language interpreter, the primary purpose of which is to translate the command lines typed at the terminal into system actions. The shell itself is a program, through which other programs are invoked

What is a shell script?

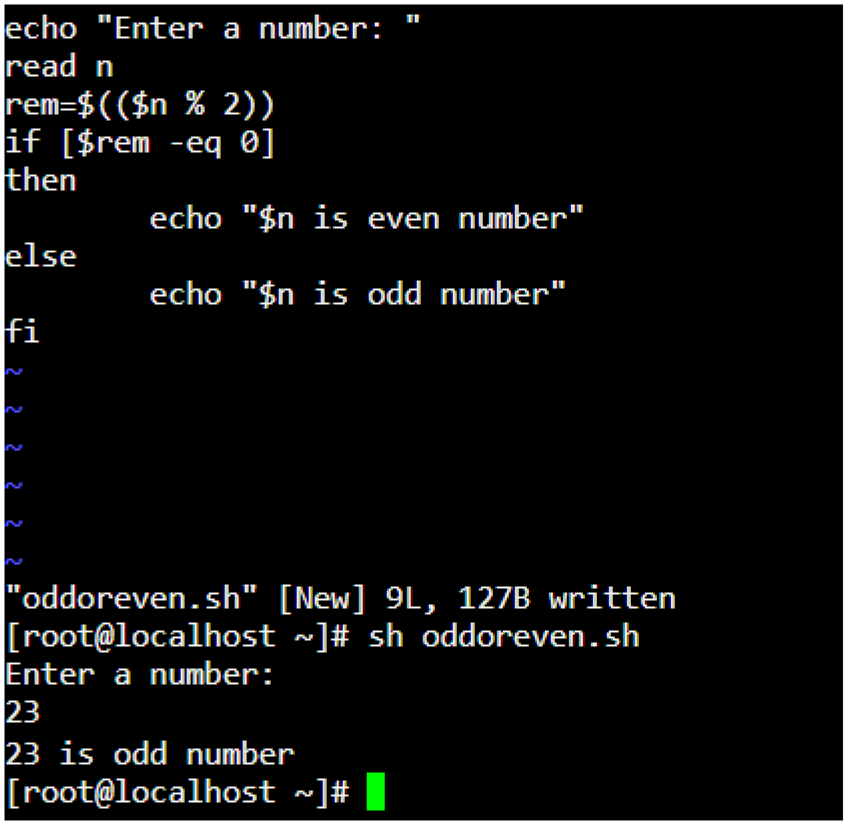
- A shell script is a file containing a list of commands to be executed by the Linux shell. shell script provides the ability to create your own customized Linux commands
- Linux shell have sophisticated programming capabilities which makes shell script powerful Linux tools

PROGRAMS:

1. **Write a Shell program to check the given number is even or odd.**

   Sample Input :
   Enter any number
   23
   Sample Output : **23** is odd

Result:

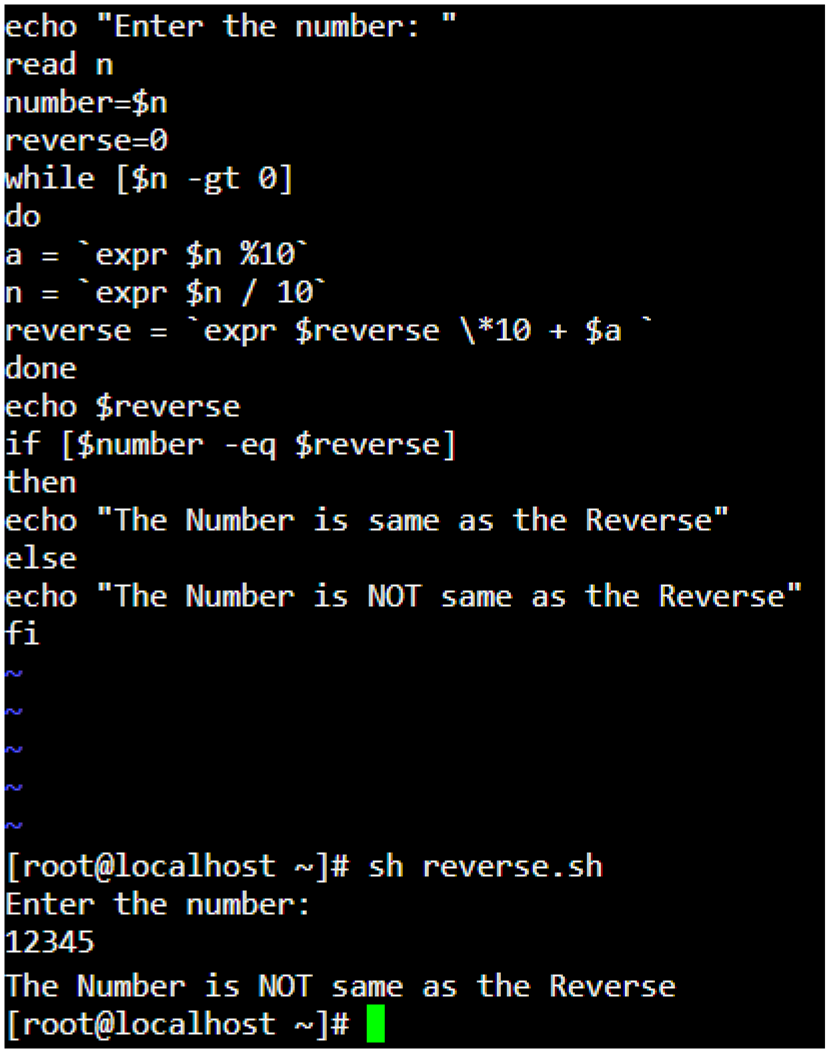


```
echo "Enter a number: "
read n
rem=$(($n % 2))
if [$rem -eq 0]
then
        echo "$n is even number"
else
        echo "$n is odd number"
fi
~
~
~
~
~
~
"oddoreven.sh" [New] 9L, 127B written
[root@localhost ~]# sh oddoreven.sh
Enter a number:
23
23 is odd number
[root@localhost ~]#
```

2. **Write a Shell program to check the given number and its reverse are same.**

   Sample Input :
   Enter a number
   252
   Sample Output : The given number and its reverse are same

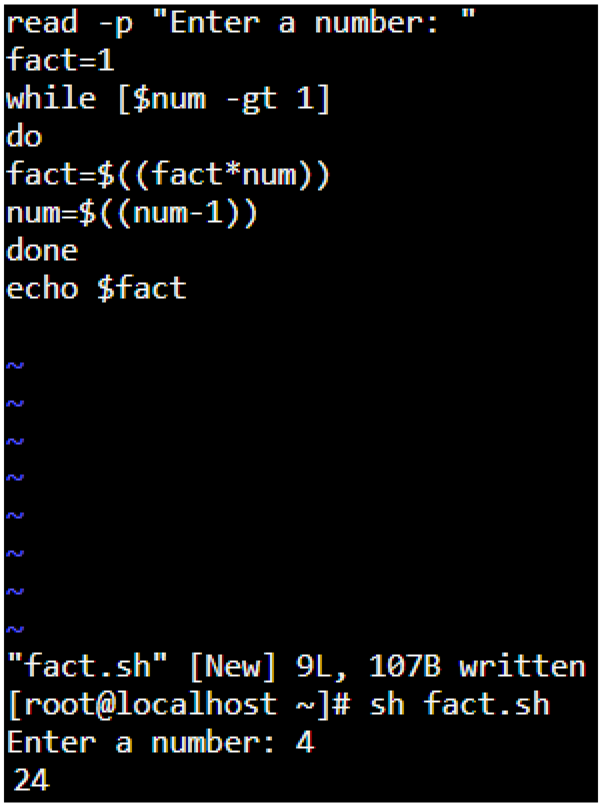
Result:



```
echo "Enter the number: "
read n
number=$n
reverse=0
while [$n -gt 0]
do
a = `expr $n %10`
n = `expr $n / 10`
reverse = `expr $reverse \*10 + $a `
done
echo $reverse
if [$number -eq $reverse]
then
echo "The Number is same as the Reverse"
else
echo "The Number is NOT same as the Reverse"
fi
~
~
~
~
~
[root@localhost ~]# sh reverse.sh
Enter the number:
12345
The Number is NOT same as the Reverse
[root@localhost ~]#
```

3. **Write a Shell Program to find a factorial of a number.**
   Sample Input :
   Enter number
   5
   Sample Output : 120

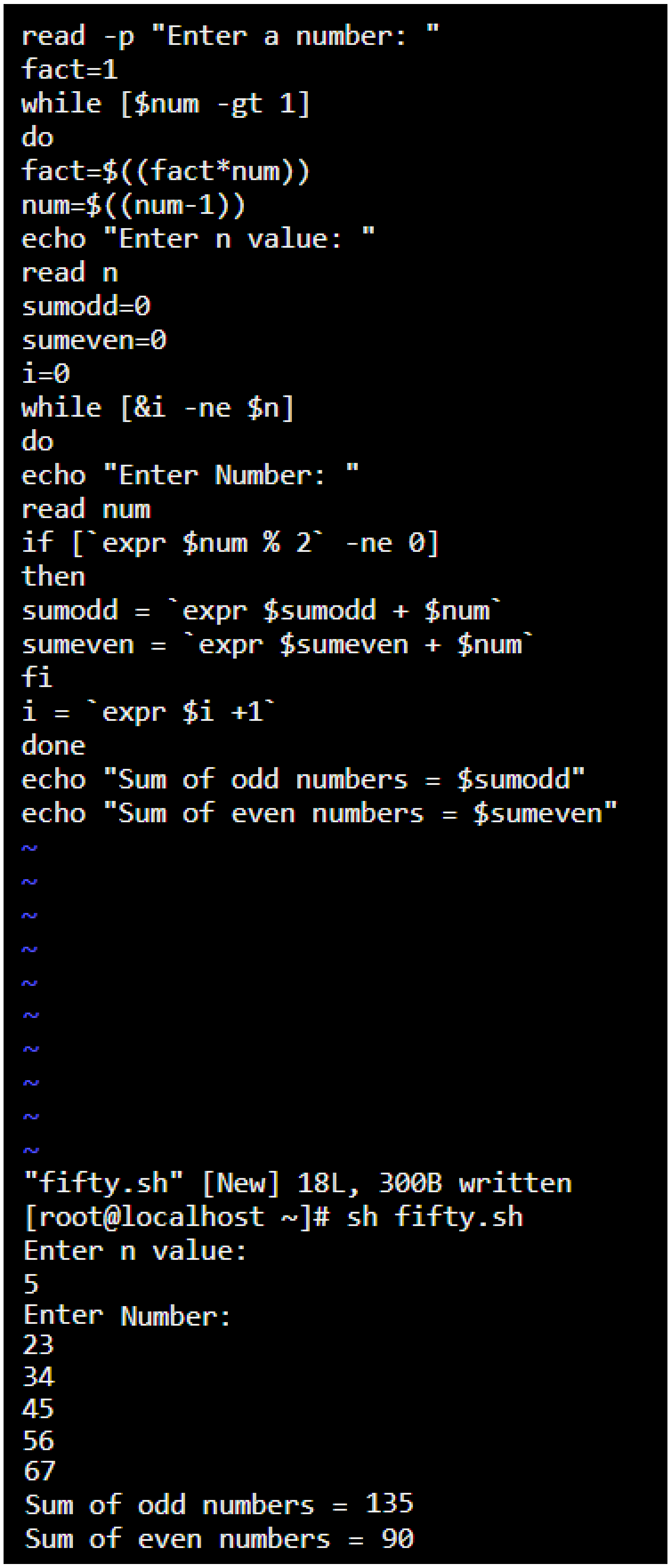
   Result:



```
read -p "Enter a number: "
fact=1
while [$num -gt 1]
do
fact=$((fact*num))
num=$((num-1))
done
echo $fact

~
~
~
~
~
~
~
~
"fact.sh" [New] 9L, 107B written
[root@localhost ~]# sh fact.sh
Enter a number: 4
 24
```

4. **Write a Shell Program for finding sum of Odd and Even numbers up to 'N'.**
   Sample Input :
   Enter the number of elements
   5
   Enter the number
   23
   34
   45
   56
   67
   Sample Output :
   The sum of odd numbers is : 135
   The sum of even numbers is : 90

Result:



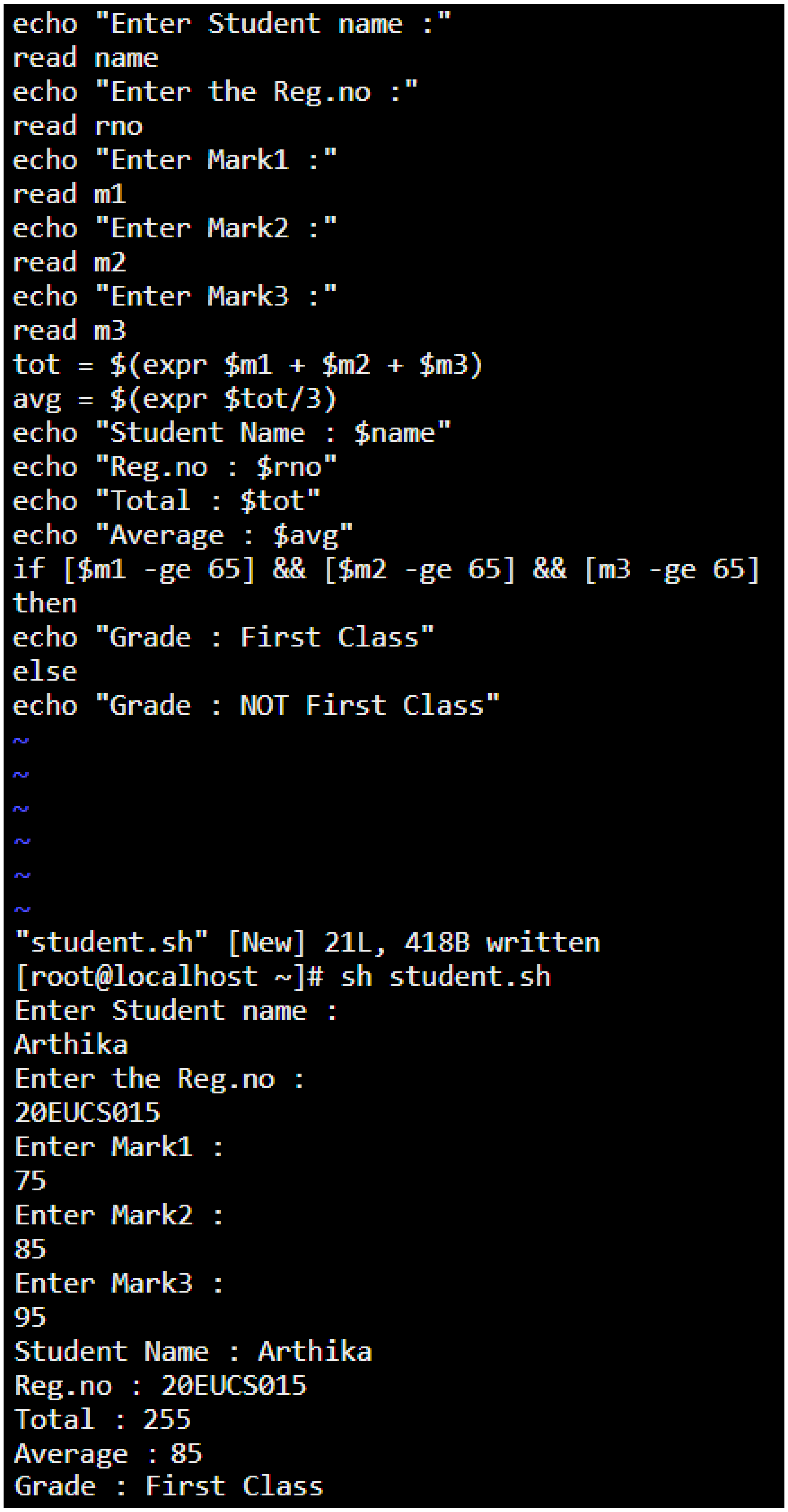
```
read -p "Enter a number: "
fact=1
while [$num -gt 1]
do
fact=$((fact*num))
num=$((num-1))
echo "Enter n value: "
read n
sumodd=0
sumeven=0
i=0
while [&i -ne $n]
do
echo "Enter Number: "
read num
if [`expr $num % 2` -ne 0]
then
sumodd = `expr $sumodd + $num`
sumeven = `expr $sumeven + $num`
fi
i = `expr $i +1`
done
echo "Sum of odd numbers = $sumodd"
echo "Sum of even numbers = $sumeven"
~
~
~
~
~
~
~
~
~
~
"fifty.sh" [New] 18L, 300B written
[root@localhost ~]# sh fifty.sh
Enter n value:
5
Enter Number:
23
34
45
56
67
Sum of odd numbers = 135
Sum of even numbers = 90
```

5. **Write a Shell program to display student grades.**

Sample Output :

```
Roll no.        Name        Total        Average    Grade

19skcet001        Anil      201        67           First class
```

Result:



```
echo "Enter Student name :"
read name
echo "Enter the Reg.no :"
read rno
echo "Enter Mark1 :"
read m1
echo "Enter Mark2 :"
read m2
echo "Enter Mark3 :"
read m3
tot = $(expr $m1 + $m2 + $m3)
avg = $(expr $tot/3)
echo "Student Name : $name"
echo "Reg.no : $rno"
echo "Total : $tot"
echo "Average : $avg"
if [$m1 -ge 65] && [$m2 -ge 65] && [m3 -ge 65]
then
echo "Grade : First Class"
else
echo "Grade : NOT First Class"
~
~
~
~
~
~
"student.sh" [New] 21L, 418B written
[root@localhost ~]# sh student.sh
Enter Student name :
Arthika
Enter the Reg.no :
20EUCS015
Enter Mark1 :
75
Enter Mark2 :
85
Enter Mark3 :
95
Student Name : Arthika
Reg.no : 20EUCS015
Total : 255
Average : 85
Grade : First Class
```

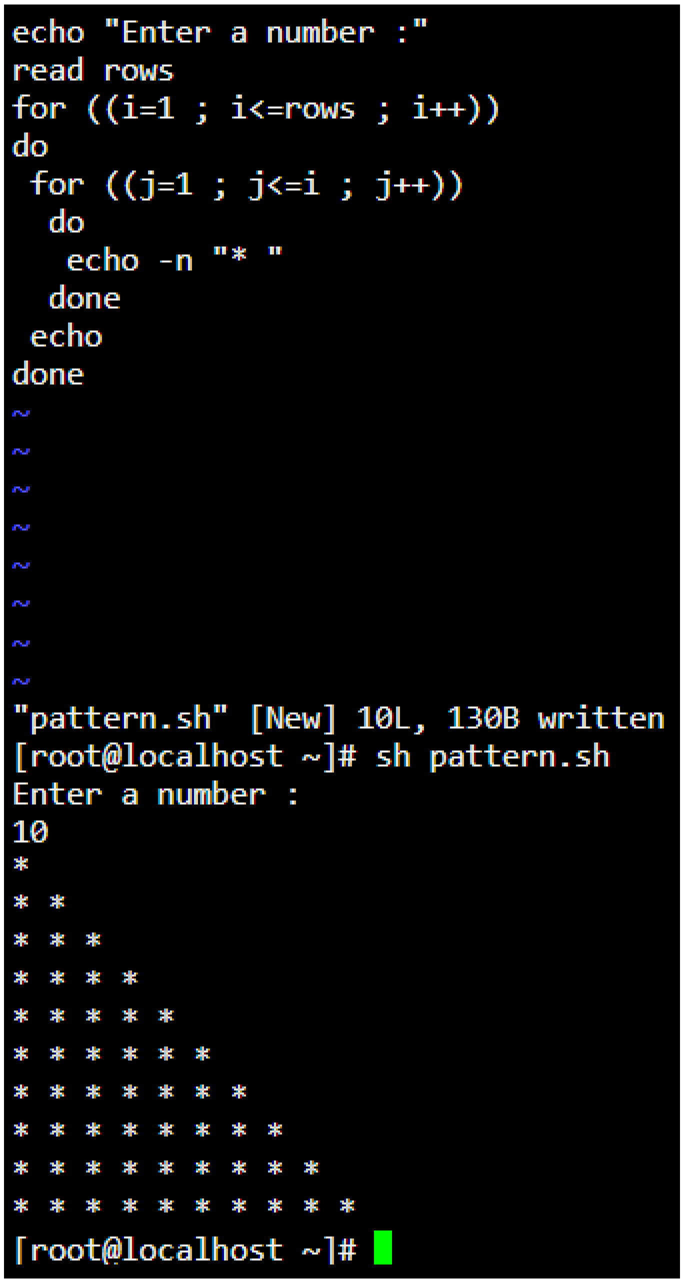
6. **Write a Shell Program to have to print half pyramid using for loop.**

Sample Input : n=10

Sample Output :

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
* * * * * * * *
* * * * * * * * *
* * * * * * * * * *
```

Result:



```
echo "Enter a number :"
read rows
for ((i=1 ; i<=rows ; i++))
do
 for ((j=1 ; j<=i ; j++))
  do
   echo -n "* "
  done
 echo
done
~
~
~
~
~
~
~
~
"pattern.sh" [New] 10L, 130B written
[root@localhost ~]# sh pattern.sh
Enter a number :
10
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
* * * * * * * *
* * * * * * * * *
* * * * * * * * * *
[root@localhost ~]#
```

7. **Write a Shell Program to perform Arithmetic Operation using Case statement.**

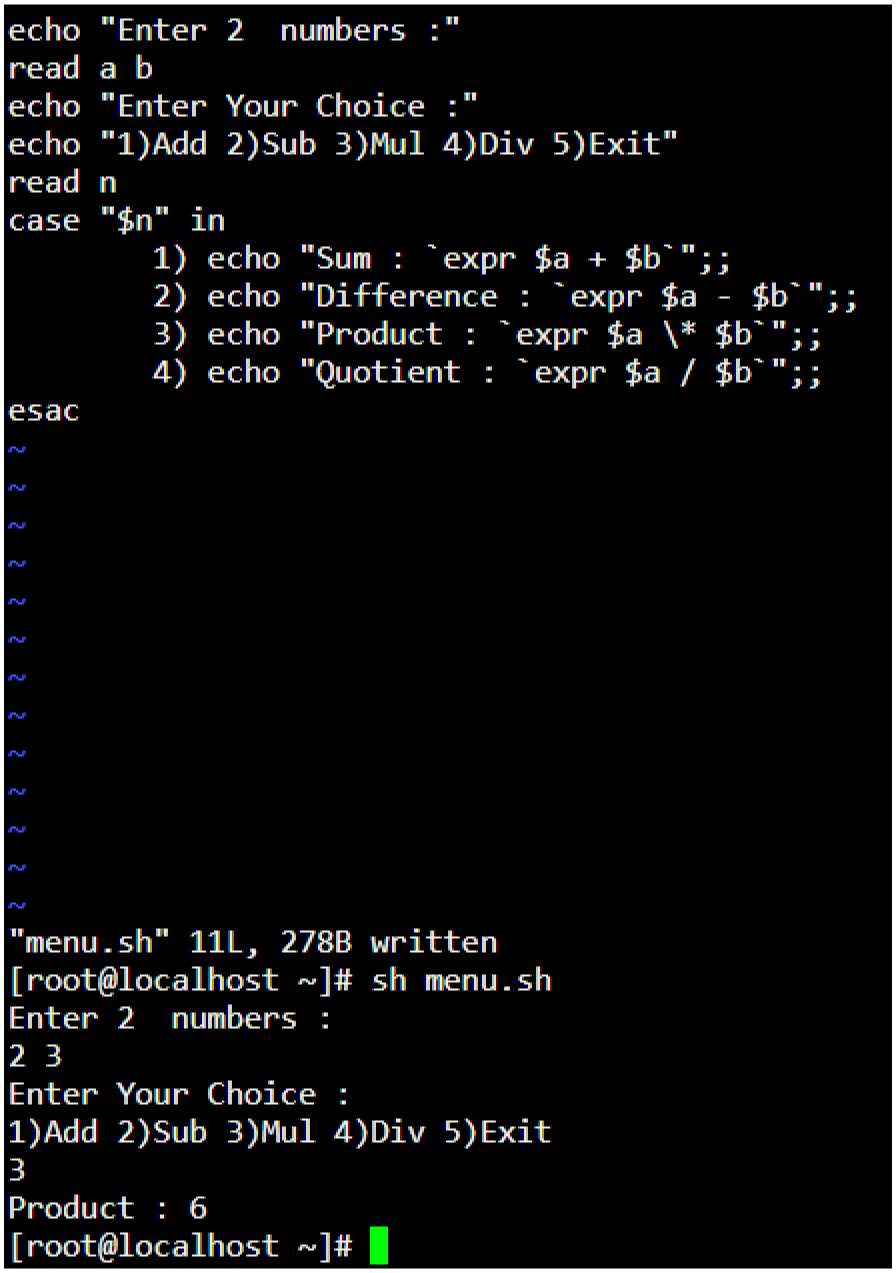
   Sample Input :

Enter two number
2 3
1. Add 2.Sub 3.Mul 4.Div 5.Exit

Enter the option: 1
Sample Output :

Addition is 5

Result:

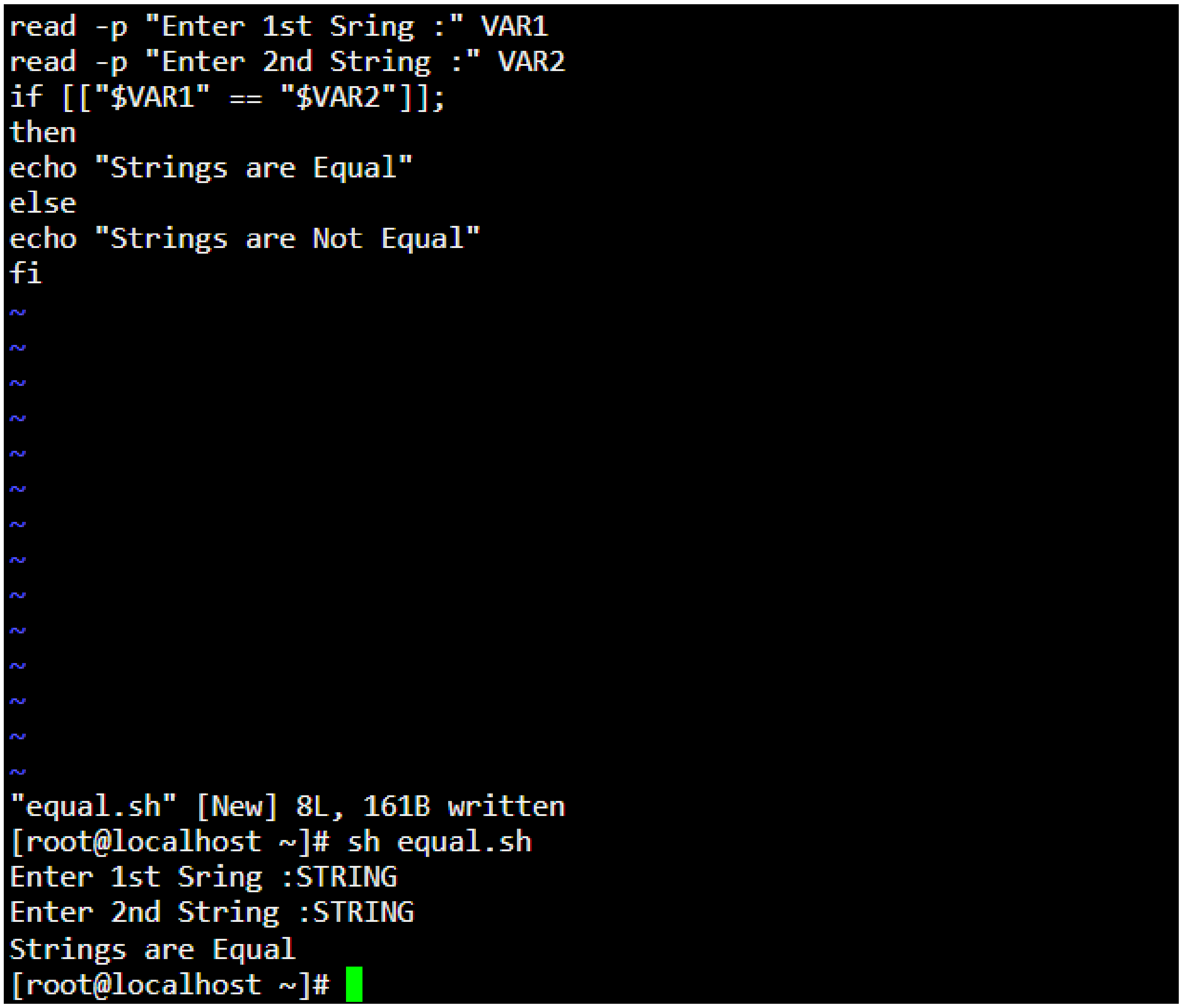


```
echo "Enter 2  numbers :"
read a b
echo "Enter Your Choice :"
echo "1)Add 2)Sub 3)Mul 4)Div 5)Exit"
read n
case "$n" in
        1) echo "Sum : `expr $a + $b`";;
        2) echo "Difference : `expr $a - $b`";;
        3) echo "Product : `expr $a \* $b`";;
        4) echo "Quotient : `expr $a / $b`";;
esac
~
~
~
~
~
~
~
~
~
~
~
~
~
"menu.sh" 11L, 278B written
[root@localhost ~]# sh menu.sh
Enter 2  numbers :
2 3
Enter Your Choice :
1)Add 2)Sub 3)Mul 4)Div 5)Exit
3
Product : 6
[root@localhost ~]#
```

8. **Write a Shell Program for comparison of strings.**
   Sample Input :
   Enter string 1: OS
   Enter string 2: OS

Sample Output :
String 1 and String 2 are identical

Result:



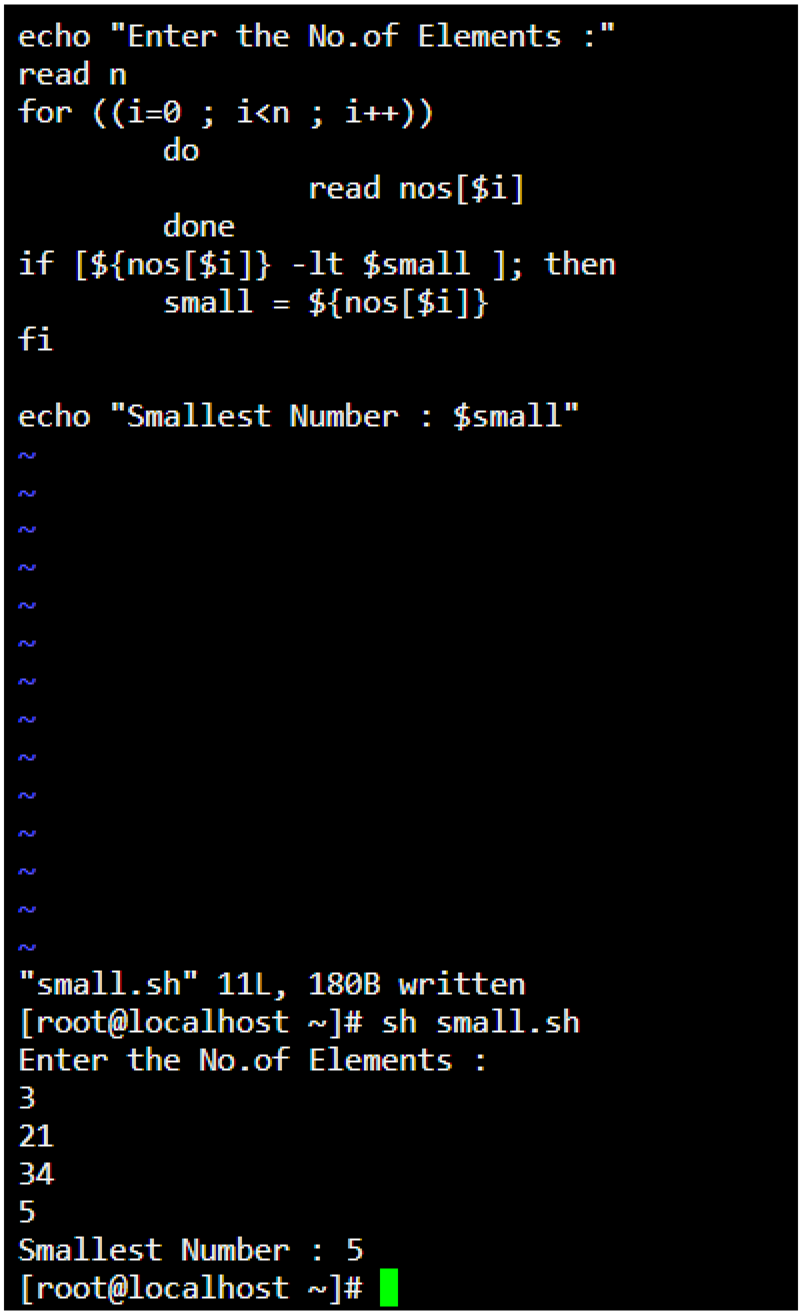
```
read -p "Enter 1st Sring :" VAR1
read -p "Enter 2nd String :" VAR2
if [["$VAR1" == "$VAR2"]];
then
echo "Strings are Equal"
else
echo "Strings are Not Equal"
fi
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"equal.sh" [New] 8L, 161B written
[root@localhost ~]# sh equal.sh
Enter 1st Sring :STRING
Enter 2nd String :STRING
Strings are Equal
[root@localhost ~]#
```

9. **Write a Shell program to find the smallest number from a set of numbers.**
Sample Input :
Enter the number of elements: 5
Enter the numbers:
34
23
45
37
56

Sample Output :

The smallest number is : 23

Result:



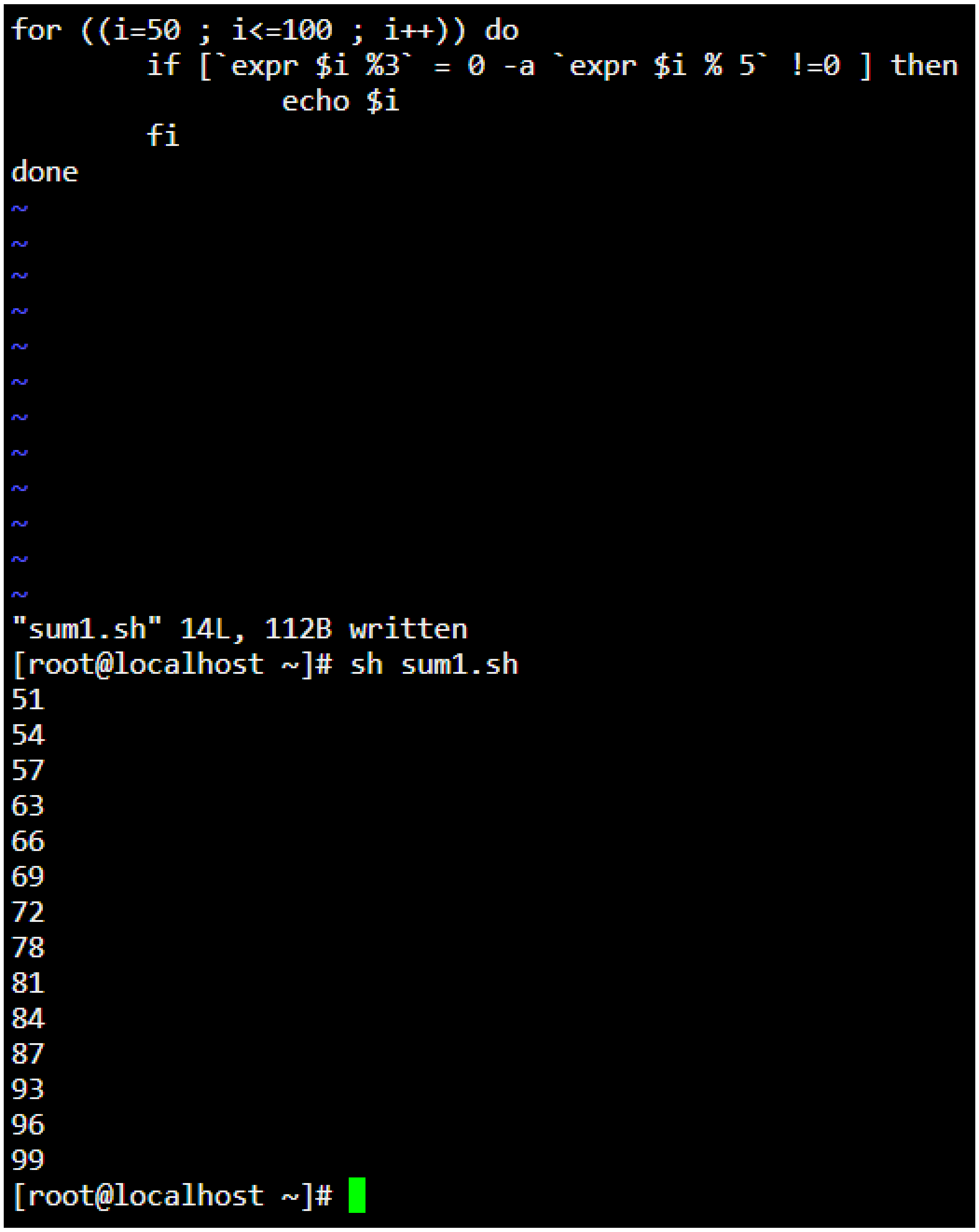
```
echo "Enter the No.of Elements :"
read n
for ((i=0 ; i<n ; i++))
        do
                read nos[$i]
        done
if [${nos[$i]} -lt $small ]; then
        small = ${nos[$i]}
fi

echo "Smallest Number : $small"
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"small.sh" 11L, 180B written
[root@localhost ~]# sh small.sh
Enter the No.of Elements :
3
21
34
5
Smallest Number : 5
[root@localhost ~]#
```

10. Write a Shell program to find the sum of all numbers between 50 and 100, which are divisible by 3 and not divisible by 5.

Sample Output :

```
The sum of all numbers between 50 and 100, which are divisible by 3 and not divisible by 5
are:
51
54
57
63
66
69
72
78
81
84
87
93
96
99
```

Result:



```
for ((i=50 ; i<=100 ; i++)) do
        if [`expr $i %3` = 0 -a `expr $i % 5` !=0 ] then
                echo $i
        fi
done
~
~
~
~
~
~
~
~
~
~
~
~
"sum1.sh" 14L, 112B written
[root@localhost ~]# sh sum1.sh
51
54
57
63
66
69
72
78
81
84
87
93
96
99
[root@localhost ~]#
```

RESULT

Thus the programs using Shell programming is successfully executed.

20EUCS015 ARTHIKA G

IMPLEMENTATION OF UNIX SYSTEM CALLS

EX NO: 3
DATE: 12/09/2021

DESCRIPTION:

When a computer is turned on, the program that gets executed first is called the ``operating system." It controls pretty much all activity in the computer. This includes who logs in, how disks are used, how memory is used, how the CPU is used, and how you talk with other computers. The operating system we use is called "Unix".

The way that programs talk to the operating system is via ``system calls." A system call looks like a procedure call but it's different -- it is a request to the operating system to perform some activity.

1. **fork( )**

Fork system call used for creating a new process, which is called ***child process***, which runs concurrently with a process (which process called system call fork) and this process is called ***parent process***. It takes no parameters and returns an integer value.
Negative Value: creation of a child process was unsuccessful.
Zero: Returned to the newly created child process.
Positive value: Returned to parent or caller. The value contains process ID of newly created child process
Syntax: fork( )

2. **wait( )**

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After the child process terminates, the parent ***continues*** its execution after the wait system call instruction.
Syntax: wait(NULL)

3. **exit( )**

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit system call. At that point, the process may return data (output) to its parent process (via the wait system call).
Syntax: exit(0)

4. **getpid( )**

It returns the process ID of the current process.
Syntax : getpid()

5. getppid( )
It returns the process ID of the parent of the current process.
Syntax : getppid()

6. perror( )
Indicate the process error.
Syntax : perror()

7. opendir( )
Open a directory.
Syntax : opendir()

8. readdir( )
Read a directory.
Syntax : readdir()

9. closedir()
Close a directory.
Syntax : closedir()

10. open()
It is used to open a file for reading, writing or both
Syntax : open(const char* path, int flags [, int mode])
Modes
O_RDONLY: read only, **O_WRONLY**: write only, **O_RDWR**: read and write, **O_CREAT**: create file if it doesn't exist, **O_EXCL**: prevent creation if it already exists.
Example:
```
int fd = open("foo.txt", O_RDONLY | O_CREAT);
printf("fd = %d/n", fd);
```

11. close ()
Tell the operating system you are done with a file descriptor and Close the file which is pointed by fd.
Syntax: close(fd);

12. read ()

The file indicated by the file descriptor fd, the read() function reads cnt bytes of input into the memory area indicated by buf. To read data from is said to be **buf.** A successful read() updates the access time for the file.
fd: file descriptor
buf: buffer to read data from
cnt: length of buffer
Syntax: read(int fd, void* buf,size_t cnt);
Example:
int fd = open("foo.txt", O_RDONLY);
read(fd, &c, 1);

13. Write()
Writes cnt bytes from buf to the file or socket associated with fd. cnt should not be greater than INT_MAX (defined in the limits.h header file). If cnt is zero, write() simply returns 0 without attempting any other action.
Syntax : write(int fd, void* buf, size_t cnt);
Example:
int fd = open("foo.txt", O_WRONLY | O_CREAT);
write(fd, "hello world", strlen("hello world"));

PROGRAMS

1) Implementation of process management using the following system calls of the UNIX operating system: **fork, wait, exec, getpid, getppid and exit.**

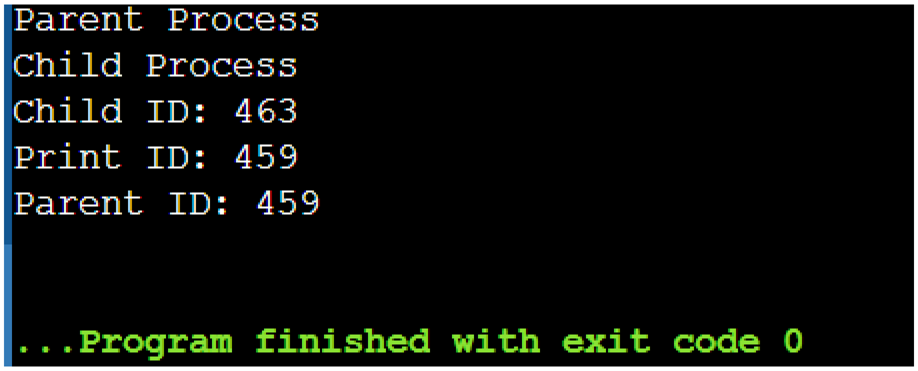
AIM: To implement the process management system call commands of fork, wait, exec, getpid, getppid and exit.

ALGORITHM:
STEP1: Declare the variable.
STEP2: Assign value to the variable by calling the function fork().
STEP3: Check if the variable is lesser than 0 and print Error.
STEP4: Else check if the variable is equal to 0 and print getpid() and getppid() and exit.
STEP5: Else call wait() function with NULL parameter, then print getpid().
PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>
int main()
```

```
{
        int pid;
        pid=fork();
        if(pid<0)
        printf("Error");
        else if(pid==0)
        {
                printf("Child Process\n");
                printf("Child ID: %d\n",getpid());
                printf("Print ID: %d\n",getppid());
                exit(0);
        }
        else    {
                printf("Parent Process\n");
                wait(NULL);
                printf("Parent ID: %d\n",getpid());
        }
        return 0;
}
```

OUTPUT:



```
Parent Process
Child Process
Child ID: 463
Print ID: 459
Parent ID: 459


...Program finished with exit code 0
```

2) Implementation of directory management using the following system calls of the UNIX operating system: **opendir, readdir, closedir.**

AIM:

To implement the directory management using unix system calls of open,close and read directories.

ALGORITHM:

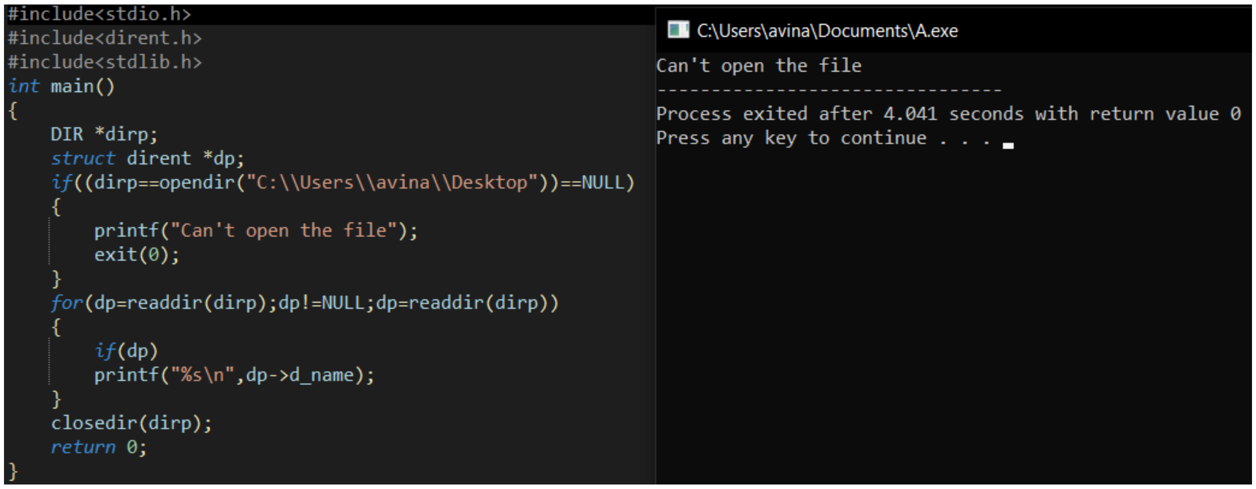
<u>STEP1:</u> Declare pointers dirp and dp.

<u>STEP2:</u> Check if drip is NULL and print can’t open the file.

STEP3: Set a for loop until dp is not NULL.
STEP4: Check dp and print the name pointed by the pointer dp.
STEP5: Close directory dirp.
PROGRAM:

```
#include<stdio.h>
#include<dirent.h>
#include<stdlib.h>
int main()
{
        DIR *dirp;
        struct dirent *dp;
        if((dirp=opendir("C:\\Users\\avina\\Desktop"))==NULL)
        {
                printf("Can't open the file");
                exit(0);
        }
        for(dp=readdir(dirp);dp!=NULL;dp=readdir(dirp))
        {
                if(dp)
                printf("%s\n",dp->d_name);
        }
        closedir(dirp);
        return 0;
}
```

OUTPUT:



```
#include<stdio.h>
#include<dirent.h>
#include<stdlib.h>
int main()
{
    DIR *dirp;
    struct dirent *dp;
    if((dirp==opendir("C:\\Users\\avina\\Desktop"))==NULL)
    {
        printf("Can't open the file");
        exit(0);
    }
    for(dp=readdir(dirp);dp!=NULL;dp=readdir(dirp))
    {
        if(dp)
        printf("%s\n",dp->d_name);
    }
    closedir(dirp);
    return 0;
}
```

```
C:\Users\avina\Documents\A.exe
Can't open the file
--------------------------------
Process exited after 4.041 seconds with return value 0
Press any key to continue . . .
```

3) Write a program using the I/O system calls of UNIX operating system: **open, read, write, close.**

AIM:

To implement the program using input and output system calls of unix operating system.

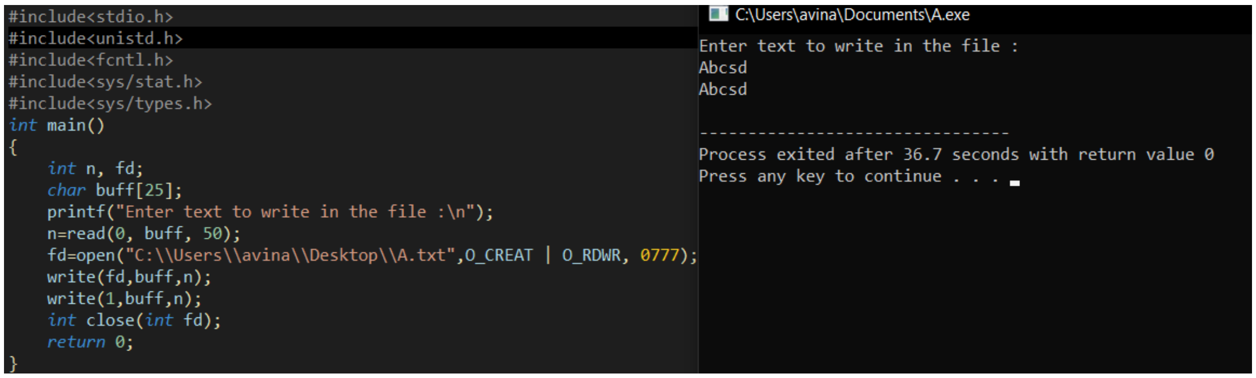
ALGORITHM:

STEP1: Declare variables n, fd and an array buff with size 25.
STEP2: Get text from the user and assign it to variable n.
STEP3: Open the text file on the variable fd.
STEP4: Write the text got from the user at the file opened at variable fd.
STEP5: Close the file opened at fd.

PROGRAM:

```
#include<stdio.h>
#include<unistd.h>
#include<fcntl.h>
#include<sys/stat.h>
#include<sys/types.h>
int main()
{
        int n, fd;
        char buff[25];
        printf("Enter text to write in the file :\n");
        n=read(0, buff, 50);
        fd=open("C:\\Users\\avina\\Desktop\\A.txt",O_CREAT | O_RDWR, 0777);
        write(fd,buff,n);
        write(1,buff,n);
        int close(int fd);
        return 0;
}
```

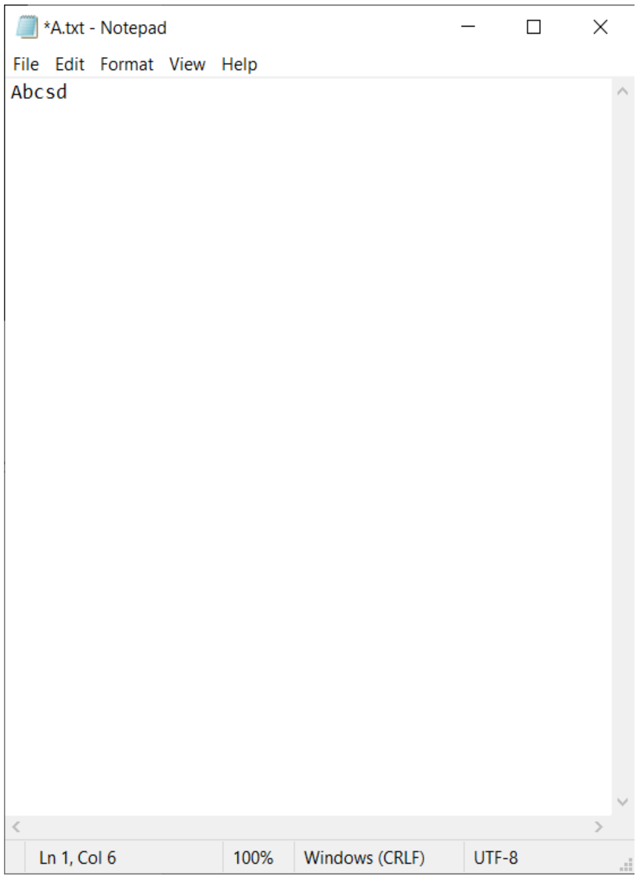
OUTPUT:



```
#include<stdio.h>
#include<unistd.h>
#include<fcntl.h>
#include<sys/stat.h>
#include<sys/types.h>
int main()
{
    int n, fd;
    char buff[25];
    printf("Enter text to write in the file :\n");
    n=read(0, buff, 50);
    fd=open("C:\\Users\\avina\\Desktop\\A.txt",O_CREAT | O_RDWR, 0777);
    write(fd,buff,n);
    write(1,buff,n);
    int close(int fd);
    return 0;
}
```

C:\Users\avina\Documents\A.exe

```
Enter text to write in the file :
Abcsd
Abcsd

--------------------------------
Process exited after 36.7 seconds with return value 0
Press any key to continue . . .
```



```
Abcsd
```

RESULT:

The above program has been executed and output has been verified.

EX NO : 04
DATE : 22/09/2021

SIMULATION AND ANALYSIS OF NON PRE-EMPTIVE AND PRE-EMPTIVE CPU SCHEDULING ALGORITHMS

OBJECTIVE :

To simulate and analyze the operation of CPU scheduling using FCFS,SJF, Priority and Round Robin algorithms.

DESCRIPTION :

When a computer is multi-programmed, it frequently has multiple processes competing for the CPU at the same time frequently. This situation occurs whenever two or more processes are simultaneously in the ready state. If only one CPU is available, a choice has to be made which process has to be in the CPU. The part of the operating system that makes the choice is called the scheduler and the algorithm is called scheduling algorithm.

In the FCFS algorithm the process which arrives first is given the c CPU after finishing its request; only it will allow the CPU to execute another process. In the SJF algorithm the process which has less service time given the CPU after finishing its request only will allow the CPU to execute the next other process. In priority scheduling, processes are allocated to the CPU on the basis of an externally assigned priority. The key to the performance of priority scheduling is in choosing priorities for the processes. In the Round Robin algorithm we are assigning some time slice .The process is allocated according to the time slice, if the process service time is less than the time slice then the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue .If the CPU burst of the currently running process is longer than time quantum; the timer will go off and will cause an interrupt to the operating system. A context switch will be executed and the process will be put at the tail of the ready queue.

ALGORITHM :

PROGRAM :

a) NON PRE-EMPTIVE FIRST COME FIRST SERVE

```
#include<stdio.h>
void findWaitingTime(int processes[], int n, int bt[], int wt[])
{
   wt[0] = 0;
   for (int  i = 1; i < n ; i++ )
       wt[i] =  bt[i-1] + wt[i-1];
}

void findTurnAroundTime( int processes[], int n, int bt[], int wt[], int tat[])
{
   for (int  i = 0; i < n ; i++)
       tat[i] = bt[i] + wt[i];
}

void findavgTime( int processes[], int n, int bt[])
{
   int wt[n], tat[n], total_wt = 0, total_tat = 0;
   findWaitingTime(processes, n, bt, wt);
   findTurnAroundTime(processes, n, bt, wt, tat);

   printf("Processes   Burst time   Waiting time   Turn around time\n");
   for (int  i=0; i<n; i++)
   {
       total_wt = total_wt + wt[i];
       total_tat = total_tat + tat[i];
       printf("   %d ",(i+1));
       printf("           %d ", bt[i] );
       printf("           %d",wt[i] );
       printf("              %d\n",tat[i] );
   }

   float s=(float)total_wt / (float)n;
   float t=(float)total_tat / (float)n;

   printf("Average waiting time = %f",s);
   printf("\n");
   printf("Average turn around time = %f ",t);
}

int main()
{
   int processes[] = { 1, 2, 3, 4};
   int n = sizeof processes / sizeof processes[0];
   int  burst_time[] = {21, 3, 6, 2};
   findavgTime(processes, n,  burst_time);
   return 0;
}
```

b) NON PRE-EMPTIVE SHORTEST JOB FIRST

```
#include<stdio.h>
 int main()
{
   int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;
   float avg_wt,avg_tat;
   printf("Enter number of process:");
   scanf("%d",&n);
   printf("\nEnter Burst Time:\n");
   for(i=0;i<n;i++)
   {
      printf("P%d:",i+1);
      scanf("%d",&bt[i]);
      p[i]=i+1;
   }

   for(i=0;i<n;i++)
   {
      pos=i;
      for(j=i+1;j<n;j++)
      {
         if(bt[j]<bt[pos])
            pos=j;
```

```
        }
        temp=bt[i];
        bt[i]=bt[pos];
        bt[pos]=temp;
        temp=p[i];
        p[i]=p[pos];
        p[pos]=temp;
    }
    wt[0]=0;
    for(i=1;i<n;i++)
    {
        wt[i]=0;
        for(j=0;j<i;j++)
            wt[i]+=bt[j];
        total+=wt[i];
    }
    avg_wt=(float)total/n;
    total=0;
    printf("\nProcess\t    Burst Time    \tWaiting Time\tTurnaround Time");
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];
        total+=tat[i];
        printf("\np%d\t\t  %d\t\t    %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
    }
    avg_tat=(float)total/n;
    printf("\n\nAverage Waiting Time=%f",avg_wt);
    printf("\nAverage Turnaround Time=%f\n",avg_tat);
}
```

c) PRE-EMPTIVE SHORTEST JOB FIRST (SHORTEST REMAINING TIME FIRST)

```
#include <stdio.h>
int main()
{
 int a[10],b[10],x[10],i,j,smallest,count=0,time,n;
 double avg=0,tt=0,end;
  printf("Enter the number of Processes : ");
  scanf("%d",&n);
 printf("Enter the Arrival Time\n");
 for(i=0;i<n;i++)
 scanf("%d",&a[i]);
 printf("Enter the Burst Time\n");
 for(i=0;i<n;i++)
 scanf("%d",&b[i]);
 for(i=0;i<n;i++)
 x[i]=b[i];

  b[9]=9999;
 for(time=0;count!=n;time++)
 {
   smallest=9;
  for(i=0;i<n;i++)
  {
   if(a[i]<=time && b[i]<b[smallest] && b[i]>0 )
   smallest=i;
  }
  b[smallest]--;
  if(b[smallest]==0)
  {
   count++;
   end=time+1;
   avg=avg+end-a[smallest]-x[smallest];
   tt= tt+end-a[smallest];
  }
 }
 printf("\n\nAverage waiting time = %lf\n",avg/n);
    printf("Average Turnaround time = %lf",tt/n);
    return 0;
}
```

d) NON PRE-EMPTIVE PRIORITY SCHEDULING

```
#include<stdio.h>

int main()

{

   int bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg_wt,avg_tat;

   printf("Enter Total Number of Process:");

   scanf("%d",&n);


   printf("\nEnter Burst Time and Priority\n");

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

   {

      printf("\nP[%d]\n",i+1);

      printf("Burst Time:");

      scanf("%d",&bt[i]);

      printf("Priority:");

      scanf("%d",&pr[i]);

      p[i]=i+1;

   }


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

   {

      pos=i;

      for(j=i+1;j<n;j++)

      {

         if(pr[j]<pr[pos])
```

```
            pos=j;
        }
        temp=pr[i];
        pr[i]=pr[pos];
        pr[pos]=temp;

        temp=bt[i];
        bt[i]=bt[pos];
        bt[pos]=temp;

        temp=p[i];
        p[i]=p[pos];
        p[pos]=temp;
    }

    wt[0]=0;
    for(i=1;i<n;i++)
    {
        wt[i]=0;
        for(j=0;j<i;j++)
            wt[i]+=bt[j];

        total+=wt[i];
    }
    avg_wt=total/n;
    total=0;
    printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time");
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];
        total+=tat[i];
        printf("\nP[%d]\t\t%d\t\t%d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
    }

    avg_tat=total/n;
    printf("\n\nAverage Waiting Time=%d",avg_wt);
    printf("\nAverage Turnaround Time=%d\n",avg_tat);

        return 0;
}
```

e) PRE-EMPTIVE PRIORITY SCHEDULING

```
#include<iostream>
using namespace std;
int main()
{
    int a[10],b[10],x[10];
```

```
 int waiting[10],turnaround[10],completion[10],p[10];
 int i,j,smallest,count=0,time,n;
 double avg=0,tt=0,end;

cout<<"\nEnter the number of Processes: ";
 cin>>n;
 for(i=0;i<n;i++)
 {
   cout<<"\nEnter arrival time of process: ";
   cin>>a[i];
 }
 for(i=0;i<n;i++)
 {
   cout<<"\nEnter burst time of process: ";
   cin>>b[i];
 }
 for(i=0;i<n;i++)
 {
   cout<<"\nEnter priority of process: ";
   cin>>p[i];
 }
 for(i=0; i<n; i++)
     x[i]=b[i];
 p[9]=-1;
 for(time=0; count!=n; time++)
 {
     smallest=9;
     for(i=0; i<n; i++)
     {
        if(a[i]<=time && p[i]>p[smallest] && b[i]>0 )
           smallest=i;
     }
     b[smallest]--;


     if(b[smallest]==0)
     {
        count++;
        end=time+1;
        completion[smallest] = end;
        waiting[smallest] = end - a[smallest] - x[smallest];
        turnaround[smallest] = end - a[smallest];
     }
 }
  cout<<"Process"<<"\t"<< "burst-time"<<"\t"<<"arrival-time" <<"\t"<<"waiting-time"
<<"\t"<<"turnaround-time"<< "\t"<<"completion-time"<<"\t"<<"Priority"<<endl;
 for(i=0; i<n; i++)
 {
```

```
cout<<"p"<<i+1<<"\t\t"<<x[i]<<"\t\t"<<a[i]<<"\t\t"<<waiting[i]<<"\t\t"<<turnaround[i]<<"\t\t"
<<completion[i]<<"\t\t"<<p[i]<<endl;
        avg = avg + waiting[i];
        tt = tt + turnaround[i];
    }
   cout<<"\n\nAverage waiting time ="<<avg/n;
    cout<<"  Average Turnaround time ="<<tt/n<<endl;
}
```

f) PRE-EMPTIVE ROUND ROBIN SCHEDULING

```
#include<stdio.h>


int main()
{
      int i, limit, total = 0, x, counter = 0, time_quantum;
      int wait_time = 0, turnaround_time = 0, arrival_time[10], burst_time[10], temp[10];
      float average_wait_time, average_turnaround_time;
      printf("\nEnter Total Number of Processes:\t");
      scanf("%d", &limit);
      x = limit;
      for(i = 0; i < limit; i++)
      {
            printf("\nEnter Details of Process[%d]\n", i + 1);
```

```
        printf("Arrival Time:\t");

        scanf("%d", &arrival_time[i]);

        printf("Burst Time:\t");

        scanf("%d", &burst_time[i]);

        temp[i] = burst_time[i];
    }

    printf("\nEnter Time Quantum:\t");
    scanf("%d", &time_quantum);
    printf("\nProcess ID\t\tBurst Time\t Turnaround Time\t Waiting Time\n");
    for(total = 0, i = 0; x != 0;)
    {
        if(temp[i] <= time_quantum && temp[i] > 0)
        {
            total = total + temp[i];
            temp[i] = 0;
            counter = 1;
        }
        else if(temp[i] > 0)
        {
                temp[i] = temp[i] - time_quantum;
                total = total + time_quantum;
            }
            if(temp[i] == 0 && counter == 1)
            {
                x--;
                printf("\nProcess[%d]\t\t%d\t\t %d\t\t\t %d", i + 1, burst_time[i], total -
arrival_time[i], total - arrival_time[i] - burst_time[i]);
                wait_time = wait_time + total - arrival_time[i] - burst_time[i];
                turnaround_time = turnaround_time + total - arrival_time[i];
                counter = 0;
            }
            if(i == limit - 1)
            {
                i = 0;
            }
            else if(arrival_time[i + 1] <= total)
            {
                i++;
            }
            else
            {
                i = 0;
            }
        }
```

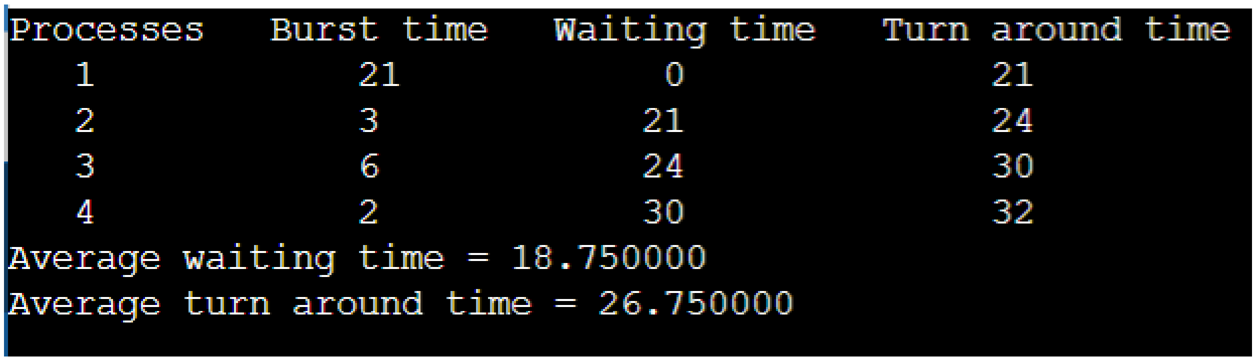
```
    average_wait_time = wait_time * 1.0 / limit;

    average_turnaround_time = turnaround_time * 1.0 / limit;

    printf("\n\nAverage Waiting Time:\t%f", average_wait_time);

    printf("\nAvg Turnaround Time:\t%f\n", average_turnaround_time);

    return 0;

}
```

SAMPLE INPUT :

PROCESS	PRIORITY	BURST TIME
P1	2	21
P2	1	3
P3	4	6
P4	3	2

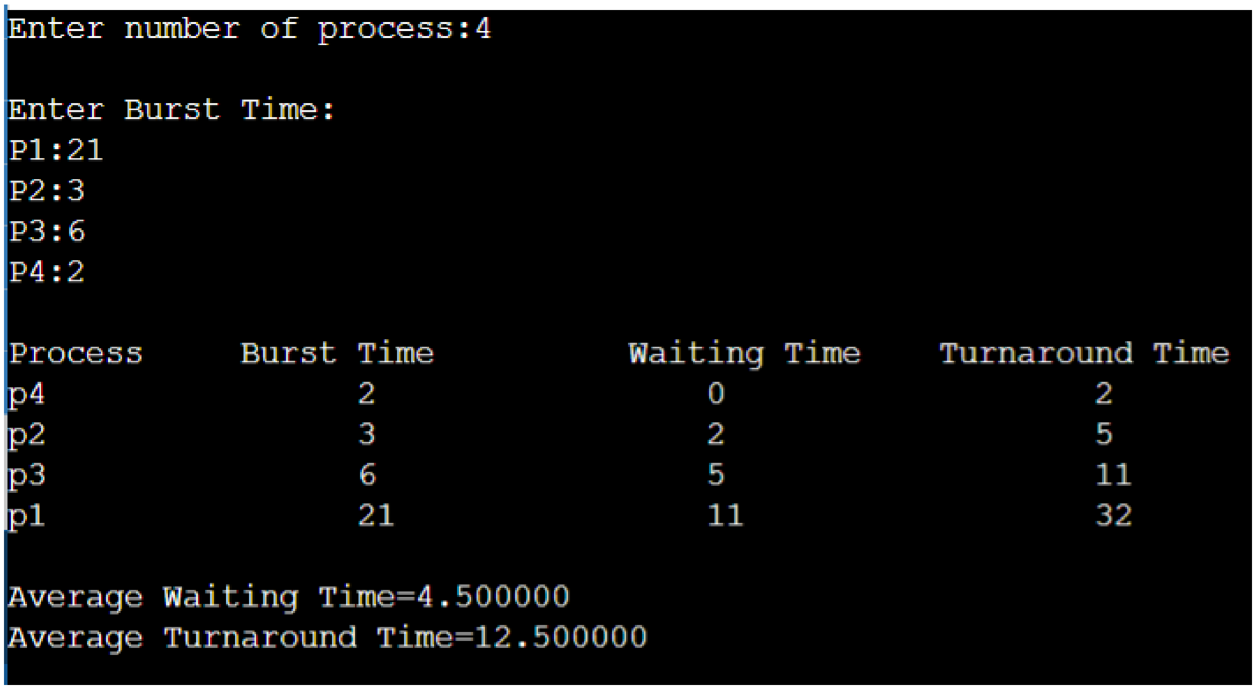
OUTPUTS :

a) NON PRE-EMPTIVE FIRST COME FIRST SERVE



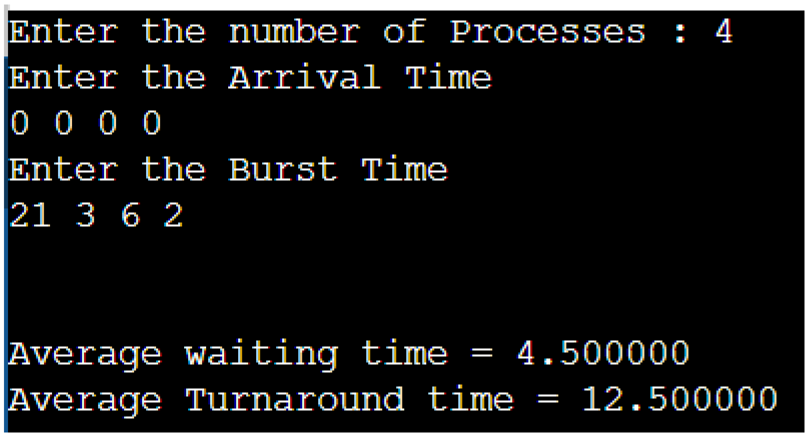
```
Processes   Burst time   Waiting time   Turn around time
   1           21            0              21
   2           3             21             24
   3           6             24             30
   4           2             30             32
Average waiting time = 18.750000
Average turn around time = 26.750000
```

b) NON PRE-EMPTIVE SHORTEST JOB FIRST



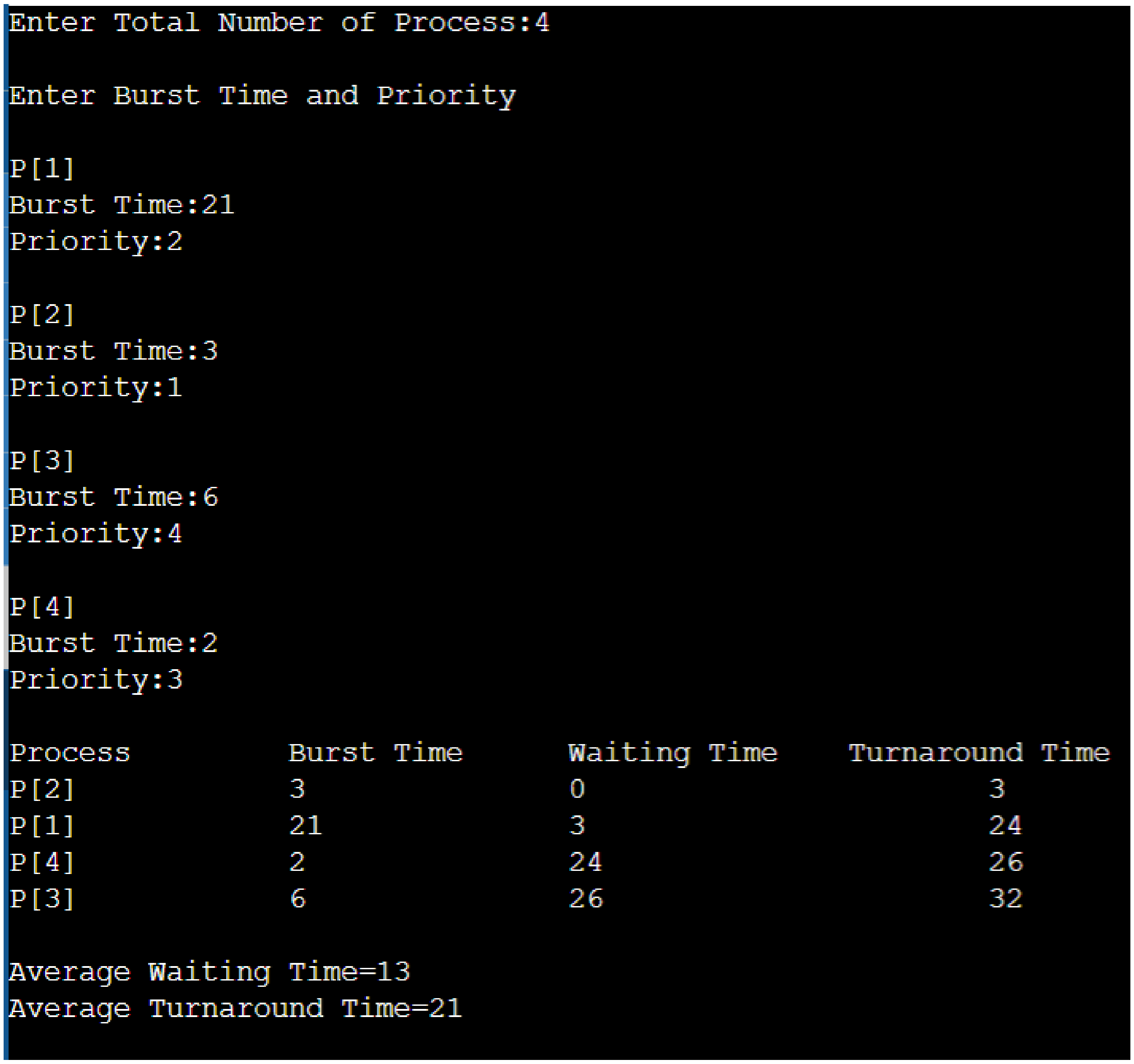
```
Enter number of process:4

Enter Burst Time:
P1:21
P2:3
P3:6
P4:2

Process      Burst Time           Waiting Time     Turnaround Time
p4                2                   0                   2
p2                3                   2                   5
p3                6                   5                   11
p1                21                  11                  32

Average Waiting Time=4.500000
Average Turnaround Time=12.500000
```

c) PRE-EMPTIVE SHORTEST JOB FIRST (SHORTEST REMAINING TIME FIRST)



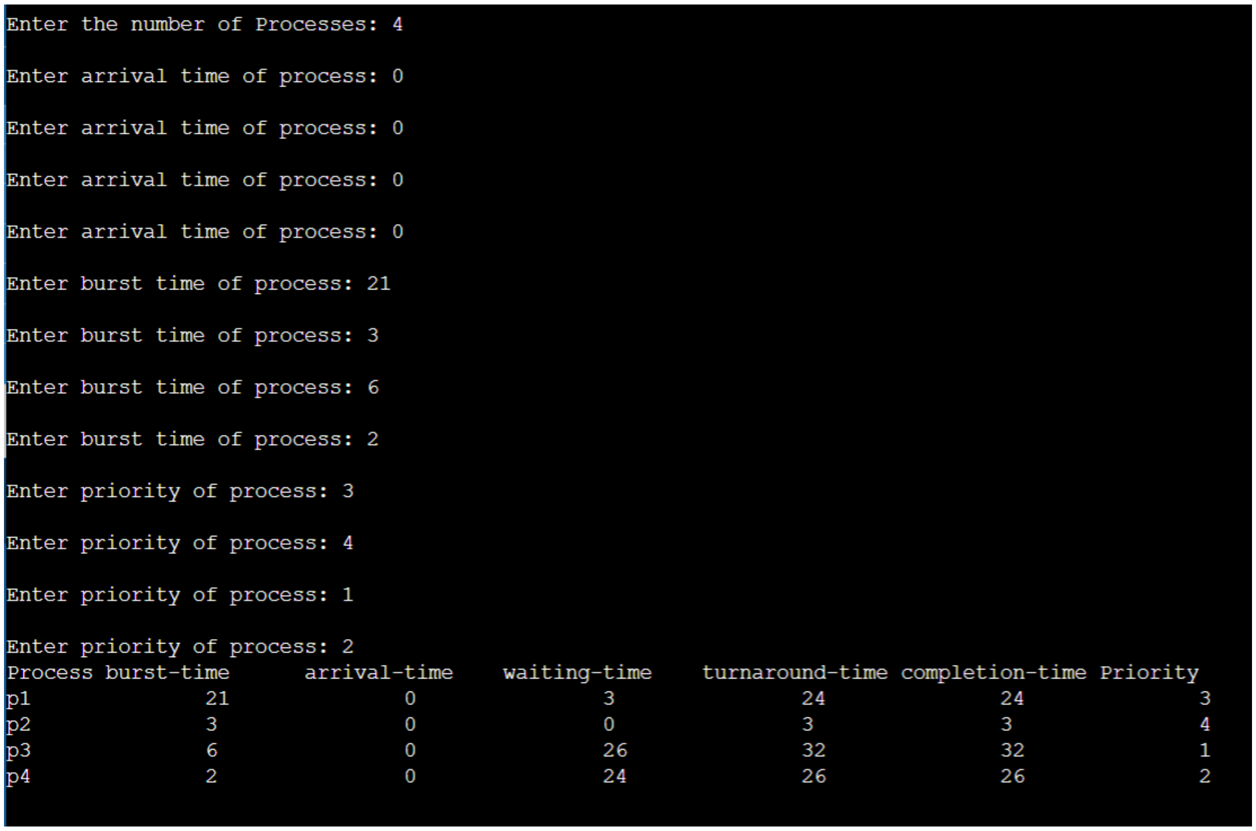
```
Enter the number of Processes : 4
Enter the Arrival Time
0 0 0 0
Enter the Burst Time
21 3 6 2


Average waiting time = 4.500000
Average Turnaround time = 12.500000
```

d) NON PRE-EMPTIVE PRIORITY SCHEDULING



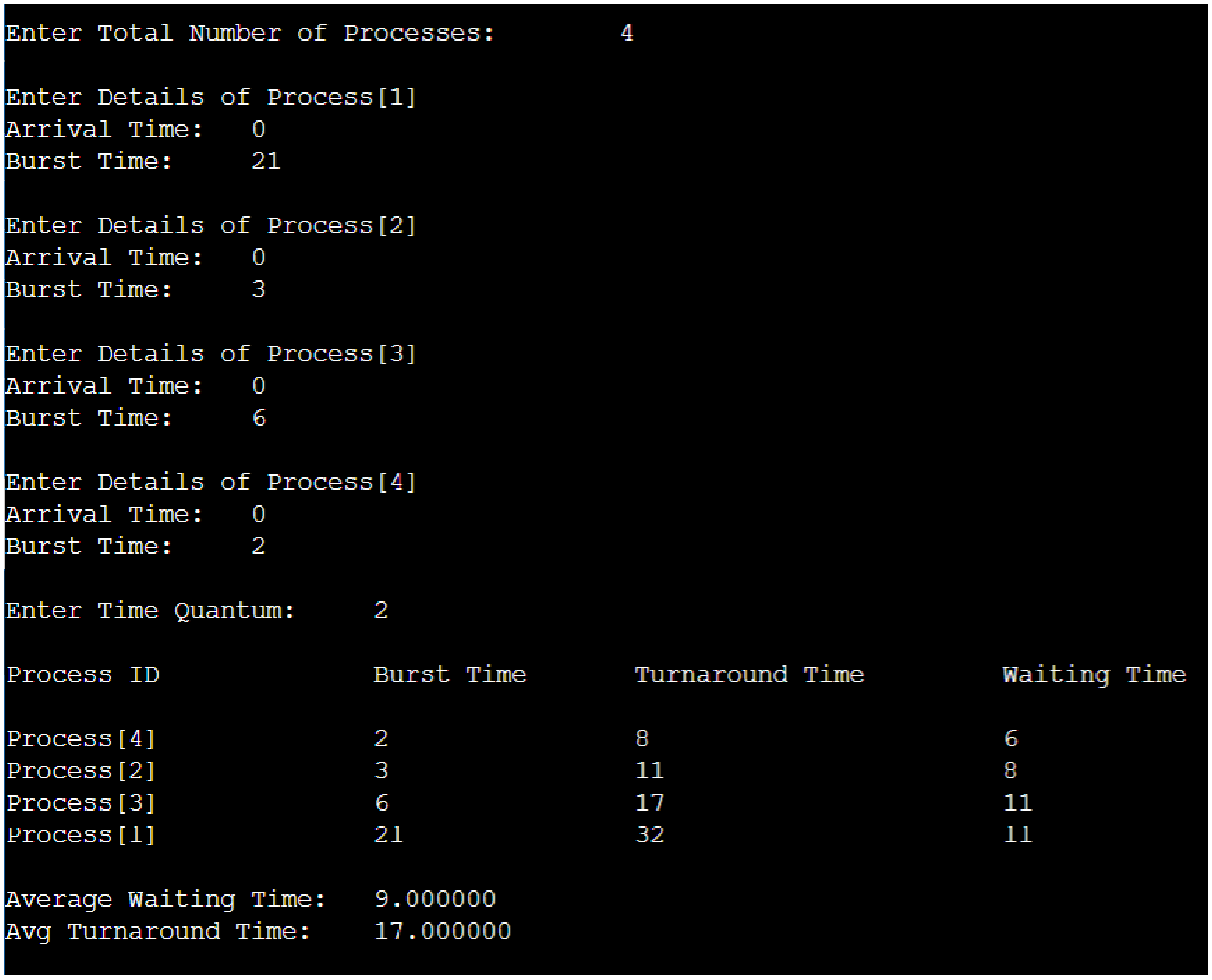
```
Enter Total Number of Process:4

Enter Burst Time and Priority

P[1]
Burst Time:21
Priority:2

P[2]
Burst Time:3
Priority:1

P[3]
Burst Time:6
Priority:4

P[4]
Burst Time:2
Priority:3

Process         Burst Time      Waiting Time    Turnaround Time
P[2]            3               0                       3
P[1]            21              3                       24
P[4]            2               24                      26
P[3]            6               26                      32

Average Waiting Time=13
Average Turnaround Time=21
```

e) PRE-EMPTIVE PRIORITY SCHEDULING



```
Enter the number of Processes: 4

Enter arrival time of process: 0

Enter arrival time of process: 0

Enter arrival time of process: 0

Enter arrival time of process: 0

Enter burst time of process: 21

Enter burst time of process: 3

Enter burst time of process: 6

Enter burst time of process: 2

Enter priority of process: 3

Enter priority of process: 4

Enter priority of process: 1

Enter priority of process: 2
Process burst-time      arrival-time    waiting-time    turnaround-time completion-time Priority
p1              21              0               3               24              24              3
p2              3               0               0               3               3               4
p3              6               0               26              32              32              1
p4              2               0               24              26              26              2
```

f) PRE-EMPTIVE ROUND ROBIN SCHEDULING



```
Enter Total Number of Processes:        4

Enter Details of Process[1]
Arrival Time:   0
Burst Time:     21

Enter Details of Process[2]
Arrival Time:   0
Burst Time:     3

Enter Details of Process[3]
Arrival Time:   0
Burst Time:     6

Enter Details of Process[4]
Arrival Time:   0
Burst Time:     2

Enter Time Quantum:     2

Process ID              Burst Time       Turnaround Time         Waiting Time

Process[4]              2                8                       6
Process[2]              3                11                      8
Process[3]              6                17                      11
Process[1]              21               32                      11

Average Waiting Time:   9.000000
Avg Turnaround Time:    17.000000
```

RESULT :

Thus, the operations of CPU scheduling using FCFS, SJF, Priority and Round Robin algorithms are simulated and analysed successfully.