OPERATING SYSTEMS LAB MANUAL

Prepared by:

PROF. PALLAVI. P. DIXIT
Assistant Professor
Department of CSE
JCER, Belagavi

OPERATING SYSTEMS LABORATORY

(Effective from the academic year 2023 -2024)

SEMESTER – III

Course Code: BCS303	**CIE Marks: 25**
Number of Contact Hours/Week 0:2:0	**SEE Marks: 25**
Total Number of Lab Contact Hours: 20	**Exam Hours: 03**
Credits: 04	

Course objectives:

- To Demonstrate the need for OS and different types of OS
- To discuss suitable techniques for management of different resources
- To demonstrate different APIs/Commands related to processor,memory, and storage and file system management.

Course Outcomes:

At the end of the course, the student will be able to:

CO1 – **Identify** the functionalities of OS and their categories.

CO2 – **Evaluate** multithread techniques and process scheduling algorithms.
CO3 – **Demonstrate** suitable techniques for resource management

CO4 – **Evaluate** file system allocation and memory management techniques architecture.
CO5 – **Review** the protection mechanisms in processing environment.

List of Experiments

CONTENT BEYOND SYLLABUS

SIMPLE SHELL PROGRAMS:

1. Develop a c program to implement the Process system calls (fork (), exec(), wait(), create process, terminate process)

i) fork()

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

 int main(int argc, char **argv)
 {
        pid_t pid;
        pid = fork();
        if(pid==0)
        {
                printf("It is the child process and pid is %d\n",getpid());
                exit(0);
         }

        else if(pid > 0)
        {
                printf("It is the parent process and pid is %d\n",getpid());
        }

        else
        {
                printf("Error while forking\n");
                exit(EXIT_FAILURE);
        }
 return 0;
 }
```

Output:

$make fork

cc fork.c -o fork

$ ./fork

And running the script, we get the result as below

It is the parent process and pid is 18097
It is the child process and pid is 18098

ii) exec()

In C programming on Linux and Ubuntu, consider the following example: We have two c files example.c and hello.c.

Code

example.c

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

int main( int argc, char *argv[] )
{
        printf("PID of example.c = %d\n", getpid());
        char *args[] = {"Hello", "C", "Programming", NULL};
        execv("./hello", args);
        printf("Back to example.c");
        return 0;
}
```

hello.c

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

int main( int argc, char *argv[] )
{
   printf("We are in Hello.c\n");
   printf("PID of hello.c = %d\n", getpid());
   return 0;
}
```

OUTPUT:

PID of example.c = 4733

We are in Hello.c

PID of hello.c = 4733

iii) wait()

```
#include<stdio.h> // printf()
#include<stdlib.h> // exit()
#include<sys/types.h> // pid_t
#include<sys/wait.h> // wait()
#include<unistd.h> // fork

int main(int argc, char **argv)
{

        pid_t pid;
        pid = fork();
        if(pid==0)
        {
                printf("It is the child process and pid is %d\n",getpid());
                int i=0;

                for(i=0;i<8;i++)
                {
                        printf("%d\n",i);
                }
        exit(0);
        }

        else if(pid > 0)
        {
                printf("It is the parent process and pid is %d\n",getpid());
                int status;
                wait(&status);
                printf("Child is reaped\n");
         }

        else
        {
                printf("Error in forking..\n");
                exit(EXIT_FAILURE);
        }

return 0;

}
```

OUTPUT:

$ make wait
cc wait.c -o wait
And running the script, we get the result as below screenshot.

$ ./wait

It is the parent process and pid is 18308

It is the child process and pid is 18309
0
1
2
3
4
5
6
7
child is reaped

iv) Create Process & Terminate Process:

CODE

```
#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 1666
```

2. Simulate the following CPU scheduling algorithms to find turnaround time and waiting time a) FCFS b) SJF c) Round Robin d) Priority.

a) FCFS:

ALGORITHM:

1. Enter all the processes and their burst time.
2. Find waiting time, **WT** of all the processes.
3. For the 1st process, **WT = 0**.
4. For all the next processes i, **WT[i] = BT[i-1] + WT[i-1]**.
5. Calculate Turnaround **time = WT + BT** for all the processes.
6. Calculate **average waiting time** = total waiting time/no. of processes.
7. Calculate **average turnaround time** = total turnaround time/no. of processes.

CODE:

```
#include <stdio.h>
int main()

{
    int pid[15];
    int bt[15];
    int n;

   printf("Enter the number of processes: ");
   scanf("%d",&n);
   printf("Enter process id of all the processes: ");

   for(int i=0;i<n;i++)
  {
        scanf("%d",&pid[i]);
   }
   printf("Enter burst time of all the processes: ");

   for(int i=0;i<n;i++)
   {
      scanf("%d",&bt[i]);
   }


   int i, wt[n];
   wt[0]=0;

  for(i=1; i<n; i++)                          //for calculating waiting time of each process
  {
      wt[i]= bt[i-1]+ wt[i-1];
   }
```

```
    printf("Process ID     Burst Time     Waiting Time     TurnAround Time\n");

    float twt=0.0;
    float tat= 0.0;

    for(i=0; i<n; i++)
    {
       printf("%d\t\t", pid[i]);
       printf("%d\t\t", bt[i]);
       printf("%d\t\t", wt[i]);
       printf("%d\t\t", bt[i]+wt[i]);     //calculating and printing turnaround time of each process
       printf("\n");

       twt += wt[i];                          //for calculating total waiting time
       tat += (wt[i]+bt[i]);                  //for calculating total turnaround time

    }
    float att,awt;
    awt = twt/n;                  //for calculating average waiting time
    att = tat/n;                  //for calculating average turnaround time

    printf("Avg. waiting time= %f\n",awt);
    printf("Avg. turnaround time= %f",att);
}
```

OUTPUT:

```
Enter the number of processes: 3
Enter process id of all the processes: 1 2 3
Enter burst time of all the processes: 5 11 11
Process ID     Burst Time     Waiting Time     TurnAround Time
1                    5                0                5
2                   11                5                16
3                   11                16               27
Avg. waiting time= 7.000000
Avg. turnaround time= 16.000000
```

b) SJF

ALGORITHM:

1. Enter number of processes.
2. Enter the **burst time** of all the processes.
3. Sort all the processes according to their **burst time**.
4. Find waiting time, **WT** of all the processes.
5. For the smallest process, **WT = 0**.
6. For all the next processes **i**, find waiting time by adding burst time of all the previously completed process.
7. Calculate **Turnaround time = WT + BT** for all the processes.
8. Calculate **average waiting time = total waiting time / no. of processes**.
9. Calculate **average turnaround time= total turnaround time / no. of processes**.

CODE:

```
#include<stdio.h>

int main()
{
   int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,totalT=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++)                          //sorting of burst times
   {
         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++)                    //finding the waiting time of all the processes
    {
         wt[i]=0;

         for(j=0;j<i;j++)
        {
          wt[i]+=bt[j];      //individual WT by adding BT of all previous completed processes
          total+=wt[i];
        }
    }


     avg_wt=(float)total/n;                    //average waiting time
     printf("\nProcess\t Burst Time \tWaiting Time\tTurnaround Time");

     for(i=0;i<n;i++)
     {
          tat[i]=bt[i]+wt[i];          //turnaround time of individual processes
          totalT+=tat[i];             //total turnaround time

          printf("\np%d\t\t %d\t\t %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
      }

       avg_tat=(float)totalT/n;              //average turnaround time

       printf("\n\nAverage Waiting Time=%f",avg_wt);
       printf("\nAverage Turnaround Time=%f",avg_tat);
}
```

OUTPUT:

Enter number of process:4

Enter Burst Time:
p1:5
p2:4
p3:12

p4:7

Process	Burst Time	Waiting Time	Turnaround Time
p2	4	0	4
p1	5	4	9
p4	7	9	16
p3	12	16	28

Average Waiting Time=7.250000
Average Turnaround Time=14.250000

c) ROUND ROBIN

ALGORITHM:

Step 1: Organize all processes according to their arrival time in the ready queue. The queue structure of the ready queue is based on the FIFO structure to execute all CPU processes.

Step 2: Now, we push the first process from the ready queue to execute its task for a fixed time, allocated by each process that arrives in the queue.

Step 3: If the process cannot complete their task within defined time interval or slots because it is stopped by another process that pushes from the ready queue to execute their task due to arrival time of the next process is reached. Therefore, CPU saved the previous state of the process, which helps to resume from the point where it is interrupted. (If the burst time of the process is left, push the process end of the ready queue).

Step 4: Similarly, the scheduler selects another process from the ready queue to execute its tasks. When a process finishes its task within time slots, the process will not go for further execution because the process's burst time is finished.

Step 5: Similarly, we repeat all the steps to execute the process until the work has finished.

CODE:

```
#include<stdio.h>
#include<conio.h>
void main()

{
   int i, NOP, sum=0,count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];
   float avg_wt, avg_tat;

   printf(" Total number of process in the system: ");
   scanf("%d", &NOP);
   y = NOP;                        // Assign the number of process to variable y

  // Use for loop to enter the details of the process like Arrival time and the Burst Time

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

  {
        printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1);
        printf(" Arrival time is: \t");            // Accept arrival time
        scanf("%d", &at[i]);
        printf(" \nBurst time is: \t");            // Accept the Burst time
        scanf("%d", &bt[i]);

        temp[i] = bt[i];                                    // store the burst time in temp array
   }

   printf("Enter the Time Quantum for the process: \t");      // Accept the Time quantum
   scanf("%d", &quant);

  // Display the process No, burst time, Turn Around Time and the waiting time
   printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");

   for (sum=0, i = 0; y!=0; )
  {
        if(temp[i] <= quant && temp[i] > 0)          // define the conditions
        {
              sum = sum + temp[i];
              temp[i] = 0;
             count=1;
        }
        else if(temp[i] > 0)
        {
              temp[i] = temp[i] - quant;
              sum = sum + quant;
        }

    if(temp[i]==0 && count==1)
```

```
    {
       y--;                                                    //decrement the process no.
       printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);
       wt = wt+sum-at[i]-bt[i];
       tat = tat+sum-at[i];
       count =0;
    }

     if(i==NOP-1)
    {
       i=0;
     }
    else if(at[i+1]<=sum)
    {
       i++;
    }
    else
    {
       i=0;
    }
  }

   avg_wt = wt * 1.0/NOP;      // represents the average waiting time and Turn Around time
   avg_tat = tat * 1.0/NOP;    // represents the average Turn Around time

   printf("\n Average Turn Around Time: \t%f", avg_wt);
  printf("\n Average Waiting Time: \t%f", avg_tat);
  getch();

}
```

OUTPUT:

Total number of process in the system: 4

Enter the Arrival and Burst time of the Process[1]

Arrival time is: 0

Burst time is: 8

Enter the Arrival and Burst time of the

Process[2]Arrival time is: 1

Burst time is: 5

Enter the Arrival and Burst time of the

Process[3]Arrival time is: 2

Burst time is: 10

Enter the Arrival and Burst time of the

Process[4]Arrival time is: 3

Burst time is: 11

Enter the Time Quantum for the process: 6

Process No	Burst Time	TAT	Waiting Time
Process No[2]	5	10	5
Process No[1]	8	25	17
Process No[3]	10	27	17
Process No[4]	11	31	20

Average Turn Around Time:

14.75000

0Average Waiting Time: 23.25000

d) PRIORITY SCHEDULEING:

ALGORITHM:

The algorithms prioritize the processes that must be carried out and schedule them accordingly. A higher priority process will receive CPU priority first, and this will continue until all of the processes are finished. The algorithm that places the processes in the ready queue in the order determined by their priority and chooses which ones to go to the job queue for execution requires both the job queue and the ready queue. Due to the process' greater priority, the Priority Scheduling Algorithm is in charge of transferring it from the ready queue to the work queue.

CODE:

```
#include <stdio.h>

void swap(int *a,int *b)

{
   int temp=*a;
   *a=*b;
   *b=temp;
}

int main()
{
   int n;
   printf("Enter Number of Processes: ");
   scanf("%d",&n);

   int burst[n],priority[n],index[n];

   for(int i=0;i<n;i++)
   {
         printf("Enter Burst Time and Priority Value for Process %d: ",i+1);
         scanf("%d %d",&burst[i],&priority[i]);
         index[i]=i+1;
   }

   for(int i=0;i<n;i++)
   {
        int temp=priority[i],
        int m=i;
```

```
        for(int j=i;j<n;j++)
        {
            if(priority[j] > temp)
            {
                temp=priority[j];
                m=j;
            }
        }
        swap(&priority[i], &priority[m]);
        swap(&burst[i], &burst[m]);
        swap(&index[i],&index[m]);
    }

    int t=0;
    printf("Order of process Execution is\n");

    for(int i=0;i<n;i++)
    {
        printf("P%d is executed from %d to %d\n",index[i],t,t+burst[i]);
        t+=burst[i];
    }

    printf("\n");
    printf("Process Id\tBurst Time\tWait Time\n");

    int wait_time=0;
    int total_wait_time = 0;

    for(int i=0;i<n;i++)
    {
        printf("P%d\t\t%d\t\t%d\n",index[i],burst[i],wait_time);
        total_wait_time += wait_time;
        wait_time += burst[i];
    }

    float avg_wait_time = (float) total_wait_time / n;
    printf("Average waiting time is %f\n", avg_wait_time);

    int total_Turn_Around = 0;

    for(int i=0; i < n; i++)
    {
        total_Turn_Around += burst[i];
    }

    float avg_Turn_Around = (float) total_Turn_Around / n;
    printf("Average TurnAround Time is %f",avg_Turn_Around);
    return 0;
}
```

OUTPUT:

```
Enter Number of Processes: 2
Enter Burst Time and Priority Value for Process 1: 5 3
Enter Burst Time and Priority Value for Process 2: 4 2
Order of process Execution is
P1 is executed from 0 to 5
P2 is executed from 5 to 9
Process Id      Burst Time      Wait Time
P1              5               0
P2              4               5
Average waiting time is 2.500000
Average TurnAround Time is 4.50
```

3. Develop a C program to simulate producer-consumer problem using semaphores

ALGORITHM:

1. Initialize the empty semaphore to the size of the buffer and the full semaphore to 0.
2. The producer acquires the empty semaphore to check if there are any empty slots in the buffer. If there are no empty slots, the producer blocks until a slot becomes available.
3. The producer acquires the mutex to access the buffer, inserts a data item into an empty slot in the buffer, and releases the mutex.
4. The producer releases the full semaphore to indicate that a slot in the buffer is now full.
5. The consumer acquires the full semaphore to check if there are any full slots in the buffer. If there are no full slots, the consumer blocks until a slot becomes available.
6. The consumer acquires the mutex to access the buffer, reads a data item from a full slot in the buffer, and releases the mutex.
7. The consumer releases the empty semaphore to indicate that a slot in the buffer is now empty.

CODE:

```
#include<stdio.h>
void main()
{
 int buffer[10], bufsize, in, out, produce, consume, choice=0;
 in = 0;
 out = 0;
 bufsize = 10;
 while(choice != 3)
 {
 printf("\n1. Produce\t2. Consume\t3. Exit");
 printf("\n Enter your choice: ");
 scanf("%d", &choice);
 switch(choice)
 {
 case 1:
 if((in+1)%bufsize == out)
```

```
printf("\nBuffer is Full");
else
{
printf("\nEnter the value: ");
scanf("%d", &produce);
buffer[in] = produce;
in = (in+1)%bufsize;
}
break;
case 2:
if(in == out)
printf("\nBuffer is Empty");
else
{
consume = buffer[out];
printf("\nThe consumed value is %d", consume);
out = (out+1)%bufsize;
}
break;
}
}
```

OUTPUT:

```
1. Produce 2. Consume 3. Exit
Enter your choice: 2
Buffer is Empty
1. Produce 2. Consume 3. Exit
Enter your choice: 1
Enter the value: 100
1. Produce 2. Consume 3. Exit
Enter your choice: 2
The consumed value is 100
1. Produce 2. Consume 3. Exit
Enter your choice: 3
```

2nd one:
1. Produce 2. Consume 3. Exit
Enter your choice: 1
Enter the value: 100
1. Produce 2. Consume 3. Exit
Enter your choice: 1
Enter the value: 300

Enter your choice: 2
The consumed value is 100
1. Produce 2. Consume 3. Exit
Enter your choice: 2
The consumed value is 300

Enter your choice: 3

4. Develop a C program which demonstrates interprocess communication between a reader process and a writer process. Use mkfifo, open, read, write and close APIs in your program.

ALGORITHM:

- It is an extension to the traditional pipe concept on Unix. A traditional pipe is "unnamed" and lasts only as long as the process.
- A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used.
- Usually a named pipe appears as a file and generally processes attach to it for inter-process communication. A FIFO file is a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.
- A FIFO special file is entered into the filesystem by calling *mkfifo()* in C. Once we have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.

CODE:

/*writer process */

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    int fd;
    char buf[1024];
   char * myfifo = "/tmp/myfifo";

    mkfifo(myfifo, 0666);
    printf("Run Reader process to read the FIFO File\n");
    fd = open(myfifo, O_WRONLY);
    write(fd,"Hi", sizeof("Hi"));                /* write "Hi" to the FIFO */
```

```
    close(fd);
    unlink(myfifo); /* remove the FIFO */
    return 0;
}
```

OUTPUT:

```
Run Reader process to read the FIFO File
```

/* Reader Process */

```
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#define MAX_BUF 1024

int main()
{
    int fd;
   /* A temp FIFO file is not created in reader */
   char *myfifo = "/tmp/myfifo";
   char buf[MAX_BUF];
   /* open, read, and display the message from the FIFO */
   fd = open(myfifo, O_RDONLY);
   read(fd, buf, MAX_BUF);
   printf("Writer: %s\n", buf);
   close(fd);
   return 0;
}
```

OUTPUT:

Writer: Hi

5. Develop a C program to simulate Bankers Algorithm for Dead Lock Avoidance.

Algorithm

1) Initialize:

Set count = 0, terminate = 0

Create an array done[] of size p and initialize all elements to 0

Create an array safe[] to store the safe sequence

2) Input Allocation and Maximum matrices:

Input allocation of resources for all processes (alc[][])

Input maximum resource requirement for all processes (max[][])

Input available resources (available[])

3) Calculate Need matrix:

For each process i from 0 to p-1:

For each resource j from 0 to c-1:

Calculate need[i][j] = max[i][j] - alc[i][j]

4) Safety Check:

Repeat until all processes are executed (count < p):

For each process i from 0 to p-1:

If process i is not executed (done[i] == 0):

For each resource j from 0 to c-1:

If need[i][j] > available[j], break the loop

If loop completes without breaking (j == c):

Add process i to the safe sequence (safe[count])

Mark process i as executed (done[i] = 1)

Release allocated resources of process i and add them to available

Increment count

Reset terminate = 0

Else:

Increment terminate

If terminate equals p - 1:

Print "Safe sequence does not exist"

Exit the loop

5) Output Results:

If terminate is not p - 1:

Print "Available resources after completion:"

For each resource j from 0 to c-1:

Print available[j]

Print "Safe sequence:"

For each process i from 0 to p-1:

Print "p" + safe[i]

CODE:

```
#include<stdio.h>
int main()
{
 int p, c, count = 0, i, j, alc[5][3], max[5][3], need[5][3], safe[5], available[3], done[5], terminate = 0;
 printf("Enter the number of process and resources \n");
 scanf("%d %d", & p, & c);
 // p is process and c is different resources
 printf("enter allocation of resource of all process %dx%d matrix \n", p, c);
 for (i = 0; i < p; i++) {
 for (j = 0; j < c; j++) {
 scanf("%d", & alc[i][j]);
 }
 }
 printf("enter the max resource process required %dx%d matrix \n", p, c);
 for (i = 0; i < p; i++) {
 for (j = 0; j < c; j++) {
 scanf("%d", & max[i][j]);
 }
```

```
}
printf("enter the available resource \n");
for (i = 0; i < c; i++)
scanf("%d", & available[i]);
printf("\n need resources matrix are\n");
for (i = 0; i < p; i++) {
for (j = 0; j < c; j++) {
need[i][j] = max[i][j] - alc[i][j];
printf("%d\t", need[i][j]);
}
printf("\n");
}
/* once process execute variable done will stop them for again execution */
for (i = 0; i < p; i++) {
done[i] = 0;
}
while (count < p) {
for (i = 0; i < p; i++) {
if (done[i] == 0) {
for (j = 0; j < c; j++) {
if (need[i][j] > available[j])
break;
}
//when need matrix is not greater then available matrix then if j==c will true
if (j == c) {
safe[count] = i;
done[i] = 1;
/* now process get execute release the resources and add them in available resources */
for (j = 0; j < c; j++) {
available[j] += alc[i][j];
```

```
}
count++;
terminate = 0;
} else {
terminate++;
}
}
}
if (terminate == (p - 1)) {
printf("safe sequence does not exist");
break;
}
}
if (terminate != (p - 1)) {
printf("\n available resource after completion\n");
for (i = 0; i < c; i++) {
printf("%d\t", available[i]);
}
printf("\n safe sequence are\n");
for (i = 0; i < p; i++) {
printf("p%d\t", safe[i]);
}
}
return 0;
}
```

Output:

```
Enter the number of process and resources
5 3
enter allocation of resource of all process 5x3 matrix
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
enter the max resource process required 5x3 matrix
7 5 3
3 2 2
9 0 2
4 2 2
5 3 3
enter the available resource
3 3 2
need resources matrix are
7 4 3
1 2 2
6 0 0
2 1 1
5 3 1
available resource after completion
10 5 7
safe sequence are
p1 p3 p4 p0 p2
Process returned 0 (0x0) execution time : 99.790 s
Press any key to continue
```

6. Develop a C program to simulate the following contiguous memory allocation Techniques:

a) Worst fit b) Best fit c) First fit.

a) WORST-FIT

ALGORITHM:

1. Get no. of Processes and no. of blocks.
2. After that get the size of each block and process requests.
3. Then select the best memory block that can be allocated using the Worst fit Technique.
4. Display the processes with the blocks that are allocated to a respective process.
5. Value of Fragmentation is optional to display to keep track of wasted memory.
6. Stop.

CODE:

```
#include<stdio.h>
#define max 25
int main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp,highest=0;
static int bf[max],ff[max];
printf("\n\tMemory Management Scheme - Worst Fit");
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
{
printf("Block %d:",i);
scanf("%d",&b[i]);
}
```

```
printf("Enter the size of the files :-\n");
for(i=1;i<=nf;i++)
{
printf("File %d:",i);
scanf("%d",&f[i]);
}
for(i=1;i<=nf;i++)
{
for(j=1;j<=nb;j++)
{
if(bf[j]!=1) //if bf[j] is not allocated
{
temp=b[j]-f[i];
if(temp>=0)
if(highest<temp)
{
ff[i]=j;
highest=temp;
}
}
}
frag[i]=highest;
bf[ff[i]]=1;
highest=0;
}
printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement");
for(i=1;i<=nf;i++)
printf("\n%d\t\t\t%d\t\t\t%d\t\t\t%d\t\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
return 0;
}
```

OUTPUT:

Memory Management Scheme - Worst Fit

Enter the number of blocks:3

Enter the number of files:2

Enter the size of the blocks:-

Block 1:5

Block 2:2

Block 3:7

Enter the size of the files :-

File 1:1

File 2:4

File_no: File_size : Block_no: Block_size: Fragement

1	1	3	7	6
2	4	1	5	1

Process returned 0 (0x0) execution time : 17.354 s

Press any key to continue.

B) BEST-FIT

ALGORITHM:

1. Get no. of Processes and no. of blocks.
2. After that get the size of each block and process requests.
3. Then select the best memory block that can be allocated using the Best fit Technique.
4. Display the processes with the blocks that are allocated to a respective process.
5. Value of Fragmentation is optional to display to keep track of wasted memory.
6. Stop.

CODE:

```
#include<stdio.h>
#define max 25
void main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp,lowest=10000;
static int bf[max],ff[max];
printf("\nMemory Management Scheme -- Best Fit");
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
{
printf("Block %d:",i);
scanf("%d",&b[i]);
}
printf("Enter the size of the files :-\n");
for(i=1;i<=nf;i++)
{
printf("File %d:",i);
```

```
scanf("%d",&f[i]);
}
for(i=1;i<=nf;i++)
{
for(j=1;j<=nb;j++)
{
if(bf[j]!=1)
{
temp=b[j]-f[i];
if(temp>=0)
if(lowest>temp)
{
ff[i]=j;
lowest=temp;
}
}
}
frag[i]=lowest;
bf[ff[i]]=1;
lowest=10000;
}
printf("\nFile No\t\t\tFile Size \t\tBlock No\t\tBlock Size\t\tFragment");
for(i=1;i<=nf && ff[i]!=0;i++)
printf("\n%d\t\t\t%d\t\t\t%d\t\t\t%d\t\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
return 0;
}
```

OUTPUT:

Memory Management Scheme -- Best Fit

Enter the number of blocks:3

Enter the number of files:2

Enter the size of the blocks:-

Block 1:5

Block 2:2

Block 3:7

Enter the size of the files :-

File 1:1

File 2:2

File No File Size Block No Block Size Fragment

1 1 2 2 1

2 2 1 5 3

Process returned 2 (0x2) execution time : 43.868 s

Press any key to continue.

C) FIRST-FIT

ALGORITHM:

1. Get no. of Processes and no. of blocks.
2. After that get the size of each block and process requests.
3. Then select the best memory block that can be allocated using the First fit Technique.
4. Display the processes with the blocks that are allocated to a respective process.
5. Value of Fragmentation is optional to display to keep track of wasted memory.
6. Stop.

CODE:

```
#include<stdio.h>
#define max 25
int main()
{
int frag[max],b[max],f[max],i,j,nb,nf,temp;
static int bf[max],ff[max];
printf("\nMemory Management Scheme -- First Fit");
printf("\nEnter the number of blocks:");
scanf("%d",&nb);
printf("Enter the number of files:");
scanf("%d",&nf);
printf("\nEnter the size of the blocks:-\n");
for(i=1;i<=nb;i++)
{
printf("Block %d:",i);
scanf("%d",&b[i]);
}
printf("Enter the size of the files :-\n");
for(i=1;i<=nf;i++)
{
```

```
printf("File %d:",i);
scanf("%d",&f[i]);
}
for(i=1;i<=nf;i++)
{
for(j=1;j<=nb;j++)
{
if(bf[j]!=1)
{
temp=b[j]-f[i];
if(temp>=0)
{
ff[i]=j;
break;
}
}
}
frag[i]=temp;
bf[ff[i]]=1;
}
printf("\n File_no:\t File_size :\t Block_no:\t Block_size:\t Fragement");
for(i=1;i<=nf;i++)
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
return 0;
}
```

OUTPUT:

Memory Management Scheme -- First Fit

Enter the number of blocks:3

Enter the number of files:2

Enter the size of the blocks:-

Block 1:5

Block 2:2

Block 3:7

Enter the size of the files :-

File 1:1

File 2:2

File_no: File_size : Block_no: Block_size: Fragement

1	1	1	5	4
2	2	2	2	0

Process returned 0 (0x0) execution time : 14.198 s

Press any key to continue.

7. Develop a C program to simulate page replacement algorithms: a) FIFO b) LRU

a. FIFO

ALGORITHM:

1. Start the process

2. Declare the size with respect to page length

3. Check the need of replacement from the page to memory

4. Check the need of replacement from old page to new page in memory

5. Form queue to hold all pages

6. Insert the page require memory into the queue

7. Check for bad replacement and page fault

8. Get the number of processes to be inserted

9. Display the values

10. Stop the process

CODE:

```
#include<stdio.h>
int main()
{
        int i,j,n,a[50],frame[10],no,k,avail,count=0;

        printf("\n ENTER THE NUMBER OF PAGES:\n");
        scanf("%d",&n);
        printf("\n ENTER THE PAGE NUMBER :\n");

        for(i=1;i<=n;i++)
        scanf("%d",&a[i]);
        printf("\n ENTER THE NUMBER OF FRAMES :");
        scanf("%d",&no);

        for(i=0;i<no;i++)
                frame[i]= -1;
                 j=0;
                 printf("\tref string\t page frames\n")

       for(i=1;i<=n;i++)
        {
printf("%d\t\t",a[i]);avail=0;
```

```
        for(k=0;k<no;k++)
            if(frame[k]==a[i])
                avail=1;
             if (avail==0)
                {
                    frame[j]=a[i];
                    j=(j+1)%no;
                    count++;
                    for(k=0;k<no;k++)
                    printf("%d\t",frame[k]);
        }
                        printf("\n");
        }
                printf("Page Fault Is %d",count);
                return 0;
}
```

OUTPUT:

ENTER THE NUMBER OF PAGES: 20
ENTER THE PAGE NUMBER : 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
ENTER THE NUMBER OF FRAMES : 3

ref string	page frames		
7	7	-1	-1
0	7	0	-1
1	7	0	1
2	2	0	1
0			
3	2	3	1
0	2	3	0
4	4	3	0
2	4	2	0
3	4	2	3
0	0	2	3
3			
2			
1	0	1	3
2	0	1	2
0			
1			
7	7	1	2
0	7	0	2
1	7	0	1

Page Fault is 15

b. LRU

ALGORITHM:

1. Initialize an empty cache of size N (number of frames) and an empty queue (used to keep track of the order of page references).
2. While processing page references:
   a. Read the next page reference.
   b. If the page is present in the cache (cache hit), move the corresponding entry to the front of the queue.
   c. If the page is not present in the cache (cache miss):
      i. If the cache is full (all frames are occupied):
         1. Remove the least recently used page (the page at the end of the queue).
         2. Add the new page to the cache and insert it at the front of the queue.
      ii. If the cache has an empty frame:
         1. Add the new page to the cache and insert it at the front of the queue.
   d. Repeat steps (a) to (c) until all page references are processed.
3. At the end of processing page references, the cache will contain the most frequently used pages.

CODE:

```
#include<stdio.h>
main()
{
        int q[20],p[50],c=0,c1,d,f,i,j,k=0,n,r,t,b[20],c2[20];
        printf("Enter no of pages:");
        scanf("%d",&n);
        printf("Enter the reference string:");
        for(i=0;i<n;i++)
        scanf("%d",&p[i]);
        printf("Enter no of frames:");
        scanf("%d",&f);
        q[k]=p[k];
        printf("\n\t%d\n",q[k]);
        c++;
        k++;
        for(i=1;i<n;i++)
        {
                c1=0;
                for(j=0;j<f;j++)
                {
                        if(p[i]!=q[j])
                        c1++;
                }
```

```
                    if(c1==f)
            {


                    c++;
                    if(k<f)
                    {

                    }
                    else
                    {


q[k]=p[i];k++;
for(j=0;j<k;j++) printf("\t%d",q[j]);printf("\n");



for(r=0;r<f;r++)
{

c2[r]=0;

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

{

if(q[r]!=p[j]) c2[r]++;

else break;

}

}

for(r=0;r<f;r++) b[r]=c2[r]; for(r=0;r<f;r++)

{

for(j=r;j<f;j++)

{

if(b[r]<b[j])

{

t=b[r]; b[r]=b[j]; b[j]=t;

}

}

}

for(r=0;r<f;r++)

{
```

```
if(c2[r]==b[0])
q[r]=p[i]; printf("\t%d",q[r]);
}
printf("\n");
}
}

}
printf("\n The no of page faults is %d",c);

}
```

OUTPUT:

```
Enter no of pages:10
Enter the reference string:7 5 9 4 3 7 9 6 2 1
Enter no of frames:3
7
7       5
7       5       9
4       5       9
4       3       9
4       3       7
9       3       7
9       6       7
9       6       2
1       6       2

The no of page faults is 10
```

8) Simulate following File Organization Techniques a) Single level directory b) Two level directory.

a. SINGLE LEVEL DIRECTORY

ALGORITHM:

Step-1: Start the program.

Step-2: Declare the count, file name, graphical interface.

Step-3: Read the number of files

Step-4: Read the file name

Step-5: Declare the root directory

Step-6: Using the file eclipse function define the files in a single level

Step-7: Display the files

Step-8: Stop the program

CODE:

```
#include<stdio.h>
struct
{
char dname[10],fname[10][10];
int fcnt;
}dir;

void main()
{
    int i,ch;
    char f[30];
    clrscr();
    dir.fcnt = 0;
    printf("\nEnter name of directory -- ");
    scanf("%s", dir.dname);
    while(1)
    {
        printf("\n\n 1. Create File\t2. Delete File\t3. Search File \n 4. Display Files\t5.      Exit\nEnter your choice
-- ");
        scanf("%d",&ch);
        switch(ch)
        {
                case 1: printf("\n Enter the name of the
                file -- ");scanf("%s",dir.fname[dir.fcnt]);
                dir.fcnt++;
                        break;
```

```
        case 2: printf("\n Enter the name of the file -- ");
                scanf("%s",f);

                for(i=0;i<dir.fcnt;i++)
                {
                   if(strcmp(f, dir.fname[i])==0)
                   {
                         printf("File %s is deleted ",f);
                         strcpy(dir.fname[i],dir.fname[dir.fcnt-1]);
                         break;
                    }
                  }
        if(i==dir.fcnt)
        printf("File %s not found",f);
        else
        dir.fcnt--;
        break;

case 3: printf("\n Enter the name of the file -- ");
        scanf("%s",f);
        for(i=0;i<dir.fcnt;i++)
        {
           if(strcmp(f, dir.fname[i])==0)
          {
             printf("File %s is found ", f);
             break;
           }
        }
        if(i==dir.fcnt)
        printf("File %s not found",f);
        break;

case 4: if(dir.fcnt==0)
        printf("\n Directory Empty");
        else
        {
           printf("\n The Files are -- ");
           for(i=0;i<dir.fcnt;i++)
           printf("\t%s",dir.fname[i]);
        }
        break;
        default: exit(0);
        }
}

        getch();
        }
```

OUTPUT:

Enter name of directory -- CSE

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 1

Enter the name of the file -- A

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 1

Enter the name of the file -- B

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 1

Enter the name of the file -- C

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 4

The Files are -- A B C

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 3

Enter the name of the file – ABC

File ABC not found
1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 2

Enter the name of the file – B
File B is deleted

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 5

b. TWO LEVEL DIRECTORY:

ALGORITHM:

Step-1: Start the program.

Step-2: Declare the count, file name, graphical interface.

Step-3: Read the number of files

Step-4: Read the file name

Step-5: Declare the root directory

Step-6: Using the file eclipse function define the files in a two level diretory.

Step-7: Display the files

Step-8: Stop the program

CODE:

```
#include<stdio.h>
struct
{
   char dname[10],fname[10][10];
   int fcnt;
   dir[10];
}

void main()
{
    int i,ch,dcnt,k;
    char f[30], d[30];
    clrscr();
    dcnt=0;
    while(1)
    {
        printf("\n\n 1. Create Directory\t 2. Create File\t 3. Delete File");
        printf("\n 4. Search File \t \t 5. Display \t 6. Exit \t Enter your choice -- ");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: printf("\n Enter name of directory -- ");
                    scanf("%s", dir[dcnt].dname);
                    dir[dcnt].fcnt=0;
                    dcnt++;
                    printf("Directory created");
                    break;
           case 2: printf("\n Enter name of the directory -- ");
                   scanf("%s",d);
```

```
            for(i=0;i<dcnt;i++)
            if(strcmp(d,dir[i].dname)==0)
            {
                printf("Enter name of the file -- ");
                scanf("%s",dir[i].fname[dir[i].fcnt]);
                dir[i].fcnt++;
                printf("File created");
                break;
            }

    if(i==dcnt)
    printf("Directory %s not found",d);
    break;

   case 3: printf("\nEnter name of the directory -- ");
            scanf("%s",d);
            for(i=0;i<dcnt;i++)
            {
              if(strcmp(d,dir[i].dname)==0)
              {
                  printf("Enter name of the file -- ");
                  scanf("%s",f);
                  for(k=0;k<dir[i].fcnt;k++)
                  {
                     if(strcmp(f, dir[i].fname[k])==0)
                    {
                          printf("File %s is deleted ",f);
                          dir[i].fcnt--;
                          strcpy(dir[i].fname[k],dir[i].fname[dir[i].fcnt]);
                          goto jmp;
                    }
                  }
printf("File %s not found",f);
goto jmp;
}
}
printf("Directory %s not found",d);
jmp :
break;

case 4: printf("\nEnter name of the directory -- ");
scanf("%s",d);
for(i=0;i<dcnt;i++)
{
   if(strcmp(d,dir[i].dname)==0)
  {
     printf("Enter the name of the file -- ");
     scanf("%s",f);
     for(k=0;k<dir[i].fcnt;k++)
     {
```

```
                if(strcmp(f, dir[i].fname[k])==0)
                {
                   printf("File %s is found ",f);
                   goto jmp1;
                 }
               }
           printf("File %s not found",f);
           goto jmp1;
         }
       }

           printf("Directory %s not found",d);
           jmp1: break;
           case 5: if(dcnt==0)
           printf("\nNo Directory's ");
           else
           {
              printf("\nDirectory\tFiles");
              for(i=0;i<dcnt;i++)
              {
                 printf("\n%s\t\t",dir[i].dname);
                 for(k=0;k<dir[i].fcnt;k++)
                 printf("\t%s",dir[i].fname[k]);
              }
            }
           break;

           default:exit(0);
           }
           }

           getch();
}
```

OUTPUT:

1. Create Directory 2. Create File 3. Delete File

4. Search File 5. Display 6. Exit Enter your choice -- 1

Enter name of directory -- DIR1

Directory created

1. Create Directory 2. Create File 3. Delete File

4. Search File 5. Display 6. Exit Enter your choice -- 1

Enter name of directory -- DIR2

Directory created

1. Create Directory 2. Create File 3. Delete File

4. Search File 5. Display 6. Exit Enter your choice -- 2

Enter name of the directory – DIR1

Enter name of the file -- A1

File created

1. Create Directory 2. Create File 3. Delete File

4. Search File 5. Display 6. Exit Enter your choice -- 2

Enter name of the directory – DIR1

Enter name of the file -- A2

File created

1. Create Directory 2. Create File 3. Delete File

4. Search File 5. Display 6. Exit Enter your choice -- 2

Enter name of the directory – DIR2

Enter name of the file -- B1

File created

1. Create Directory 2. Create File 3. Delete File

4. Search File 5. Display 6. Exit Enter your choice -- 5

Directory Files

DIR1 A1 A2

DIR2 B1

1. Create Directory 2. Create File 3. Delete File

4. Search File 5. Display 6. Exit Enter your choice -- 4

Enter name of the directory – DIR

Directory not found

1. Create Directory 2. Create File 3. Delete File

4. Search File 5. Display 6. Exit Enter your choice -- 3

Enter name of the directory – DIR1

Enter name of the file -- A2

File A2 is deleted

1. Create Directory 2. Create File 3. Delete File

4. Search File 5. Display 6. Exit Enter your choice – 6

9. Develop a C program to simulate the Linked file allocation strategies.

ALGORITHM:

Step 1: Create a queue to hold all pages in memory

Step 2: When the page is required replace the page at the head of the queue

Step 3: Now the new page is inserted at the tail of the queue

Step 4: Create a stack.

Step 5: When the page fault occurs replace page present at the bottom of the stack

Step 6: Stop the allocation.

CODE:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
        int f[50], p,i, st, len, j, c, k, a;
        clrscr();

        for (i=0;i<50;i++)
                f[i]=0;
                printf("Enter how many blocks already allocated: ");
                scanf("%d",&p);
                printf("Enter blocks already allocated: ");
        for(i=0;i<p;i++)
        {
                scanf("%d",&a);
                f[a]=1;
        }
        x: printf("Enter index starting block and length: ");
        scanf("%d%d", &st,&len);
        k=len;
        if(f[st]==0)
        {
                for(j=st;j<(st+k);j++)
                {
                        if(f[j]==0)
                        {


                        }

                        else
```

```
                    {
                    printf("%d Block is already allocated \n",j);
                    k++;
                    }
                }
        }
        else
                printf("%d starting block is already allocated \n",st);
                printf("Do you want to enter more file(Yes - 1/No - 0)");
                scanf("%d", &c);
        if(c==1)
                goto x;
                else
                exit(0);
                getch();
}
```

OUTPUT:

Enter how many blocks already allocated: 3

Enter blocks already allocated: 1 3 5

Enter index starting block and length: 2 2

2.....>1

3 Block is already allocated

4.....>1

Do you want to enter more file(Yes - 1/No - 0)0

10. Develop a C program to simulate SCAN disk scheduling algorithm

Algorithm:

1. Input max range of the disk, initial head position, size of the queue request, and queue of disk positions.
2. Initialize variables: seek = 0, temp1 = 0, temp2 = 0.
3. Split the queue into two separate queues based on head position:
 - If position >= head, add it to queue1, else add it to queue2.
4. Sort queue1 and queue2 in ascending order.
5. Combine queue1, head position, and queue2 into a single queue named queue.
6. Iterate through the combined queue:
 - Calculate the seek time by finding the absolute difference between consecutive disk positions.
 - Add the seek time to the total seek time.
 - Print the movement of the disk head between each position and the corresponding seek time.
7. Print the total seek time.
8. Calculate and print the average seek time.
9. End.

Code:

```
#include <stdio.h>
#include <math.h>
int main()
{
 int queue[20], n, head, i, j, k, seek = 0, max, diff, temp, queue1[20],
 queue2[20], temp1 = 0, temp2 = 0;
 float avg;
 printf("Enter the max range of disk\n");
 scanf("%d", &max);
 printf("Enter the initial head position\n");
 scanf("%d", &head);
 printf("Enter the size of queue request\n");
 scanf("%d", &n);
 printf("Enter the queue of disk positions to be read\n");
 for (i = 1; i <= n; i++)
 {
 scanf("%d", &temp);
 if (temp >= head)
 {
 queue1[temp1] = temp;
 temp1++;
 }
 else
 {
 queue2[temp2] = temp;
 temp2++;
 }
```

```
}
for (i = 0; i < temp1 - 1; i++)
{
for (j = i + 1; j < temp1; j++)
{
if (queue1[i] > queue1[j])
{
temp = queue1[i];
queue1[i] = queue1[j];
queue1[j] = temp;
}
}
}
for (i = 0; i < temp2 - 1; i++)
{
for (j = i + 1; j < temp2; j++)
{
if (queue2[i] > queue2[j])
{
temp = queue2[i];
queue2[i] = queue2[j];
queue2[j] = temp;
}
}
}
for (i = 1, j = 0; j < temp1; i++, j++)
queue[i] = queue1[j];
queue[i] = max;
queue[i + 1] = 0;
for (i = temp1 + 3, j = 0; j < temp2; i++, j++)
queue[i] = queue2[j];
queue[0] = head;
for (j = 0; j <= n + 1; j++)
{
diff = abs(queue[j + 1] - queue[j]);
seek += diff;
printf("Disk head moves from %d to %d with seek %d\n", queue[j],
queue[j + 1], diff);
}
printf("Total seek time is %d\n", seek);
avg = seek / (float)n;
printf("Average seek time is %f\n", avg);
return 0;
}
```

Output:

```
Enter the max range of disk
500
Enter the initial head position
50
Enter the size of queue request
10
Enter the queue of disk positions to be read
89
65
34
20
75
66
30
100
99
115
Disk head moves from 50 to 65 with seek 15
Disk head moves from 65 to 66 with seek 1
Disk head moves from 66 to 75 with seek 9
Disk head moves from 75 to 89 with seek 14
Disk head moves from 89 to 99 with seek 10
Disk head moves from 99 to 100 with seek 1
Disk head moves from 100 to 115 with seek 15
Disk head moves from 115 to 500 with seek 385
Disk head moves from 500 to 0 with seek 500
Disk head moves from 0 to 20 with seek 20
Disk head moves from 20 to 30 with seek 10
Disk head moves from 30 to 34 with seek 4
Total seek time is 984
Average seek time is 98.400002
Process returned 0 (0x0) execution time : 24.687 s
Press any key to continue
```

CONTENT BEYOND SYLLABUS

1. Write a Shell program to add the given two numbers

ALGORITHM:

- Read the numbers which will be given by user.
- Use $((a+b)) to add the numbers
- Use $var to store the value after adding the numbers and print the value.

CODE:

```
echo “Enter the first number”
read a
echo “Enter the second number”
read b
var= $((a+b))
echo $var
```

OUTPUT:

```
Enter the first number
50
Enter the second number
50
100
```

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

ALGORITHM:

- Read a number which will be given by user.
- Use $(($n % 2)). If it equal to 0 then it is even otherwise it is odd.
- Use if-else statements to make this program more easier.
- Must include 'fi' after written 'if-else' statements.

CODE:

```
# HOW TO FIND A NUMBER IS EVEN OR ODD IN SHELL SCRIPT
clear
echo "---- EVEN OR ODD IN SHELL SCRIPT     "
echo -n "Enter a number:"
read n
rem=$(( $n%2))
if [ $rem -eq 0 ]
then
echo "$n is even"
else
 echo "$n is Odd"
 fi
```

OUTPUT:

```
---- EVEN OR ODD IN SHELL SCRIPT   "
Enter a number:23
23 is odd
```

3. Write a Shell program to check the given year is leap year or not

ALGORITHM:

Step 1: start shell script

Step 2: clear the screen.

Step 3: take a year input by the user.

Step 4: implement if-else statement.

Step 5: give condition #year % 4

Step 6: if result ==0 then it is leap year otherwiser it is not

Step 7: stop shell script

CODE:

```
leap=$(date +%Y)
echo taking year as $leap
if [ $((leap % 4)) -eq 0 ]
then
echo leap year
else
echo is not a leap
year
fi
```

OUTPUT:

taking year as 2023

2023 is not a leap year

4. Write a Shell program to find the factorial of a number

ALGORITHM:

1. Get a number
2. Use for loop or while loop to compute the factorial by using the below formula
3. fact(n) = n * n-1 * n-2 * .. 1
4. Display the result.

CODE:

```
#shell script for factorial of a number #factorial using while loop
echo ”Enter a number”
read num
fact=1
while [ $num -gt 1 ]
do
fact=$((fact * num))
num=$((num - 1))
done
echo $fact
```

OUTPUT:

```
Enter a number: 3
6
Enter a number: 4
24
```

5) Write a Shell program to swap the two integers.

ALGORITHM:

1. Store the value of the first number into a temp variable.
2. Store the value of the second number in the first number.
3. Store the value of temp into the second variable.

CODE:

```
# Static input of the
# numbers
first=5
second=10

temp=$first
first=$second
second=$temp

echo "After swapping, numbers are:"
echo "first = $first, second = $second"
```

OUTPUT:

After swapping, numbers are:

first = 10, second = 5

6) Write a Shell program to find the greatest of two integers.

ALGORITHM:

1. Enter and Read X
2. Enter and Read Y
3. Use if statement to check the condition for greater than using -gt
4. Also Check the condition for lesser than using -lt using elif
5. Check the condition for equal using -eq using elif
6. Print according to the conditions in the loop

CODE:

```
echo “Enter X”
read X
echo “Enter Y”
read Y
if [ $X –gt $Y ]
then
echo X is greater than Y
elif [ $X –lt $Y ]
then
echo X is lesser than Y
elif [ $X –eq $Y]
then
echo X is equal to Y
fi
```

OUTPUT:

```
Enter X
10
Enter Y
5
X is greater than Y
```