SIMATS ENGINEERING
Engineer to Excel

SIMATS
Saveetha Institute of Medical And Technical Sciences
(Declared as Deemed to be University under Section 3 of UGC Act 1956)

CSA0477

OPERATING SYSTEMS

HARIGOWTHAM. A
192324198

S. NO.	LAB EXPERIMENTS	CO
1	Create a new process by invoking the appropriate system call. Get the process identifier of the currently running process and its respective parent using system calls and display the same using a C program.	CO1
2	Identify the system calls to copy the content of one file to another and illustrate the same using a C program.	CO1
3	Design a CPU scheduling program with C using First Come First Served technique with the following considerations. a. All processes are activated at time 0. b. Assume that no process waits on I/O devices	CO2
4	Construct a scheduling program with C that selects the waiting process with the smallest execution time to execute next	CO2
5	Construct a scheduling program with C that selects the waiting process with the highest priority to execute next.	CO2
6	Construct a C program to implement preemptive priority scheduling algorithm	CO2
7	Construct a C program to implement a non-preemptive SJF algorithm.	CO2
8	Construct a C program to simulate Round Robin scheduling algorithm with C.	CO2
9	Illustrate the concept of inter-process communication using shared memory with a C program.	CO3
10	Illustrate the concept of inter-process communication using message queue with a C program.	CO3
11	Illustrate the concept of multithreading using a C program.	CO3
12	Design a C program to simulate the concept of Dining-Philosophers problem	CO3
13	Construct a C program for implementation of the various memory allocation strategies.	CO3
14	Construct a C program to organise the file using a single level directory.	CO3
15	Design a C program to organise the file using a two level directory structure.	CO3

16	Develop a C program for implementing random access file for processing the employee details.	CO3
17	Illustrate the deadlock avoidance concept by simulating Banker’s algorithm with C.	CO2

18	Construct a C program to simulate producer-consumer problem using semaphores.	CO2
19	Design a C program to implement process synchronization using mutex locks.	CO2
20	Construct a C program to simulate Reader-Writer problem using Semaphores.	CO2
21	Develop a C program to implement the worst fit algorithm of memory management.	CO3
22	Construct a C program to implement the best fit algorithm of memory management.	CO3
23	Construct a C program to implement the first fit algorithm of memory management.	CO3
24	Design a C program to demonstrate UNIX system calls for file management.	CO1
25	Construct a C program to implement the I/O system calls of UNIX (fcntl, seek, stat, opendir, readdir)	CO5
26	Construct a C program to implement the file management operations.	CO4
27	Develop a C program for simulating the function of ls UNIX Command.	CO5
28	Write a C program for simulation of GREP UNIX command	CO5
29	Write a C program to simulate the solution of Classical Process Synchronization Problem	CO2
30	Write C programs to demonstrate the following thread related concepts. (i)create (ii) join (iii) equal (iv) exit	CO2
31	Construct a C program to simulate the First in First Out paging technique of memory management.	CO3
32	Construct a C program to simulate the Least Recently Used paging technique of memory management.	CO3

33	Construct a C program to simulate the optimal paging technique of memory management	CO3
34	Consider a file system where the records of the file are stored one after another both physically and logically. A record of the file can only be accessed by reading all the previous records. Design a C program to simulate the file allocation strategy.	CO4
35	Consider a file system that brings all the file pointers together into an index block. The ith entry in the index block points to the ith block of the file. Design a C program to simulate the file allocation strategy.	CO4
36	With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block. Design a C program to simulate the file allocation strategy.	CO4
37	Construct a C program to simulate the First Come First Served disk scheduling algorithm.	CO4
38	Design a C program to simulate SCAN disk scheduling algorithm.	CO4
39	Develop a C program to simulate C-SCAN disk scheduling algorithm.	CO4
40	Illustrate the various File Access Permission and different types of users in Linux.	CO4

1. Create a new process by invoking the appropriate system call. Get the process identifier of the currently running process and its respective parent using system calls and display the same using a C program.

AIM: Create a new process by invoking the appropriate system call. Get the process identifier of the currently running process and its respective parent using system calls and display the same using a C program.

ALGORITHM:

1. Include necessary headers: Include the necessary header files like <stdio.h> and <unistd.h> for using system calls.

2. Declare variables: Declare variables to hold the process ID (pid_t pid) and the parent process ID (pid_t ppid).

3. Get the current process ID: Use the getpid() system call to retrieve the process ID of the current process.

4. Get the parent process ID: Use the getppid() system call to retrieve the parent process ID.

5. Display the process IDs: Print the process IDs to the console.

6. Create a new process: Use the fork() system call to create a new process. Check the return value of fork() to determine whether the code is running in the parent or child process.

7. Display process IDs in child and parent processes: Depending on whether the process is the parent or the child, display the respective process IDs.

PROGRAM:

```
#include<stdio.h>
#include<unistd.h> int
main()
{ printf("Process ID: %d\n", getpid() );
  printf("Parent Process ID: %d\n", getpid() ); return
  0;
}
```

OUTPUT


C:\Users\jaswa\OneDrive\Doc
Process ID: 31556
Parent Process ID: 31556

Process exited after 0.05791 seconds with return value 0
Press any key to continue . . .

2. Identify the system calls to copy the content of one file to another and illustrate the same using a C program

AIM: Identify the system calls to copy the content of one file to another and illustrate the same using a C program

ALGORITHM:

- Include necessary headers:
- Include the necessary header files like <stdio.h>, <fcntl.h>, and <unistd.h> for working with file-related system calls.
- Declare variables:
- Declare variables to hold file descriptors, buffer, and other necessary information.
- Open the source and destination files:
- Use the open system call to open the source and destination files, obtaining file descriptors for each.
- Read from the source file:
- Use the read system call to read data from the source file into a buffer. Write to the destination file:
- Use the write system call to write the data read from the source file into the destination file.
- Close the files:

Use the close system call to close the source and destination files.

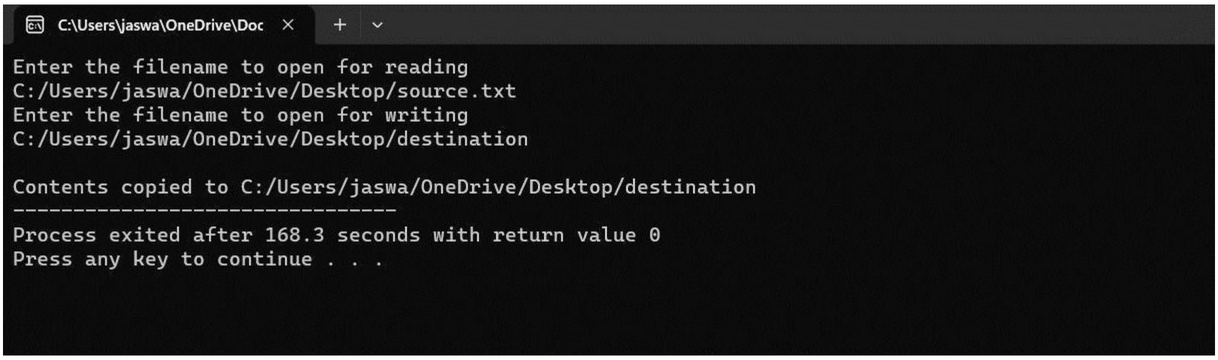
PROGRAM:

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

int main()
{
   FILE *fptr1, *fptr2; char
   filename[100], c;

   printf("Enter the filename to open for reading \n");
   scanf("%s", filename);
   fptr1 = fopen(filename, "r");i f
   (fptr1 == NULL)
   { printf("Cannot open file %s \n", filename);
      exit(0);
```

```
}

printf("Enter the filename to open for writing \n");
scanf("%s", filename);
fptr2 = fopen(filename, "w"); if
(fptr2 == NULL)
{ printf("Cannot open file %s \n", filename);
   exit(0);
}       c       =
fgetc(fptr1);
while (c != EOF)
{ fputc(c, fptr2); c
   = fgetc(fptr1);
}

printf("\nContents copied to %s", filename);

fclose(fptr1);
fclose(fptr2);
return 0;

}
```

OUTPUT

3. Design a CPU scheduling program with C using First Come First Served



technique with the following considerations.

a. All processes are activated at time 0.

b. Assume that no process waits on I/O devices.

AIM: Design a CPU scheduling program with C using First Come First Served technique with the following considerations.

a. All processes are activated at t ime 0.

b. Assume that no process waits on I/O devices.

ALGORITHM:

1. Include necessary headers:
2. Include the necessary header files like <stdio.h> for input/output operations.
3. Define the process structure:
4. Define a structure to hold the process information, such as process ID, arrival time, burst time, waiting time, and turnaround time.
5. Input the number of processes and their details:

   Input the number of processes and their arrival time and burst time.

6. Sort processes by arrival time (if not given):

   If the processes are not already sorted by arrival time, sort them based 7.Calculate waiting time and turnaround time.

1. Display the scheduling information:

   Display the process details including process ID, arrival time, burst time, waiting time, and turnaround time.

PROGRAM:

```
#include <stdio.h> int
main()
{ int A[100][4];
        int i, j, n, total = 0, index, temp; float
        avg_wt,   avg_tat;   printf("Enter
        number of process: "); scanf("%d",
        &n);
        printf("Enter Burst Time:\n"); for (i
        = 0; i < n; i++) { printf("P%d:
                ", i + 1); scanf("%d",
                &A[i][1]); A[i][0] = i
                + 1;
        }
        for (i = 0; i < n; i++) {
                index = i;
```

```
for (j = i + 1; j < n; j++) if
        (A[j][1] < A[index][1])
        index = j;
    temp = A[i][1]; A[i][1] =
    A[index][1]; A[index][1]
    = temp;


    temp = A[i][0]; A[i][0] =
    A[index][0]; A[index][0]
    = temp; } A[0][2] =
0; for (i = 1; i < n; i++) {
A[i][2] = 0; for (j = 0; j < i;
j++)
        A[i][2] += A[j][1];
    total += A[i][2];
} avg_wt = (float)total /
n; total = 0;
printf("P      BT    WT    TAT\n");

for (i = 0; i < n; i++) {
    A[i][3] = A[i][1] + A[i][2];
    total += A[i][3];
    printf("P%d %d      %d    %d\n", A[i][0],A[i][1],
A[i][2],  A[i][3]);  }  avg_tat  =
(float)total / n;
printf("Average Waiting Time= %f", avg_wt); printf("\nAverage
Turnaround Time= %f", avg_tat);
}
```

OUTPUT

```
Enter number of process: 4
Enter Burst Time:
P1: 12
P2: 14
P3: 15
P4: 16
P        BT      WT      TAT
P1       12      0       12
P2       14      12      26
P3       15      26      41
P4       16      41      57
Average Waiting Time= 19.750000
Average Turnaround Time= 34.000000
--------------------------------
Process exited after 17.9 seconds with return value 0
Press any key to continue . . .
```

4. Construct a scheduling program with C that selects the waiting process with the smallest execution time to execute next.

AIM: Construct a scheduling program with C that selects the waiting process with the smallest execution time to execute next.

ALGORITHM:

1. Include necessary headers:

Include the necessary header files like <stdio.h> for input/output operations.

2. Define the process structure:

3. Define a structure to hold the process information, such as process ID, arrival time, burst time, waiting time, and turnaround time.
4. Input the number of processes and their details:

Input the number of processes and their arrival time and burst time. 4.Sort processes by burst time:

Sort the processes based on burst time in ascending order. 5.Calculate waiting time and turnaround time:

Calculate the waiting time and turnaround time for each process based on the SJF scheduling algorithm.

6.Display the scheduling information:

Display the process details including process ID, arrival time, burst time, waiting time, and turnaround time.

PROGRAM:

```
#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("nProcesst    Burst Time       tWaiting TimetTurnaround Time\n");
for(i=0;i<n;i++)
{ tat[i]=bt[i]+wt[i];
    total+=tat[i];
    printf("np%dtt
    %dtt
      %dttt%d",p[i],b
    t[i],wt[i],tat[i]);
```

```
    } avg_tat=(float)total/n; printf("nnAverage Waiting
    Time=%f",avg_wt); printf("nAverage Turnaround
    Time=%fn",avg_tat);
}
```

OUTPUT

```
Enter number of process:3
nEnter Burst Time:
p1:45
p2:32
p3:18
nProcesst    Burst Time    tWaiting TimetTurnaround Time
np3tt  18tt    0ttt18np2tt  32tt    18ttt50np1tt  45tt    50ttt95nnAverage Waiting Time=22.666666nAverage Turnaround Time=54.333332n
--------------------------------
Process exited after 8.49 seconds with return value 0
Press any key to continue . . .
```

5. Construct a scheduling program with C that selects the waiting process with the highest priority to execute next.

Aim:- Construct a scheduling program with C that selects the waiting process with the highest priority to execute next.

Algorithm:-

1. Initialize the necessary data structures to store process information, including process ID, arrival time, burst time, and priority.
2. Read the number of processes (N) from the user.
3. For each process, read the following information:
4. Process ID (PID)
5. Arrival Time
6. Burst Time (time required for execution)
7. Priority (lower values indicate higher priority)
8. Sort the processes based on their arrival time in ascending order.
9. Initialize a variable current_time to 0 (representing the current time in the simulation).
10. Initialize a variable completed to 0 (to keep track of the number of completed processes).
11. Create a priority queue or data structure to store processes based on their priority.

12. Initialize a variable total_waiting_time to 0.
13. While there are still processes to execute (i.e., completed < N), repeat the following:
    a. For each process that has arrived but has not been completed, add it to the priority queue.
    b. Pop the process with the highest priority from the queue.
    c. Calculate the waiting time for the process as current_time - arrival time.
    d. Add the waiting time to total_waiting_time.
    e. Update current_time by adding the process's burst time.
    f. Print the process ID, arrival time, burst time, waiting time, and turnaround time.
    g. Mark the process as completed.
14. Calculate the average waiting time as total_waiting_time / N. 15.Print the average waiting time.

Program:-

```
#include<stdio.h> struct
priority_scheduling { char
  process_name; int
burst_time; int
waiting_time; int
turn_around_time; int
priority; }; int main() {
```

```
int number_of_process; int
total = 0;
struct priority_scheduling temp_process; int
ASCII_number = 65;
int          position;          float
average_waiting_time;       float
average_turnaround_time;
printf("Enter the total number of Processes: "); scanf("%d",
& number_of_process);
struct priority_scheduling process[number_of_process]; printf("\nPlease Enter the
Burst Time and Priority of each process:\n"); for (int i = 0; i < number_of_process;
i++) {
  process[i].process_name = (char) ASCII_number;
  printf("\nEnter    the    details of    the    process      %c    \n",
      process[i].process_name);
  printf("Enter the burst time: "); scanf("%d", &
  process[i].burst_time); printf("Enter the
  priority: "); scanf("%d", &
  process[i].priority); ASCII_number++; }
for (int i = 0; i < number_of_process; i++) {
  position = i;
  for (int j = i + 1; j < number_of_process; j++) {
    if (process[j].priority > process[position].priority) position
     = j; } temp_process = process[i];
  process[i] =
process[position]; process[position] = temp_process;
} process[0].waiting_time =
0; for (int i = 1; i < number_of_process;
i++) {
  process[i].waiting_time = 0;
  for (int j = 0; j < i; j++) {
    process[i].waiting_time += process[j].burst_time;          }
  total  +=  process[i].waiting_time;  }  average_waiting_time  =
(float) total / (float) number_of_process; total = 0;

  printf("\n\nProcess_name \t Burst Time \t Waiting Time \t              Turnaround
```

```
    Time\n");
printf("________________________________________________\n");
for (int i = 0; i < number_of_process; i++) {
  process[i].turn_around_time     =     process[i].burst_time  +
    process[i].waiting_time;
  printf("\t %c \t\t %d \t\t %d \t\t %d", process[i].process_name, process[i].burst_time,
        process[i].waiting_time, process[i].turn_around_time);
  printf("\n________________________________________________\n"); }
average_turnaround_time = (float) total / (float) number_of_process; printf("\n\n
Average Waiting Time : %f", average_waiting_time); printf("\n Average
Turnaround Time: %f\n", average_turnaround_time); return 0;
```

```
Enter the total number of Processes: 3

Please Enter the  Burst Time and Priority of each process:

Enter the details of the process A
Enter the burst time: 2
Enter the priority: 1

Enter the details of the process B
Enter the burst time: 10
Enter the priority: 3

Enter the details of the process C
Enter the burst time: 6
Enter the priority: 2


Process_name     Burst Time      Waiting Time     Turnaround Time
------------------------------------------------------------
          B              10              0                10
-----------------------------------------------------------
          C              6               10               16
-----------------------------------------------------------
          A              2               16               18
-----------------------------------------------------------


 Average Waiting Time : 8.666667
 Average Turnaround Time: 14.666667
```

6. Construct a C program to simulate Round Robin scheduling algorithm with C.

Aim:- Construct a C program to simulate Round Robin scheduling algorithm with C.

Algorithm:-

1. Initialize the necessary data structures to store process information, including process ID, burst time, and remaining time.
2. Read the number of processes (N) from the user.
3. Read the time quantum (slice time) from the user.
4. For each process, read the following information:
5. Process ID (PID)
6. Burst Time (time required for execution)
7. Create a queue data structure to store the processes.
8. Enqueue all processes into the queue.
9. Initialize a variable current_time to 0 (representing the current time in the simulation). 10. Initialize a variable total_waiting_time to 0. 11.While the queue is not empty, repeat the following:
   a. Dequeue a process from the front of the queue.
   b. Calculate the execution time for the process, which is the minimum of the time quantum and the remaining time for the process.
   c. Update the process's remaining time.
   d. Update current_time by adding the execution time.
   e. If the process still has remaining time, enqueue it back into the queue.
   f. Calculate the waiting time for the process as current_time - arrival time, where arrival time is the time when the process was first enqueued.
   g. Add the waiting time to total_waiting_time. 12.Calculate the average waiting time as total_waiting_time / N. 13.Print the average waiting time.

Program:-

```
#include<stdio.h>

#include<conio.h> int

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;

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

printf(" \nBurst time is: \t");

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

```
bt[i];
}
printf("Enter the Time Quantum for the process: \t"); scanf("%d", &quant);
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)
{ 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--;
            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; avg_tat = tat
* 1.0/NOP;
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: 3

 Enter the Arrival and Burst time of the Process[1]
 Arrival time is:       2

Burst time is:  33334

 Enter the Arrival and Burst time of the Process[2]
 Arrival time is:       23

Burst time is:  45

 Enter the Arrival and Burst time of the Process[3]
 Arrival time is:       27

Burst time is:  67
Enter the Time Quantum for the process:         9

 Process No              Burst Time              TAT             Waiting Time
Process No[2]            45                              121                     76
Process No[3]            67                              175                     108
Process No[1]            33334                           33444                   110
 Average Turn Around Time:      98.000000
 Average Waiting Time:  11246.666992
```

7. Construct a C program to implement non-preemptive SJF algorithm

AIM: Construct a C program to implement non-preemptive SJF algorithm

ALGORITHM:

1. Include necessary headers:
   Include the necessary header files like <stdio.h> for input/output operations.
2. Define the process structure:
   Define a structure to hold the process information, such as process ID, arrival time, burst time, waiting time, and turnaround time.
3. Input the number of processes and their details:
   Input the number of processes and their arrival time and burst time.
4. Sort processes by burst time:
   Sort the processes based on burst time in ascending order.
5. Calculate waiting time and turnaround time:
   Calculate the waiting time and turnaround time for each process based on the nonpreemptive SJF scheduling algorithm.
6. Display the scheduling information:
   Display the process details including process ID, arrival time, burst time, waiting time, and turnaround time.

PROGRAM:

```
#include<stdio.h> int
main()
{
        int at[10],bt[10],pr[10]; int
n,i,j,temp,time=0,count,over=0,sum_wait=0,sum_turnaround=0,start;
```

```
float  avgwait,avgturn;  printf("Enter  the
number of processes\n"); scanf("%d",&n);
for(i=0;i<n;i++)
{    printf("Enter the arrival time and execution time for process
%d\n",i+1);
        scanf("%d%d",&at[i],&bt[i]);
        pr[i]=i+1;
}
for(i=0;i<n-1;i++)
{ for(j=i+1;j<n;j++)
        { if(at[i]>at[j])
                { temp=at[i];
                        at[i]=at[j];
                        at[j]=temp;
                        temp=bt[i];
                        bt[i]=bt[j];
                        bt[j]=temp;
                        temp=pr[i];
                        pr[i]=pr[j];
                        pr[j]=temp;
                }

        }
}
printf("\n\nProcess\t|Arrival          time\t|Execution          time\t|Start
time\t|End time\t|waiting                    time\t|Turnaround time\n\n");
while(over<n)
{ count=0;
        for(i=over;i<n;i++)
        {
                if(at[i]<=time)
                count++; else
                break;



        }
        if(count>1)
        { for(i=over;i<over+count-1;i++)
                { for(j=i+1;j<over+count;j++)
                        { if(bt[i]>bt[j])
                                { temp=at[i];
                                        at[i]=at[j];
                                        at[j]=temp;
                                        temp=bt[i];
                                        bt[i]=bt[j];
                                        bt[j]=temp;
                                        temp=pr[i]
```

```
                                                        ;
                                                        pr[i]=pr[j];
                                                        pr[j]=temp
                                                        ;
                                                }
                                        }
                                }
                        } start=time;
                        time+=bt[over];

printf("p[%d]\t|\t%d\t|\t%d\t|\t%d\t|\t%d\t|\t%d\t|\t%d\n",pr[over],
                                at[over],bt[over],start,time,time-at[over]-
bt[over],time-at[over]);
                        sum_wait+=time-at[over]-bt[over];



                        sum_turnaround+=time-at[over];
                        over++;
                }
                avgwait=(float)sum_wait/(float)n;
                avgturn=(float)sum_turnaround/(float)n;     printf("Average
                waiting time is %f\n",avgwait); printf("Average turnaround
                time is %f\n",avgturn); return 0;
}
```

OUTPUT

```
Enter the number of processes
3
Enter the arrival time and execution time for process 1
 1 3
Enter the arrival time and execution time for process 2
2 6
Enter the arrival time and execution time for process 3
3 8


Process |Arrival time   |Execution time |Start time     |End time       |waiting

p[1]    |       1       |       3       |       0       |       3       |       -1      |       2
p[2]    |       2       |       6       |       3       |       9       |       1       |       7
p[3]    |       3       |       8       |       9       |       17      |       6       |       14
Average waiting time is 2.000000
Average turnaround time is 7.666667

--------------------------------
Process exited after 20.18 seconds with return value 0
Press any key to continue . . .
```

8. Construct a C program to simulate Round Robin scheduling algorithm with C.

AIM: Construct a C program to simulate Round Robin scheduling algorithm with C.

ALGORITHM:

1. Include necessary headers:
Include the necessary header files like <stdio.h> for input/output operations.
2. Define the process structure:
Define a structure to hold the process information, such as process ID, arrival time, burst time, waiting time, and turnaround time.
3. Input the number of processes and their details:
Input the number of processes and their arrival time and burst time. 4.Input the time quantum for Round Robin:
Input the time quantum to be used in the Round Robin scheduling. 5.Simulate Round Robin scheduling:
Implement the Round Robin scheduling algorithm, including a queue to keep track of the processes.
6.Display the scheduling information:
Display the process details including process ID, arrival time, burst time, waiting time, and turnaround time.

PROGRAM:

```
#include<stdio.h> #include<conio.h> int 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;
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"); scanf("%d", &at[i]);


scanf("%d", &bt[i]); temp[i] = bt[i]; } printf("Enter
the Time Quantum for the process: \t");
scanf("%d", &quant);
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)
{ 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--;
       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; avg_tat
= tat * 1.0/NOP;
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:       1

Burst time is:  23

 Enter the Arrival and Burst time of the Process[2]
 Arrival time is:       2

Burst time is:  32

 Enter the Arrival and Burst time of the Process[3]
 Arrival time is:       3

Burst time is:  2

 Enter the Arrival and Burst time of the Process[4]
 Arrival time is:       4

Burst time is:  45
Enter the Time Quantum for the process:         5

 Process No              Burst Time              TAT             Waiting Time
Process No[3]            2                               9                       7
Process No[1]            23                              64                      41
Process No[2]            32                              85                      53
Process No[4]            45                              98                      53
 Average Turn Around Time:      38.500000
 Average Waiting Time:  64.000000
```

9 Illustrate the concept of inter-process communication using shared memory with a C program

AIM:

To implement the concept of inter-process communication using shared memory using C programming.

ALGORITHM:

1. Create a shared memory segment:

- Use shmget() function to create a new shared memory segment or get the identifier of an existing one.
- Ensure to handle errors if the shared memory creation fails.

2. Attach shared memory to processes:

- Use shmat() function to attach the shared memory segment to the process address space.
- This allows processes to read and write data to the shared memory.

3. Read/Write data in shared memory:

- Processes can read and write data directly to the shared memory location.
- Ensure proper synchronization mechanisms (like semaphores) are used to avoid race conditions and maintain data consistency.

4. Detach shared memory and clean up:

- Use shmdt() function to detach the shared memory segment from the process when done.
- Optionally, remove the shared memory segment using shmctl() with the IPC_RMID command.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
#define SHM_SIZE 1024 // Size of the shared memory segment int main() {
    key_t key = ftok("shmfile", 65); // Generate a unique key for the shared
memory segment

    // Create a new shared memory segment (or get the identifier of an existing
one) int shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666); if (shmid
    == -1) {
        perror("shmget");
        exit(EXIT_FAILURE);
    }

    // Attach the shared memory segment to the process address space
    char *shm_ptr = (char*)shmat(shmid, NULL, 0); if
    (shm_ptr == (char*)(-1)) {
        perror("shmat");
        exit(EXIT_FAILURE);
    }

    // Write data to the shared memory strcpy(shm_ptr,
    "Hello, shared memory!");

    // Detach the shared memory segment from the process if
    (shmdt(shm_ptr) == -1) {
        perror("shmdt");
        exit(EXIT_FAILURE);
    }
printf("Data written to shared memory: %s\n", shm_ptr);

    // Optional: Remove the shared memory segmenti f
    (shmctl(shmid, IPC_RMID, NULL) == -1)
        { perror("shmctl");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

OUTPUT:

```vbnet
Data written to shared memory: Hello, shared memory!
```

10. Illustrate the concept of inter-process communication using message queue with a c program

AIM :
To implement the concept of inter-process communication using message queue with a c program

ALGORITHM :

1. Create a message queue:
   i. Use msgget() function to create a new message queue or get the identifier of an existing one.
   ii. Ensure to handle errors if the message queue creation fails.
2. Send a message to the queue (Producer process):
   i. Define a structure for the message containing necessary data fields.
   ii. Populate the message structure with appropriate data. iii. Use msgsnd() function to send the message to the message queue.
   iv. Handle errors if message sending fails.
3. Receive a message from the queue (Consumer process):
   i. Define a structure for the message to receive data. ii. Use msgrcv() function to receive a message from the message queue. iii. Process the received message as needed. iv. Handle errors if message receiving fails.
4. Remove the message queue (Optional):
   i. Use msgctl() function with IPC_RMID command to remove the message queue when it's no longer needed.

PROGRAM :

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

struct message {
    long msg_type;
    char msg_text[100];
```

```
};

int main() { key_t key = ftok("msgqfile", 65); // Generate a unique key for
    the message
queue

    // Create a new message queue (or get the identifier of an existing one) int msgid
    = msgget(key, IPC_CREAT | 0666);
    if (msgid == -1) {
        perror("msgget");
        exit(EXIT_FAILURE);
    }

    struct message msg;
    msg.msg_type = 1; // Message type (can be any positive number)

    // Producer: Send a message to the message queue
    strcpy(msg.msg_text, "Hello, message queue!"); if
    (msgsnd(msgid, (void*)&msg, sizeof(msg.msg_text),
IPC_NOWAIT) == -1) { perror("msgsnd");
        exit(EXIT_FAILURE);
    }
printf("Producer: Data sent to message queue: %s\n", msg.msg_text);

    // Consumer: Receive a message from the message queue if
    (msgrcv(msgid, (void*)&msg, sizeof(msg.msg_text), 1, 0) == -1) {
    perror("msgrcv"); exit(EXIT_FAILURE);
    }

    printf("Consumer: Data received from message queue: %s\n",
msg.msg_text);

    // Remove the message queue if
    (msgctl(msgid, IPC_RMID, NULL) == -1) {
    perror("msgctl"); exit(EXIT_FAILURE);
    }

    return 0;
}
```

OUTPUT :

```arduino
Producer: Data sent to message queue: Hello, message queue!
Consumer: Data received from message queue: Hello, message queue!
```

11. Illustrate the concept of multithreading using a C program

AIM :

To implement the concept of multithreading using C program

ALGORITHM :

Include Necessary Libraries:

Include the necessary header files for multithreading. For POSIX threads in C, include pthread.h.

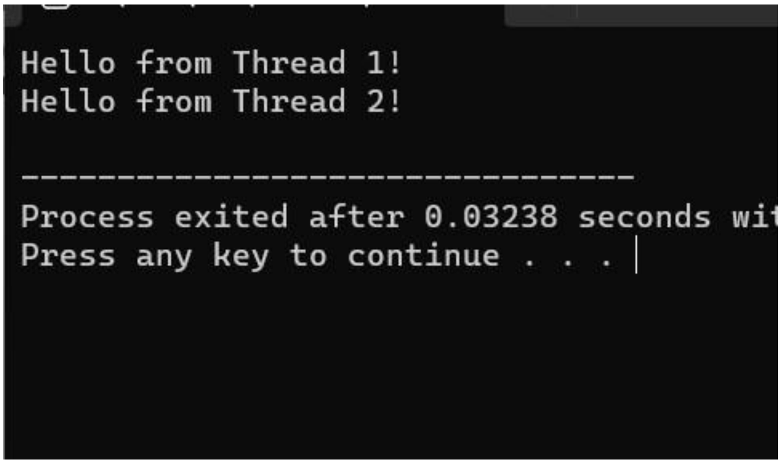
1. Define Thread Function:
   i. Define a function that will be executed by the threads. This function should have a void* return type and a void* parameter, which can be used to pass data to the thread.
2. Create Threads:
   i. In the main function or any other function, create thread objects using pthread_t.
   ii. Use pthread_create() to create threads and specify the thread function as well as any parameters to pass.
3. Thread Execution:
   i. Threads execute concurrently and perform the tasks defined in the thread function.
4. Synchronization and Coordination (Optional):
   i. Use synchronization mechanisms such as mutexes, semaphores, or condition variables to coordinate the execution of threads and ensure data consistency.
5. Wait for Threads to Complete (Optional):
   i. Use pthread_join() to wait for threads to finish their execution if the main thread needs to synchronize with the created threads.
   ii. Thread Termination (Optional):
   iii. Threads can exit by returning from the thread function or by calling pthread_exit(). The main thread can also call pthread_exit() to terminate the entire process.

PROGRAM :

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

void* threadFunction(void* arg) { char*
    message = (char*)arg; printf("%s\n",
    message);
    return NULL;
}

int main() { pthread_t
    thread1, thread2;
    char* message1 = "Hello from Thread 1!"; char*
    message2 = "Hello from Thread 2!";

    // Create threads
    pthread_create(&thread1, NULL, threadFunction, (void*)message1);
    pthread_create(&thread2, NULL, threadFunction, (void*)message2);
```

```
    // Wait for threads to complete pthread_join(thread1,
        NULL); pthread_join(thread2, NULL);

    return 0;
}
```

OUTPUT :



12. Design a C program to simulate the concept of Dining-Philosophers problem

AIM :

To design a C program to simulate the concept of Dining-Philosophers problem

ALGORITHM :

1. Initialize Semaphores:
   i. Create an array of semaphores, one for each philosopher, to control access to
the chopsticks.

ii. Initialize each semaphore to 1, indicating that the chopstick is available.

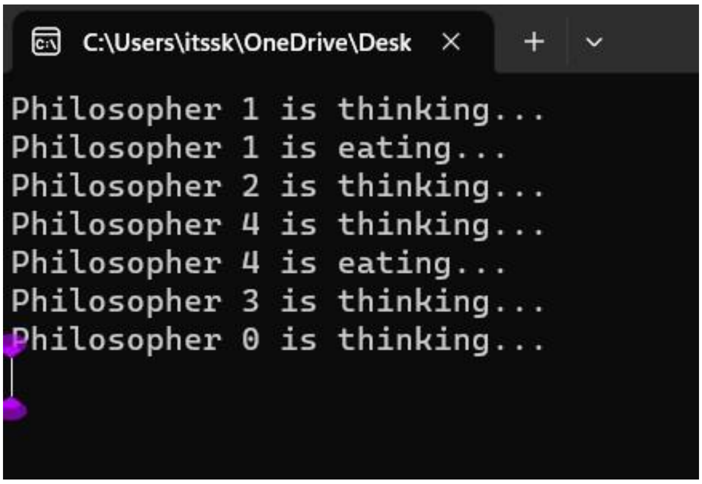
2. Define Philosopher Structure:
   i. Define a structure to represent a philosopher, which includes their ID and the semaphores representing the left and right chopsticks.
3. Philosopher Lifecycle:
   i. Each philosopher runs as a separate thread. In the thread function: ii. Think: Philosopher thinks for a random amount of time.
   iii. Pick up Chopsticks: Philosopher tries to pick up the left and right chopsticks (acquire semaphores). If both are available, the philosopher proceeds to eat. If not, they release the acquired chopstick(s) and return to thinking. iv. Eat: Philosopher eats for a random amount of time.
   v. Put Down Chopsticks: Philosopher releases the left and right chopsticks (release semaphores) after eating. vi. Repeat the above steps indefinitely to simulate the philosopher's life cycle.
4. Implement Deadlock Avoidance:
   i. To avoid deadlock, impose a constraint such that a philosopher can only pick up both chopsticks if both are available.
   ii. One way to achieve this is by introducing a global semaphore that limits the number of philosophers allowed to pick up chopsticks simultaneously. For example, if there are 5 philosophers, allow at most 4 philosophers to pick up chopsticks simultaneously. This prevents the circular wait condition and avoids deadlock.
5. Compile and Run:
   i. Compile the program with appropriate multithreading flags (e.g., -pthread for gcc on Unix-based systems). ii. Run the program to observe the dining philosophers problem simulation.

PROGRAM :

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

#define NUM_PHILOSOPHERS 5

pthread_mutex_t chopsticks[NUM_PHILOSOPHERS];

void* philosopherLifeCycle(void* arg) { int id =
    *((int*)arg); int
    left_chopstick = id;
    int right_chopstick = (id + 1) % NUM_PHILOSOPHERS;

      while (1) {
```

```
        // Think printf("Philosopher %d is
        thinking...\n", id);

        // Pick up chopsticks pthread_mutex_lock(&chopsticks[left_chopstick]);
        pthread_mutex_lock(&chopsticks[right_chopstick]);

        // Eat printf("Philosopher %d is
        eating...\n", id);
        sleep(rand() % 3 + 1); // Eating time

        // Put down chopsticks
        pthread_mutex_unlock(&chopsticks[left_chopstick]);
        pthread_mutex_unlock(&chopsticks[right_chopstick]);

        // Repeat the cycle
    }
}

int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS]; int
    philosopher_ids[NUM_PHILOSOPHERS];

    // Initialize mutex locks
    for (int i = 0; i < NUM_PHILOSOPHERS; ++i) {
        pthread_mutex_init(&chopsticks[i], NULL);
    }

    // Create philosopher threads for (int i = 0; i <
    NUM_PHILOSOPHERS;          ++i)          {
    philosopher_ids[i]           =           i;
    pthread_create(&philosophers[i],      NULL,
    philosopherLifeCycle,
(void*)&philosopher_ids[i]);
    }

    // Wait for threads to finish (although they run indefinitely) for (int i =
    0; i < NUM_PHILOSOPHERS; ++i) { pthread_join(philosophers[i],
        NULL);
    }

    // Destroy mutex locks
    for (int i = 0; i < NUM_PHILOSOPHERS; ++i)
    { pthread_mutex_destroy(&chopsticks[i]); }

    return 0;
}
```

OUTPUT :



13. Construct a C program to implement various memory allocation strategies.

AIM :

To construct a C program to implement various memory allocation strategies.

ALGORITHM :

1. Include Necessary Libraries:
   Include the necessary header files such as stdio.h, stdlib.h, etc.

2. Define Process Control Block (PCB) Structure:
   i. Define a structure to represent a Process Control Block (PCB) that
   ii. contains information about each process, including process ID, iii.
   memory size, and allocation status.

3. Implement Memory Allocation Functions:

i. Implement functions for memory allocation strategies like First Fit, Best Fit, and Worst Fit. ii. Each function should search for a suitable block of memory in the memory pool based on the specific strategy (first fit, best fit, or worst fit).

iii. Allocate memory to the process by updating the allocation status in the PCB and updating the memory pool accordingly.

4. Implement Memory Deallocation Function:
   i. Implement a function to deallocate memory occupied by a process.
   ii. Update the allocation status in the PCB and release the memory block, merging it with adjacent free blocks if necessary.
5. Main Function:
   i. In the main function, initialize the memory pool (an array representing the available memory).
   ii. Create PCBs for processes with specific memory requirements.
   iii. Call the appropriate memory allocation functions based on the desired strategy for each process.
   iv. Deallocate memory for completed processes using the memory deallocation function.
6. Print Memory Allocation Status:
   i. Implement a function to print the memory allocation status after each allocation and deallocation operation.
7. Compile and Run:
   i. Compile the program and run it to observe how different memory allocation strategies work.

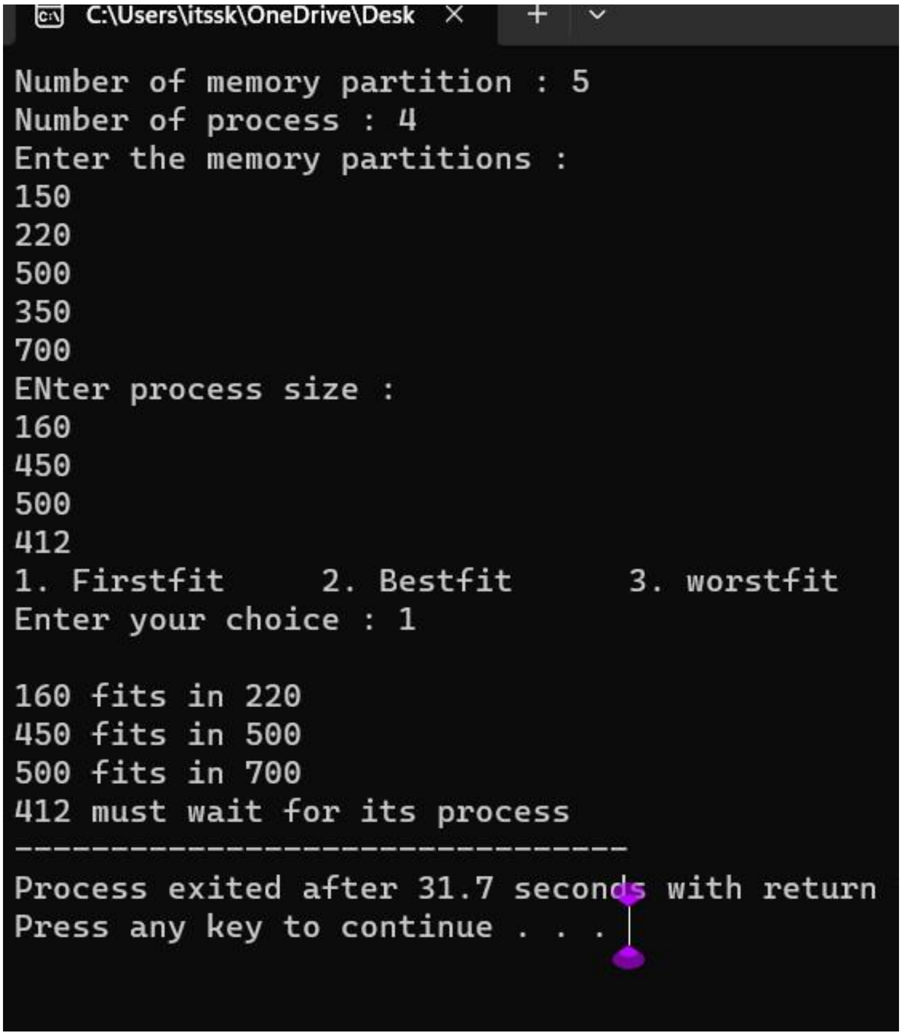
PROGRAM :

```
#include<stdio.h>
void bestfit(int mp[],int p[],int m,int n){ int j=0; for(int
        i=0;i<n;i++){ if(mp[i]>p[j]){ printf("\n%d fits in
        %d",p[j],mp[i]); mp[i]=mp[i]-p[j++]; i=i-1;
                }
        }
        for(int i=j;i<m;i++)
```

```
        {
                printf("\n%d must wait for its process",p[i]);
        }
}


void rsort(int a[],int n){ for(int
        i=0;i<n;i++){ for(int
        j=0;j<n;j++){ if(a[i]>a[j]){ int
        t=a[i]; a[i]=a[j]; a[j]=t;
                        }
                }
        }
}

void sort(int a[],int n){ for(int
        i=0;i<n;i++){ for(int
        j=0;j<n;j++){ if(a[i]<a[j]){ int
        t=a[i]; a[i]=a[j]; a[j]=t;
                        }
                }
        }
}
void firstfit(int mp[],int p[],int m,int n){
        sort(mp,n); sort(p,m);
        bestfit(mp,p,m,n);
}
void worstfit(int mp[],int p[],int m,int n){
        rsort(mp,n); sort(p,m);
        bestfit(mp,p,m,n);
} int main(){ int m,n,mp[20],p[20],ch; printf("Number
of memory partition : "); scanf("%d",&n);
printf("Number of process : "); scanf("%d",&m);
printf("Enter the memory partitions : \n"); for(int
i=0;i<n;i++){ scanf("%d",&mp[i]);
```

```
	}
	printf("ENter process size : \n");
	for(int i=0;i<m;i++){
	scanf("%d",&p[i]);

	}
	printf("1. Firstfit\t2. Bestfit\t3. worstfit\nEnter your choice :
	"); scanf("%d",&ch); switch(ch){ case 1: bestfit(mp,p,m,n);
	break; case 2: firstfit(mp,p,m,n); break; case 3:
	worstfit(mp,p,m,n); break;
	default:
		printf("invalid");
		break;
	}
}
```

OUTPUT :



```
C:\Users\itssk\OneDrive\Desk

Number of memory partition : 5
Number of process : 4
Enter the memory partitions :
150
220
500
350
700
ENter process size :
160
450
500
412
1. Firstfit     2. Bestfit      3. worstfit
Enter your choice : 1

160 fits in 220
450 fits in 500
500 fits in 700
412 must wait for its process
--------------------------------
Process exited after 31.7 seconds with return
Press any key to continue . . .
```

14. Construct a C program to organize the file using single level directory

AIM:

To construct a c program to organize the file using single level directory

ALGORITHM :

Step 1: Define Structures

Define structures to represent files and the directory. Step 2:

Initialize Directory

Create a function or code segment to initialize the directory structure. Set the initial file count to 0.

Step 3: Add Files

Implement a function or code segment to add files to the directory. This function should handle adding files, updating the file count, and handling errors if the directory is full.

Step 4: List Files

Create a function or code segment to list all the files in the directory. This function should iterate through the file list and print the file names.

Step 5: Delete Files (Optional)

Implement a function or code segment to delete files from the directory. This function should handle removing files, updating the file count, and handling errors if the file is not found.

Step 6: Implement User Interface

Create a user interface for interacting with the program. This could be a menudriven interface where users can choose to add files, list files, delete files, or exit the program.

Step 7: Test the Program

Compile the program using a C compiler and test it by adding files, listing files, and deleting files. Make sure the program handles different scenarios and errors gracefully.

Step 8: Refine and Expand (Optional)

Refine your program based on testing results. You can also expand the functionality by adding more features, error handling, or optimizing the code.

Step 9: Document Your Code (Optional)

Document your code by adding comments to explain the functionality of different sections of your program. This will make it easier for others (and yourself) to understand the code in the future.

Step 10: Compile and Distribute

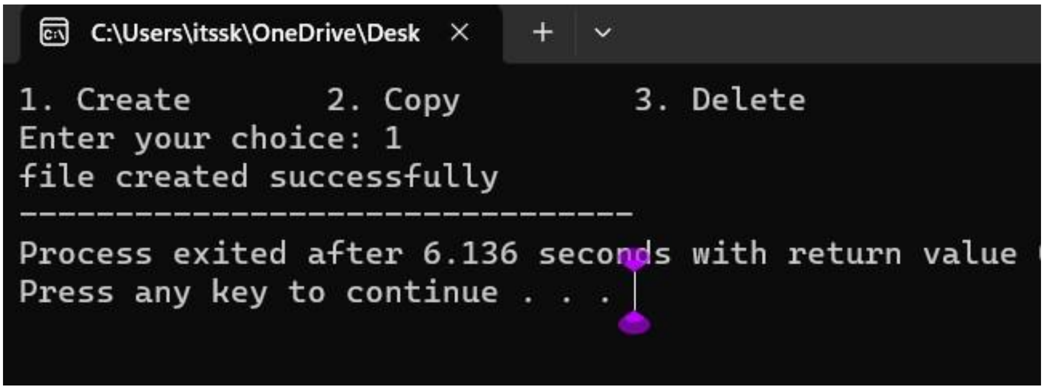
Once your program is complete and thoroughly tested, compile it into an executable file. If you want to distribute the program, you can create an installer or provide the executable along with necessary instructions.

PROGRAM :

```
#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <unistd.h>

#define BUFFER_SIZE 4096

void copy(){ const char

*sourcefile=

"C:/Users/itssk/OneDrive/Desktop/sasi.txt";        const        char

   *destination_file="C:/Users/itssk/OneDrive/Desktop/sk.txt"; int source_fd

   = open(sourcefile, O_RDONLY); int dest_fd = open(destination_file,

   O_WRONLY | O_CREAT | O_TRUNC,

0666); char buffer[BUFFER_SIZE]; ssize_t bytesRead, bytesWritten; while

   ((bytesRead = read(source_fd, buffer, BUFFER_SIZE)) > 0) { bytesWritten =

   write(dest_fd, buffer, bytesRead);

   }
```

```
    close(source_fd);
    close(dest_fd);
    printf("File copied successfully.\n");
} void
create()
{ char
path[100];
            FILE *fp; fp=fopen("C:/Users/itssk/OneDrive/Desktop/sasi.txt","w");
            printf("file created successfully");
}
int main(){
            int n;
            printf("1. Create \t2. Copy \t3. Delete\nEnter your choice: "
            ); scanf("%d",&n); switch(n){
                        case 1:
            create(); break;
            case 2:
                        copy();
                        break;
                        case 3:
                                    remove("C:/Users/itssk/OneDrive/Desktop/sasi.txt"); printf("Deleted
                                    successfully");
}}
```

OUTPUT :



15. Design a C program to organize the file using two level directory structure.

AIM :

To design a C program to organize the file using two level directory structure

Algorithm :

1. Define Structures: Define structures for files and directories. Each directory structure should contain an array for files and an array for subdirectories.
2. Initialize Root Directory: Create a root directory structure. This serves as the starting point for the two-level directory structure.
3. Add Files to Directories: Implement a function to add files to a specific directory. Handle adding files, updating the file count, and handling errors if the directory is full.
4. Add Subdirectories: Implement a function to add subdirectories to a specific directory. Manage adding directories, updating the directory count, and handling errors if the parent directory is full.
5. List Files and Subdirectories: Create functions to list all the files and subdirectories in a directory. These functions should iterate through the file and subdirectory arrays and print their names.
6. Delete Files and Subdirectories (Optional): Implement functions to delete files and subdirectories from a directory. Handle removing files or directories, updating the counts, and handling errors if the file or directory is not found.
7. Implement User Interface: Design a user interface for interacting with the program. This could be a menu-driven interface where users can add files, add subdirectories, list files, list subdirectories, delete files, delete subdirectories, or exit the program.
8. Test the Program: Compile the program and test it thoroughly. Add files, add subdirectories, list files, list subdirectories, delete files, and delete subdirectories. Ensure the program handles different scenarios and errors gracefully.

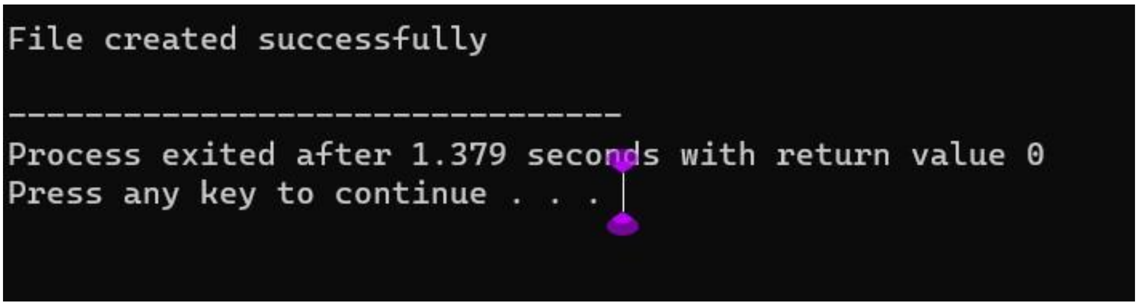
9. Refine and Expand (Optional): Refine the program based on testing results. Expand the functionality by adding more features, error handling, or optimizing the code.

10. Document Your Code (Optional): Document your code by adding comments to explain the functionality of different sections. This will make it easier for others to understand the code in the future.

PROGRAM :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h> int main() { char mainDirectory[] =
"C:/Users/itssk/OneDrive/Desktop"; char subDirectory[] = "os";
    char   fileName[]   =   "example.txt";
    char          filePath[200];          char
    mainDirPath[200];
   snprintf(mainDirPath, sizeof(mainDirPath), "%s/%s/", mainDirectory, subDirectory);
         snprintf(filePath, sizeof(filePath), "%s%s", mainDirPath, fileName); FILE *file
    = fopen(filePath, "w"); if (file ==
    NULL)  {  printf("Error  creating
    file.\n"); return 1;
    } fprintf(file, "This is an example file
content."); printf("File created successfully:
%s\n"); }
```

OUTPUT :



16. Develop a C program for implementing random access file for processing the employee details

AIM :

To develop a C program for implementing random access file for processing the employee details

ALGORITHM :

1. Define Structure: Define a structure to represent employee details. Include attributes like employee ID, name, salary, and any other relevant information.

2. Open File in Binary Mode: Open a file in binary mode using the fopen function. Specify the file path and mode ("rb+" for reading and writing binary files).

3. Menu-Driven Interface: Create a menu-driven interface for the user to perform operations. Options could include adding a new employee, updating existing employee details, searching for an employee, deleting an employee, listing all employees, and exiting the program.

4. Implement Functions: Implement functions corresponding to each menu option. For example, implement functions to add a new employee, update employee details, search for an employee by ID, delete an employee, and list all employees. These functions should perform file operations like reading and writing records.

5. Random Access File Operations: Utilize fseek and ftell functions to perform random access file operations. Use fseek to move the file pointer to the desired record based on the employee ID and ftell to determine the current position of the file pointer.

6. File Read and Write: Implement functions to read and write employee records to the file. Use fread and fwrite functions to read and write structures to the file.

7. Error Handling: Implement error handling to deal with situations where the file cannot be opened or when operations like adding, updating, or deleting employees fail. Display appropriate error messages to the user.

8. Close the File: Close the file using the fclose function when the program is exiting or when the file operations are completed.

9. Testing: Test the program thoroughly by adding, updating, searching, and deleting employee records. Ensure that the program handles edge cases and errors gracefully.

10. Documentation (Optional): Add comments and documentation to your code to explain the functionality of different sections, making it easier for others (and yourself) to understand the code in the future.

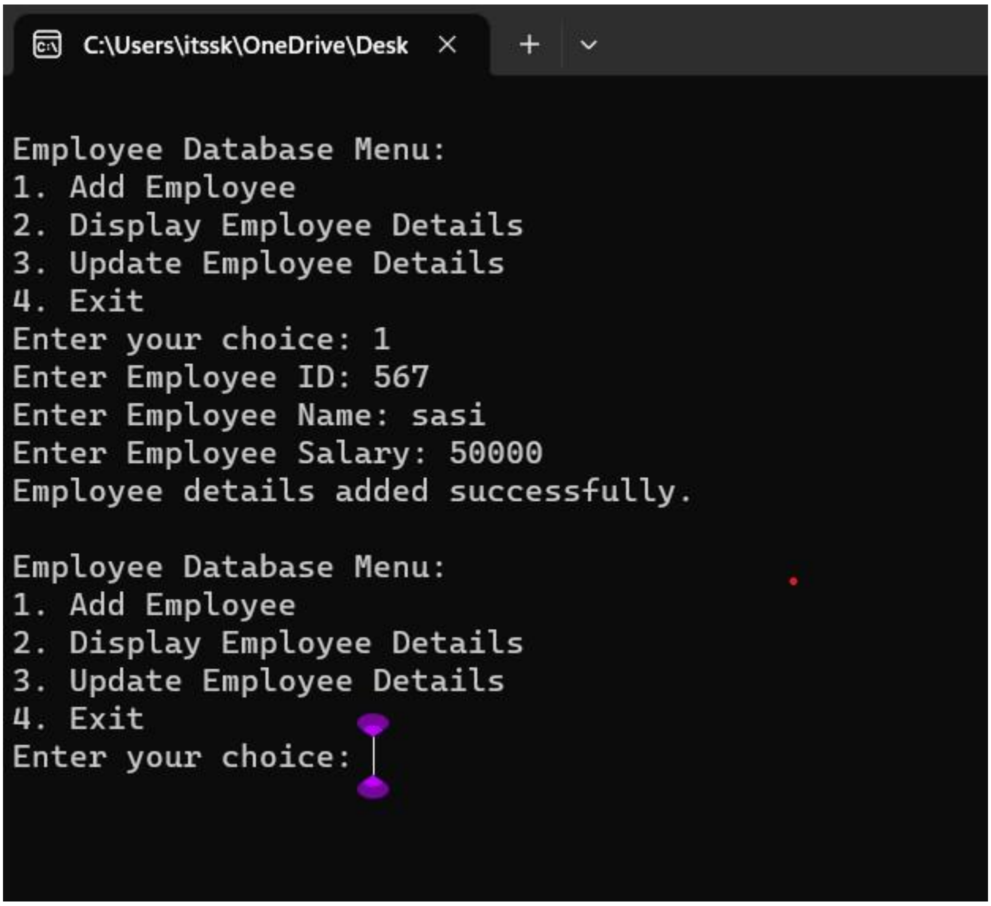
PROGRAM :

```
#include        <stdio.h>

#include        <stdlib.h>

struct Employee {

    int empId; char

    empName[50];

    float empSalary;};

int main() { FILE *filePtr; struct Employee

    emp; filePtr = fopen("employee.dat",

    "rb+"); if

    (filePtr == NULL) { filePtr =

        fopen("employee.dat", "wb+");i f

        (filePtr == NULL) { printf("Error

            creating the file.\n"); return 1;

                    }

    } int

    choice; do

    {

        printf("\nEmployee      Database      Menu:\n");

        printf("1. Add Employee\n"); printf("2. Display

        Employee    Details\n");    printf("3.    Update

        Employee Details\n");

        printf("4.                Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice); switch
```

```
(choice) {
    case 1:
        printf("Enter Employee ID: "); scanf("%d",
        &emp.empId); printf("Enter Employee
        Name: ");
        scanf("%s",         emp.empName);
        printf("Enter Employee Salary: ");
        scanf("%f", &emp.empSalary);
        fseek(filePtr, (emp.empId - 1) * sizeof(struct Employee),
SEEK_SET); fwrite(&emp,   sizeof(struct    Employee),     1,
            filePtr); printf("Employee details added
        successfully.\n"); break;
    case 2:
        printf("Enter Employee ID to display: "); scanf("%d",
        &emp.empId);
        fseek(filePtr, (emp.empId - 1) * sizeof(struct Employee),
SEEK_SET);
        fread(&emp, sizeof(struct Employee), 1, filePtr);
        printf("Employee ID: %d\n", emp.empId); printf("Employee
        Name: %s\n", emp.empName); printf("Employee Salary:
        %.2f\n", emp.empSalary); break;
    case 3:
        printf("Enter Employee ID to update: ");
        scanf("%d", &emp.empId);
        fseek(filePtr, (emp.empId - 1) * sizeof(struct Employee),
SEEK_SET);
        fread(&emp, sizeof(struct Employee), 1, filePtr);
        printf("Enter  Employee  Name:  ");  scanf("%s",
        emp.empName); printf("Enter Employee Salary:
        "); scanf("%f", &emp.empSalary);
        fseek(filePtr, (emp.empId - 1) * sizeof(struct Employee),
SEEK_SET);
        fwrite(&emp, sizeof(struct Employee), 1, filePtr);
        printf("Employee details updated successfully.\n"); break;
    case 4:
        break;
```

```
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice !=
    4); fclose(filePtr);
    return 0;
```

OUTPUT :



17. Illustrate the deadlock avoidance concept by simulating Banker's algorithm using C.

AIM :

To illustrate the deadlock avoidance concept by simulating Banker's algorithm using C.

ALGORITHM :

1. Define Data Structures: Define appropriate data structures to store the available resources, maximum resources, allocated resources, and need matrix for each process.

2. Initialize Matrices: Initialize the available, maximum, allocated, and need matrices based on the system's resources and the maximum demand of each process.

3. Input Request: Implement a function to input resource request from processes. This function should validate if the request is within the maximum limit specified by each process.

4. Safety Algorithm: Implement the Banker's safety algorithm to check if the system is in a safe state. Use the available, allocated, and need matrices to determine if the system can allocate resources to processes without entering into a deadlock state.

5. Resource Allocation: Implement resource allocation functions to handle the request from processes. Check if the request can be granted safely using the Banker's algorithm. If the request can be granted, update the allocated and available matrices accordingly.

6. User Interface: Create a user interface to interact with the program. Allow users to input resource requests and display the current state of the system, including available resources and resource allocation status.

7. Deadlock Scenario: Introduce scenarios where a deadlock can potentially occur (e.g., requesting more resources than available or requesting resources in a circular wait condition).

8. Testing: Test the program with different resource request scenarios. Ensure that the system handles requests properly without entering into a deadlock state. Also, test scenarios where the system should deny requests to prevent deadlock.

9. Documentation (Optional): Add comments and documentation to the code to explain the functionality of different sections. Document the Banker's algorithm steps and how it prevents deadlocks in the system.

10. Error Handling: Implement error handling mechanisms to deal with invalid input, unexpected scenarios, or any issues that might occur during resource allocation and deallocation.

PROGRAM :

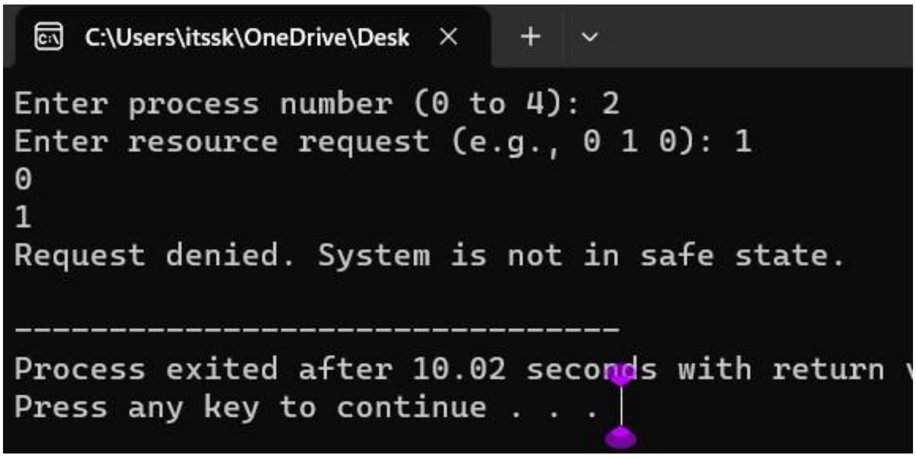
```
#include <stdio.h>


#define MAX_PROCESSES 5

#define MAX_RESOURCES 3 int

is_safe(); int available[MAX_RESOURCES] = {3, 3, 2}; // Available instances of

each resource

int maximum[MAX_PROCESSES][MAX_RESOURCES] = {{7, 5, 3},
{3, 2, 2}, {9, 0, 2}, {2, 2, 2}, {4, 3, 3}};

int allocation[MAX_PROCESSES][MAX_RESOURCES] = {{0, 1, 0},
{2, 0, 0}, {3, 0, 2}, {2, 1, 1}, {0, 0, 2}};


int request_resources(int process_num, int request[]) {
```

```
}

    // Check if request can be granted for (int i =
    0; i < MAX_RESOURCES; i++) {
            if (request[i] > available[i] || request[i] > maximum[process_num][i]
- allocation[process_num][i])
          return 0; // Request cannot be granted
    }

    // Try allocating resources temporarily for (int
    i = 0; i < MAX_RESOURCES; i++) {
    available[i]              -=              request[i];
    allocation[process_num][i] += request[i];
        // Update maximum and need matrix if request is granted
        maximum[process_num][i] -= request[i];
    }

    // Check if system is in safe state after allocation if
    (is_safe()) { return 1; // Request
        is granted
    } else {
        // Roll back changes if not safe for (int i = 0; i
        < MAX_RESOURCES; i++) { available[i]
        += request[i]; allocation[process_num][i] -=
        request[i]; maximum[process_num][i] +=
        request[i];
        }
        return 0; // Request is denied
    }
}

int is_safe() {
    int work[MAX_RESOURCES];
    int finish[MAX_PROCESSES] = {0};
```

```
// Initialize work array
for (int i = 0; i < MAX_RESOURCES; i++) { work[i]
    = available[i];
}

// Check if processes can finish int count = 0;
while (count < MAX_PROCESSES) { int found
= 0; for (int i = 0; i < MAX_PROCESSES; i++)
{ if
        (finish[i] == 0) { int
            j;
            for (j = 0; j < MAX_RESOURCES; j++) { if
                (maximum[i][j] - allocation[i][j] > work[j]) break;
            }
            if (j == MAX_RESOURCES) {
                // Process can finish, update work and mark as finished for (int k
                = 0; k < MAX_RESOURCES; k++) {
                    work[k] += allocation[i][k];
                } finish[i] =
                1; found =
                1; count++;
            }
        }
    }
    if (found == 0) { return 0; // No process can
    finish, not safe state }
}
return 1; // All processes can finish, safe state
}


int main() {
    int process_num, request[MAX_RESOURCES];
    printf("Enter process number (0 to 4): "); scanf("%d",
    &process_num);
```

```
    printf("Enter resource request (e.g., 0 1 0): "); for (int
    i = 0; i < MAX_RESOURCES; i++) {
        scanf("%d", &request[i]);
    }

    if (request_resources(process_num, request)) {
        printf("Request granted.\n");
    } else { printf("Request denied. System is not in safe
        state.\n");
    }
  return 0;
```

OUTPUT :



18. Construct a C program to simulate producer consumer problem using semaphores.

AIM :

To construct a C program to simulate producer consumer problem using semaphores.

ALGORITHM :

1. Include Libraries: Include necessary libraries such as stdio.h, stdlib.h, pthread.h for threads, and semaphore.h for semaphores.

2. Define Constants: Define constants such as the size of the buffer (maximum number of items), number of producer and consumer threads, etc.

3. Declare Global Variables: Declare global variables including the buffer (an array to hold the items), indices to track the next position for inserting and removing items, and semaphores for synchronization.

4. Initialize Semaphores: Initialize semaphores for controlling access to the buffer, tracking empty spaces in the buffer, and tracking available items in the buffer.

5. Create Producer and Consumer Threads: Create threads for producers and consumers. Each thread should have its own unique identifier (for example, an integer value).

6. Define Producer and Consumer Functions: Implement functions for producers and consumers. These functions will be executed by the corresponding threads. The producer function will generate items and insert them into the buffer, while the consumer function will remove items from the buffer.

7. Implement Buffer Operations: Implement functions for inserting items into the buffer (enqueue operation) and removing items from the buffer (dequeue operation). Use semaphores to control access to the buffer and update the indices accordingly.

8. Synchronize Producer and Consumer Threads: Use semaphores to synchronize the producer and consumer threads. The producer should wait if the buffer is full, and the consumer should wait if the buffer is empty.

9. Handle Thread Joining and Cleanup: After creating the threads, ensure that the main program waits for all threads to finish their execution. Use pthread_join for this purpose. Also, clean up any resources allocated during the program execution.

10. Compile and Run: Compile the C program using a C compiler (such as gcc) and run the executable. Observe the behavior of the producer and consumer threads, ensuring that they are properly synchronized and the buffer operations are correctly implemented.

PROGRAM :

```
#include <stdio.h> #include
<pthread.h> #include
<semaphore.h>
#include<Windows.h>

#define BUFFER_SIZE 5
#define MAX_ITEMS 10 // Maximum number of items to be
produced/consumed


int buffer[BUFFER_SIZE];
sem_t empty, full;
int produced_items = 0, consumed_items = 0;
```

```
void*    producer(void*    arg)    {    while
   (produced_items    <    MAX_ITEMS)    {
   sem_wait(&empty);
      // Critical section: add item to buffer for
      (int i = 0; i < BUFFER_SIZE; ++i) { if
         (buffer[i]    ==    0)    {    buffer[i]    =
            produced_items          +          1;
            printf("Produced: %d\n", buffer[i]);
            produced_items++; break;
         } }
      sem_post(&full)
      ;
      Sleep(1); // Sleep for a while
   } return
   NULL;
}


void* consumer(void* arg) {
   while (consumed_items < MAX_ITEMS) {
      sem_wait(&full);
      // Critical section: remove item from buffer for
      (int i = 0; i < BUFFER_SIZE; ++i) { if (buffer[i]
      != 0) { printf("Consumed: %d\n", buffer[i]);
      buffer[i] = 0;
            consumed_items++;
            break;
         }
      }
      sem_post(&empty); Sleep(2); // Sleep
      for a while
   } return
   NULL;
}
```

```
int main() {
    pthread_t producer_thread, consumer_thread;

    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);

    // Create producer and consumer threads pthread_create(&producer_thread,
    NULL, producer, NULL); pthread_create(&consumer_thread, NULL,
    consumer, NULL);

    // Wait for threads to finish
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread,
    NULL);

    // Destroy semaphores
    sem_destroy(&empty);
    sem_destroy(&full);

    return 0;
}
```

OUTPUT :

C:\Users\itssk\OneDrive\Desk

```
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
Produced: 5
Consumed: 5
Produced: 6
Consumed: 6
Produced: 7
Consumed: 7
Produced: 8
Consumed: 8
Produced: 9
Consumed: 9
Produced: 10
Consumed: 10
```

19. esign a C program to implement process synchronization using mutex locks.

AIM:

To design a C program to implement process synchronization using mutex locks.

ALGORITHM :

Step 1: Include Necessary Libraries: Include the required header files for pthreads and mutex locks.

Step 2: Declare Global Variables: Declare any global variables needed for synchronization, such as mutex variables.

Step 3: Initialize Mutex: In the main function or initialization function, initialize the mutex using pthread_mutex_init function.

Step 4: Define Functions: Define functions that represent the actions of threads. These functions should include the critical sections where the mutex lock is acquired and released.

Step 5: Create Threads: In the main function or any other appropriate function, create threads and assign the functions to execute for each thread. Pass NULL or any necessary data as arguments to the functions.

Step 6: Implement Mutex Synchronization: Inside the functions that represent the actions of threads, use pthread_mutex_lock to acquire the mutex lock and pthread_mutex_unlock to release the lock. This ensures that only one thread can execute the critical section at a time.

Step 7: Join Threads and Cleanup: In the main function or any other appropriate function, wait for the threads to finish using pthread_join. After the threads have finished their execution, destroy the mutex using pthread_mutex_destroy function.

Step 8: Compile and Run: Compile the program using a C compiler with the appropriate flags (for example, -pthread for GCC) to link the pthread library. Then, run the compiled executable to observe the synchronized behavior of threads due to mutex locks.

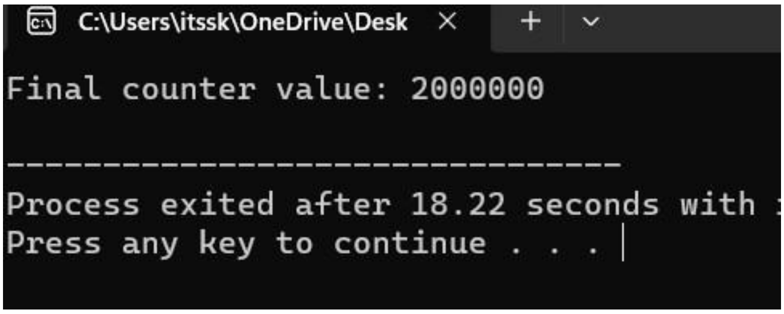
PROGRAM :

```
#include <stdio.h>

#include <pthread.h>


// Shared variables int

counter = 0;

pthread_mutex_t mutex;
```

```
// Function to be executed by threads void
*threadFunction(void *arg) {
    int i;
    for (i = 0; i < 1000000; ++i) { } return
    NULL;
}

int main() {
```

```
pthread_mutex_init(&mutex, NULL);
 pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, threadFunction, NULL);
  pthread_create(&thread2, NULL, threadFunction, NULL);

  // Wait for the threads to finish pthread_join(thread1,
       NULL); pthread_join(thread2, NULL);

  // Destroy the mutex
  pthread_mutex_destroy(&mutex);

  // Print the final value of the counter printf("Final
  counter value: %d\n", counter);


  return 0;
}
```

OUTPUT :



20. Construct a C program to simulate Reader-Writer problem using semaphores

AIM :

To construct a C program to simulate Reader-Writer problem using semaphores

ALGORITHM :

1. Include Libraries: Include necessary libraries for using semaphores, threads, and other required functionalities.

2. Initialize Semaphores: Create semaphores to control access to the shared resources:

- Semaphore for Readers Count: Initialize a semaphore to 1 (binary semaphore).

- Semaphore for Writers Count: Initialize a semaphore to 1 (binary semaphore).

- Semaphore for Readers Waiting: Initialize a semaphore to 1 (binary semaphore).

- Semaphore for Writers Waiting: Initialize a semaphore to 1 (binary semaphore).

- Semaphore for Mutex: Initialize a semaphore to 1 (binary semaphore).

3. Reader Function: Create a function for readers to execute. This function should handle the logic for readers accessing the shared resource.

4. Writer Function: Create a function for writers to execute. This function should handle the logic for writers accessing the shared resource.

5. Implement Reader-Writer Logic: Inside the reader and writer functions, implement the logic that ensures proper synchronization using semaphores. Readers should check and update the readers count semaphore and writers should check and update the writers count semaphore.

6. Create Threads: In your main function, create multiple threads for readers and writers to simulate concurrent access.

7. Join Threads: Use thread joining functions to wait for all threads to complete their execution.

8. Clean Up: Destroy the semaphores and perform any necessary clean-up operations before exiting the program.

PROGRAM :

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

<pthread.h> #include

<semaphore.h>
```

```
sem_t mutex, writeBlock;
int data = 0, readersCount = 0;

void *reader(void *arg) { int
          i=0;
    while (i<10) {
        sem_wait(&mutex);
        readersCount++; if
        (readersCount == 1) {
            sem_wait(&writeBlock);
        }
        sem_post(&mutex);

        // Reading operation
        printf("Reader reads data: %d\n", data);

        sem_wait(&mutex);
        readersCount--; if
        (readersCount == 0) {
            sem_post(&writeBlock);
        } sem_post(&mutex);
        i++;

    }
}


void *writer(void *arg) { int
          i=0;
    while (i<10) {
        sem_wait(&writeBlock);

        //      Writing      operation      data++;
        printf("Writer writes data: %d\n", data);
```

```
        sem_post(&writeBlock); i++;
    }
}


int main() {
    pthread_t readers, writers; sem_init(&mutex,
    0, 1);
    sem_init(&writeBlock, 0, 1); pthread_create(&readers,
   NULL, reader, NULL); pthread_create(&writers, NULL,
   writer, NULL); pthread_join(readers, NULL);
   pthread_join(writers, NULL); sem_destroy(&mutex);
   sem_destroy(&writeBlock);
    return 0;
}
```

OUTPUT :

```
Writer writes data: 1
Reader reads data: 1
Writer writes data: 2
Reader reads data: 2
Writer writes data: 3
Reader reads data: 3
Writer writes data: 4
Reader reads data: 4
Writer writes data: 5
Reader reads data: 5
Writer writes data: 6
Reader reads data: 6
Writer writes data: 7
Reader reads data: 7
Writer writes data: 8
Reader reads data: 8
Writer writes data: 9
Reader reads data: 9
Writer writes data: 10
Reader reads data: 10

--------------------------------
Process exited after 12.44 seconds with
```

21. Develop a C program to implement worst fit algorithm of memory management.

PROGRAM:

```
#include <stdio.h>

#define MAX_MEMORY 1000 int

memory[MAX_MEMORY];

// Function to initialize memory void
initializeMemory() {
    for (int i = 0; i < MAX_MEMORY; i++) { memory[i] = -1; // -1
        indicates that the memory is unallocated
    }
}

// Function to display memory status void
displayMemory() {
    int i, j;
    int count = 0; printf("Memory
    Status:\n");

    for (i = 0; i < MAX_MEMORY; i++) {
        if (memory[i] == -1) {
            count++;
            j = i;
            while (memory[j] == -1 && j < MAX_MEMORY) { j++;
            } printf("Free memory block %d-%d\n", i, j - 1); i =
            j - 1;
        }
    }
```

```
    if (count == 0) { printf("No free memory
        available.\n");
    }
}

// Function to allocate memory using worst-fit algorithm
void allocateMemory(int processId, int size) { int start = -1;
int blockSize = 0;

    for (int i = 0; i < MAX_MEMORY; i++) { if
        (memory[i] == -1) { if
            (blockSize == 0) {
            start = i; }
            blockSize++;
        } else { blockSize
            = 0;
        }

        if (blockSize >= size) {
            break;
        }
    }

    if (blockSize >= size) { for (int i = start; i
        < start + size; i++) { memory[i] =
        processId;
        }
        printf("Allocated memory block %d-%d to Process %d\n", start, start + size - 1,
processId);
    } else { printf("Memory allocation for Process %d failed (not enough
        contiguous
memory).\n", processId);
    }
}

// Function to deallocate memory void
deallocateMemory(int processId) { for (int i =
0; i < MAX_MEMORY; i++) { if
        (memory[i] == processId) {
            memory[i] = -1;
        }
    }
    printf("Memory released by Process %d\n", processId);
}

int main() {
    initializeMemory();
    displayMemory();
```

```
    allocateMemory(1, 200);
    displayMemory();

    allocateMemory(2, 300);
    displayMemory();

    deallocateMemory(1);
    displayMemory();

    allocateMemory(3, 400);
    displayMemory();

    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_
```

22. Construct a C program to implement best fit algorithm of memory management.

PROGRAM:

```
#include <stdio.h>

#define MAX_MEMORY 1000 int

memory[MAX_MEMORY];

// Function to initialize memory void
initializeMemory() {
    for (int i = 0; i < MAX_MEMORY; i++) { memory[i] = -1; // -1
        indicates that the memory is unallocated
    }
}


// Function to display memory status void
displayMemory() {
    int i, j;
    int count = 0; printf("Memory
    Status:\n");

    for (i = 0; i < MAX_MEMORY; i++) {
        if (memory[i] == -1) {
            count++;
            j = i;
            while (memory[j] == -1 && j < MAX_MEMORY) { j++;
            }
            printf("Free memory block %d-%d\n", i, j - 1); i = j -
            1;
        }
    }
```

```
    if (count == 0) { printf("No free memory
        available.\n");
    }
}

// Function to allocate memory using best-fit algorithm
void allocateMemory(int processId, int size) { int start = -
1;
    int blockSize = MAX_MEMORY; int
    bestStart = -1;
    int bestSize = MAX_MEMORY;

    for (int i = 0; i < MAX_MEMORY; i++) { if
        (memory[i] == -1) { if (blockSize ==
            MAX_MEMORY) { start = i;
            }
            blockSize++;
        } else { if (blockSize >= size && blockSize < bestSize) {
            bestSize
                = blockSize; bestStart
                = start;
            }
            blockSize = 0;
        }
    }

    if (bestSize >= size) { for (int i = bestStart; i <
        bestStart + size; i++) { memory[i] = processId;
        }
        printf("Allocated memory block %d-%d to Process %d\n", bestStart, bestStart +
size - 1, processId);
    } else { printf("Memory allocation for Process %d failed (not enough
        contiguous
memory).\n", processId);
    }
}

// Function to deallocate memory void
deallocateMemory(int processId) { for (int i =
0; i < MAX_MEMORY; i++) { if
        (memory[i] == processId) {
            memory[i] = -1;
        }
    }
    printf("Memory released by Process %d\n", processId);
}
```

```
int main() {
    initializeMemory();
    displayMemory();

    allocateMemory(1, 200);
    displayMemory();

    allocateMemory(2, 300);
    displayMemory();

    deallocateMemory(1);
    displayMemory();

    allocateMemory(3, 400);
    displayMemory();


    return 0;
}
```

OUTPUT:

```
Memory Status:
Free memory block 0-999
Allocated memory block -1-198 to Process 1
Memory Status:
Free memory block 199-999
Allocated memory block -1-298 to Process 2
Memory Status:
Free memory block 299-999
Memory released by Process 1
Memory Status:
Free memory block 299-999
Allocated memory block -1-398 to Process 3
Memory Status:
Free memory block 399-999

--------------------------------
Process exited after 0.06954 seconds with return value 0
Press any key to continue . . .
```

23. Construct a C program to implement first fit algorithm of memory management.

PROGRAM:

```
#include <stdio.h>

#define MAX_MEMORY 1000 int

memory[MAX_MEMORY];

// Function to initialize memory void
initializeMemory() {
    for (int i = 0; i < MAX_MEMORY; i++) { memory[i] = -1; // -1
    indicates that the memory is unallocated }
}


// Function to display memory status void
displayMemory() {
    int i, j;
    int   count   =   0;   printf("Memory
    Status:\n");   for   (i   =   0;   i   <
    MAX_MEMORY;     i++)     {     if
    (memory[i] == -1) {
            count++;
            j = i;
           while (memory[j] == -1 && j < MAX_MEMORY) { j++;
            } printf("Free memory block %d-%d\n", i, j - 1); i =
            j - 1;
        }
    }
```

```
    if (count == 0) { printf("No free memory
        available.\n");
    }
}

// Function to allocate memory using first-fit algorithm
void allocateMemory(int processId, int size) { int start = -
1; int blockSize = 0;

    for (int i = 0; i < MAX_MEMORY; i++) { if
        (memory[i] == -1) { if
            (blockSize == 0) {
            start = i; }
            blockSize++;
        } else {
            blockSize = 0;
        }

        if (blockSize >= size) {
            break;
        }
    }

    if (blockSize >= size) { for (int i = start; i
        < start + size; i++) { memory[i] =
        processId;
        }
        printf("Allocated memory block %d-%d to Process %d\n", start, start + size - 1,
processId);
    } else { printf("Memory allocation for Process %d failed (not enough
        contiguous
memory).\n", processId);
    }
}

// Function to deallocate memory void
deallocateMemory(int processId) { for (int i =
0; i < MAX_MEMORY; i++) { if
        (memory[i] == processId) {
            memory[i] = -1;
        }
    }
    printf("Memory released by Process %d\n", processId);
}

int main() {
    initializeMemory();
    displayMemory();
```

```
    allocateMemory(1, 200);
    displayMemory();

    allocateMemory(2, 300);
    displayMemory();

    deallocateMemory(1);
    displayMemory();

    allocateMemory(3, 400);
    displayMemory();


    return 0;
}
```

OUTPUT:

```
Memory Status:
Free memory block 0-999
Allocated memory block 0-199 to Process 1
Memory Status:
Free memory block 200-999
Allocated memory block 200-499 to Process 2
Memory Status:
Free memory block 500-999
Memory released by Process 1
Memory Status:
Free memory block 0-199
Free memory block 500-999
Allocated memory block 500-899 to Process 3
Memory Status:
Free memory block 0-199
Free memory block 900-999

--------------------------------
Process exited after 0.01792 seconds with return value 0
Press any key to continue . . .
```

24. Design a C program to demonstrate UNIX system calls for file management.

PROGRAM:

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

int main() {
    int fd;
    char buffer[100];

    // Creating a new file
    fd = creat("sample.txt", S_IRWXU); if (fd
    == -1) {
        perror("create");
        exit(1);
    } else { printf("File 'sample.txt' created successfully.\n");
        close(fd);
    }

    // Opening an existing file for writing
    fd = open("sample.txt", O_WRONLY | O_APPEND); if (fd ==
    -1) {
        perror("open");
        exit(1);
    } else { printf("File 'sample.txt' opened for
        writing.\n");
    }
```

```
// Writing data to the file write(fd,
"Hello, World!\n", 14);
printf("Data written to 'sample.txt'.\n");
close(fd);

// Opening the file for reading
fd = open("sample.txt", O_RDONLY); if (fd
== -1) {
    perror("open");
    exit(1);
} else { printf("File 'sample.txt' opened for
reading.\n"); }

// Reading data from the file
int bytesRead = read(fd, buffer, sizeof(buffer)); if
(bytesRead == -1) {
    perror("read");
    exit(1);
} else { printf("Data read from 'sample.txt':\n"); write(STDOUT_FILENO,
    buffer, bytesRead);
}
close(fd);

// Deleting the file if
(remove("sample.txt") == -1) {
perror("remove");
    exit(1);
} else { printf("File 'sample.txt'
deleted.\n"); }
```

```
    return 0;
}
```

OUTPUT:

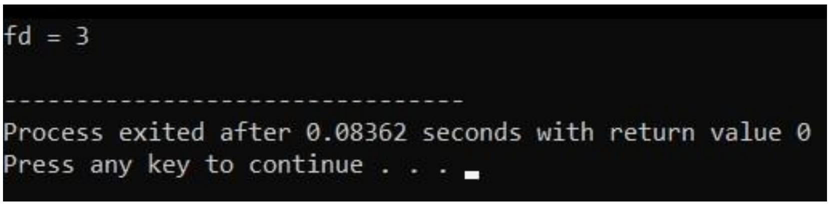
```
File 'sample.txt' created successfully.
File 'sample.txt' opened for writing.
Data written to 'sample.txt'.
File 'sample.txt' opened for reading.
Data read from 'sample.txt':
Hello, World!
File 'sample.txt' deleted.

--------------------------------
Process exited after 0.02066 seconds with return value 0
Press any key to continue . . .
```

25) Construct a C program to implement the I/O system calls of UNIX (fcntl, seek, stat, opendir, readdir)

PROGRAM:

```
#include<stdio.h>
#include<fcntl.h>
#include<errno.h>
extern int errno; int
main()
{

        int fd = open("foo.txt", O_RDONLY | O_CREAT);
        printf("fd = %d\n", fd);
        if (fd ==-1)
        { printf("Error Number % d\n", errno);
                perror("Program");
        } return
        0;
}
```

OUTPUT:



26) Construct a C program to implement the file management operations.

PROGRAM:

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

```
#include <stdlib.h> int main()
{
     FILE *file; file =
   fopen("example.txt", "w");i f
   (file == NULL) { printf("Error opening the file for
   writing.\n"); return 1; }
   fprintf(file, "Hello, World!\n");
   fprintf(file, "This is a C file management example.\n"); fclose(file);
   file = fopen("example.txt", "r"); if
   (file == NULL) { printf("Error opening the file for
   reading.\n"); return 1; } char buffer[100];
   while (fgets(buffer, sizeof(buffer), file) != NULL) {
      printf("%s", buffer);
   }
   fclose(file);

   return 0;
}
```

OUTPUT:

```
Hello, World!
This is a C file management example.

--------------------------------
Process exited after 0.1135 seconds with return value 0
Press any key to continue . . .
```

27) Develop a C program for simulating the function of ls UNIX Command.

PROGRAM:

```
 #include<stdio.h>
 #include<dirent.h>
   int main()
       { char fn[10], pat[10], temp[200];
     FILE
     *fp;
     printf("\n Enter file name : ");
     scanf("%s", fn); printf("Enter the
     pattern: "); scanf("%s", pat); fp =
     fopen(fn, "r"); while (!feof(fp)) {
     fgets(temp, sizeof(fp), fp);i f
```

```
    (strcmp(temp, pat)) printf("%s",
    temp);
    }
    fclose(fp);
    return 1;

}
```

OUTPUT:

```
This is a sample line.
Hello, World!
Sample pattern in this line.
Another sample line.
```

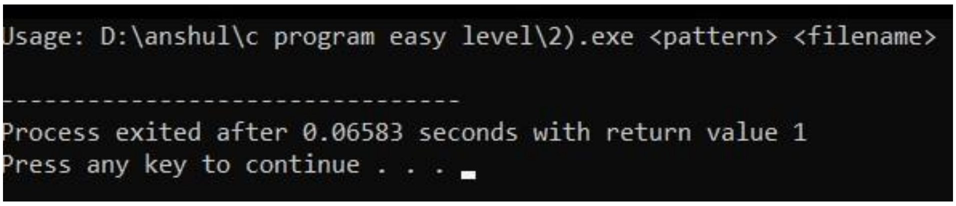
28) Write a C program for simulation of GREP UNIX command.

PROGRAM :

```
#include <stdio.h> #include
<stdlib.h>
#include <string.h>
#define MAX_LINE_LENGTH 1024
void searchFile(const char *pattern, const char *filename)
{
FILE *file = fopen(filename, "r"); if
  (file == NULL) { perror("Error
  opening file"); exit(1);
  }
  char line[MAX_LINE_LENGTH]; while
  (fgets(line, sizeof(line), file)) { if
    (strstr(line, pattern) != NULL) {
    printf("%s", line);
    }
  }
  fclose(file);
}
```

```
int main(int argc, char *argv[]) { if
    (argc != 3) { fprintf(stderr, "Usage: %s <pattern> <filename>\n",
        argv[0]); return 1;
    } const char *pattern = argv[1];
    const char *filename = argv[2];
    searchFile(pattern,         filename);
    return 0;
}
```

OUTPUT:



29) Write a C program to simulate the solution of Classical Process Synchronization Problem

PROGRAM:

```
#include      <stdio.h>
#include <stdlib.h> int mutex
= 1;
int full = 0; int empty
= 10, x = 0; void
producer()
{
    --mutex;
    ++full; --
    empty; x++;
    printf("\nProducer
        produces" "item %d",
        x);
    ++mutex;
}
void consumer()
{
```

```
    --mutex;
    --full;
    ++empty;
    printf("\nConsumer consumes " "item
          %d",
         x);
    x--;
    ++mutex;
} int
main()
{ int n, i;
    printf("\n1. Press 1 for Producer"
         "\n2. Press 2 for Consumer" "\n3.
         Press 3 for Exit");
#pragma omp critical for (i
    = 1; i > 0; i++)
          { printf("\nEnter your
        choice:"); scanf("%d", &n);
        switch (n) { case 1:
            if ((mutex == 1) &&
                (empty != 0)) {
                producer();
            }
            else
                                { printf("Buffer
                is full!");
            } break;
        case 2:
            if ((mutex == 1) &&
                (full != 0)) {
                consumer();
            } else { printf("Buffer is
            empty!");
            }
            break;

        case 3: exit(0);
            break;
        }
    }
}
```

OUTPUT:

```
1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 for Exit
Enter your choice:1

Producer producesitem 1
Enter your choice:1

Producer producesitem 2
Enter your choice:1

Producer producesitem 3
Enter your choice:2

Consumer consumes item 3
Enter your choice:3

--------------------------------
Process exited after 6.797 seconds with return value 0
Press any key to continue . . .
```

30. Write C programs to demonstrate the following thread related concepts.

PROGRAM:

```
#include <pthread.h>
#include     <stdio.h>
#include    <stdlib.h>
void* func(void* arg)
{ pthread_detach(pthread_self()); printf("Inside the
        thread\n"); pthread_exit(NULL);
} void
fun()
{ pthread_t ptid;
        pthread_create(&ptid, NULL, &func, NULL);
        printf("This line may be printed"
                " before thread terminates\n");
        if(pthread_equal(ptid, pthread_self()))
        { printf("Threads are equal\n");
        }

        else printf("Threads are not equal\n");
        pthread_join(ptid, NULL);
        printf("This line will be printed" "
        after thread ends\n");
        pthread_exit(NULL);
}

int main()
{ fun(); return
        0;
}
```

OUTPUT:

```
This line may be printed before thread terminates
Inside the thread
Threads are not equal
This line will be printed after thread ends
```

31. Construct a C program to simulate the First in First Out paging technique of memory management.

AIM: Construct a C program to simulate the First in First Out paging technique of memory management.

ALGORITHM:

1. Create an array to represent the page frames in memory.
2. Initialize all page frames to -1, indicating that they are empty.
3. Initialize a queue to keep track of the order in which pages are loaded into memory.
4. Initialize variables for page hits and page faults to zero.
5. Read the reference string (sequence of page numbers) from the user or use a predefined array.
6. For each page in the reference string, do the following:
7. Check if the page is already in memory (a page hit).
   If it's a page hit, update the display and move to the next page.
8. If it's a page fault (page not in memory), do the following: Increment the page fault count.
9. Remove the oldest page in memory (the one at the front of the queue).
10. .Load the new page into the memory and enqueue it.
11. Update the display to show the page replacement.
12. Continue this process for all pages in the reference string.
13. After processing all pages, display the total number of page faults.

PROGRAM:

```
#include <stdio.h>

#define MAX_FRAMES 3 // Maximum number of frames in memory

void printFrames(int frames[], int n) { for
   (int i = 0; i < n; i++) {
      if (frames[i] == -1) {
      printf(" - ");
      } else { printf(" %d ",
         frames[i]);
      } }
   printf("\n")
   ;
}
```

```
int main() { int referenceString[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3,
    2}; int n = sizeof(referenceString) / sizeof(referenceString[0]);
    int frames[MAX_FRAMES];
    int framePointer = 0; // Points to the current frame to be replaced

    for (int i = 0; i < MAX_FRAMES; i++)
    { frames[i] = -1; // Initialize all frames to -1 (indicating empty)
    }

    printf("Reference String: "); for
    (int i = 0; i < n; i++) { printf("%d ",
        referenceString[i]);
    }printf("\n\n");
    printf("Page Replacement
    Order:\n"); for (int i = 0; i < n; i++) {
    int page = referenceString[i]; int
    pageFound = 0;
        // Check if the page is already in memory for (int j
        = 0; j < MAX_FRAMES; j++) { if
            (frames[j] == page) {
                pageFound = 1; break;
            }
        }


        if (!pageFound) {
            printf("Page %d -> ", page); frames[framePointer] =
            page; framePointer = (framePointer + 1) %
            MAX_FRAMES; printFrames(frames,
            MAX_FRAMES);
        }
    }


    return 0;
}
```

OUTPUT:

```
Reference String: 7 0 1 2 0 3 0 4 2 3 0 3 2

Page Replacement Order:
Page 7 ->  7  -  -
Page 0 ->  7  0  -
Page 1 ->  7  0  1
Page 2 ->  2  0  1
Page 3 ->  2  3  1
Page 0 ->  2  3  0
Page 4 ->  4  3  0
Page 2 ->  4  2  0
Page 3 ->  4  2  3
Page 0 ->  0  2  3

--------------------------------
Process exited after 0.0623 seconds with return value 0
Press any key to continue . . .
```

32. Construct a C program to simulate the Least Recently Used paging technique of memory management.

AIM: Construct a C program to simulate the Least Recently Used paging technique of memory management.

ALGORITHM:

1. Create an array to represent the page frames in memory.
2. Initialize all page frames to -1, indicating that they are empty.
3. Create a queue or a data structure (e.g., a doubly-linked list) to maintain the order of pages based on their usage history.
4. Initialize a counter for page hits and page faults to zero.
5. Read the reference string (sequence of page numbers) from the user or use a predefined array.
6. For each page in the reference string, do the following:
7. Check if the page is already in memory (a page hit).
8. If it's a page hit, update the position of the page in the usage history data structure to indicate it was recently used.
9. If it's a page fault (page not in memory), do the following:
10. Increment the page fault count.
11. Find the least recently used page in the usage history data structure (e.g., the front of the queue or the tail of the list).
12. Remove the least recently used page from memory and the usage history.
13. Load the new page into memory and add it to the back of the usage history. 14. Update the display to show the page replacement. 15.Continue this process for all pages in the reference string.
15. After processing all pages, display the total number of page faults.

PROGRAM:

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

#define MAX_FRAMES 3

void printFrames(int frames[], int n) { for
   (int i = 0; i < n; i++) {
      if (frames[i] == -1) {
      printf(" - ");
      } else { printf(" %d ",
         frames[i]);
      }
   }
```

```
    printf("\n");
}

int main() {
    int frames[MAX_FRAMES]; int usageHistory[MAX_FRAMES]; // To store
    the usage history of pages for (int i = 0; i < MAX_FRAMES; i++) { frames[i]
    = -1; // Initialize frames to -1 (empty) usageHistory[i] = 0; // Initialize usage
    history
    }


    int pageFaults = 0; int referenceString[] = {7, 0, 1, 2, 0, 3, 0, 4,
    2, 3, 0, 3, 2}; int n = sizeof(referenceString) /
    sizeof(referenceString[0]);

    printf("Reference String: "); for
    (int i = 0; i < n; i++) { printf("%d ",
        referenceString[i]);
    } printf("\n\n"); printf("Page

    Replacement Order:\n"); for


    (int i = 0; i < n; i++) { int page =
        referenceString[i]; int
        pageFound = 0;

        // Check if the page is already in memory (a page hit) for (int j
        = 0; j < MAX_FRAMES; j++) {
            if (frames[j] == page) {
            pageFound = 1;
                // Update the usage history by incrementing other pages for (int k
                = 0; k < MAX_FRAMES; k++) {
                    if (k != j) {
                        usageHistory[k]++;
                    }
                }
            usageHistory[j] = 0; // Reset the usage counter for the used page break;
            }
        }


        if (!pageFound) {
            printf("Page %d -> ", page);


            // Find the page with the maximum usage counter (least recently
used)  int lruPage = 0;
```

```
            for (int j = 1; j < MAX_FRAMES; j++) { if
                (usageHistory[j] > usageHistory[lruPage]) { lruPage
                = j;
                }
            }

            int replacedPage = frames[lruPage];
            frames[lruPage] = page;
            usageHistory[lruPage] = 0;

            if (replacedPage != -1) { printf("Replace %d with %d:
                ", replacedPage, page);
            } else { printf("Load into an empty
                frame: ");
            }

            printFrames(frames, MAX_FRAMES);
            pageFaults++;
        }
    }

    printf("\nTotal Page Faults: %d\n", pageFaults);

    return 0;
}
```

OUTPUT:

```
Reference String: 7 0 1 2 0 3 0 4 2 3 0 3 2

Page Replacement Order:
Page 7 -> Load into an empty frame:  7  -  -
Page 0 -> Replace 7 with 0:  0  -  -
Page 1 -> Replace 0 with 1:  1  -  -
Page 2 -> Replace 1 with 2:  2  -  -
Page 0 -> Replace 2 with 0:  0  -  -
Page 3 -> Replace 0 with 3:  3  -  -
Page 0 -> Replace 3 with 0:  0  -  -
Page 4 -> Replace 0 with 4:  4  -  -
Page 2 -> Replace 4 with 2:  2  -  -
Page 3 -> Replace 2 with 3:  3  -  -
Page 0 -> Replace 3 with 0:  0  -  -
Page 3 -> Replace 0 with 3:  3  -  -
Page 2 -> Replace 3 with 2:  2  -  -

Total Page Faults: 13

--------------------------------
Process exited after 0.05045 seconds with return value 0
Press any key to continue . . .
```

33. Construct a C program to simulate the optimal paging technique of memory management

AIM: Construct a C program to simulate the optimal paging technique of memory management

ALGORITHM:

1. Create an array to represent the page frames in memory.
2. Initialize all page frames to -1, indicating that they are empty.
3. Initialize a variable for page faults to zero.
4. Read the reference string (sequence of page numbers) from the user or use a predefined array.
5. For each page in the reference string, do the following:
6. Check if the page is already in memory (a page hit).
7. If it's a page hit, move to the next page.
8. If it's a page fault (page not in memory), do the following:
9. Increment the page fault count.
10. Calculate the future references of each page in memory by scanning the remaining part of the reference string.
11. Find the page that will not be used for the longest time in the future (the optimal page to replace).
12. Replace the optimal page with the new page. 13.Continue this process for all pages in the reference string.
13. After processing all pages, display the total number of page faults.

PROGRAM:

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

#define MAX_FRAMES 3

void printFrames(int frames[], int n) { for
   (int i = 0; i < n; i++) {
      if (frames[i] == -1) {
      printf(" - ");
      } else { printf(" %d ",
         frames[i]);
      }
   }
   printf("\n");
}
```

```
int main() {
    int frames[MAX_FRAMES];
    for (int i = 0; i < MAX_FRAMES; i++) { frames[i] = -1;
        // Initialize frames to -1 (empty)
    }

    int pageFaults = 0; int referenceString[] = {7, 0, 1, 2, 0, 3, 0, 4,
    2, 3, 0, 3, 2}; int n = sizeof(referenceString) /
    sizeof(referenceString[0]);

    printf("Reference String: "); for
    (int i = 0; i < n; i++) { printf("%d ",
        referenceString[i]);
    } printf("\n\n"); printf("Page

    Replacement Order:\n"); for


    (int i = 0; i < n; i++) { int page
        = referenceString[i];
        int pageFound = 0;

        // Check if the page is already in memory (a page hit) for (int j
        = 0; j < MAX_FRAMES; j++) {
            if (frames[j] == page) {
                pageFound = 1; break;
            }
        }


        if (!pageFound) { printf("Page
            %d -> ", page);

            int optimalPage = -1; int
            farthestDistance = 0;

            for (int j = 0; j < MAX_FRAMES; j++) { int
                futureDistance = 0; for (int k = i + 1; k < n;
                k++) { if (referenceString[k] == frames[j]) {
                break;
                    }
                    futureDistance++;
                }


                if (futureDistance > farthestDistance) {
                    farthestDistance = futureDistance;
                    optimalPage = j;
                }
```

```
            }

            frames[optimalPage] = page;

            printFrames(frames, MAX_FRAMES);
            pageFaults++;
        }
    }

    printf("\nTotal Page Faults: %d\n", pageFaults);

    return 0;
}
```

OUTPUT

```
Reference String: 7 0 1 2 0 3 0 4 2 3 0 3 2

Page Replacement Order:
Page 7 ->  7  -  -
Page 0 ->  0  -  -
Page 1 ->  0  1  -
Page 2 ->  0  2  -
Page 3 ->  0  2  3
Page 4 ->  4  2  3
Page 0 ->  0  2  3

Total Page Faults: 7

--------------------------------
Process exited after 0.05286 seconds with return value 0
Press any key to continue . . .
```

34. Consider a file system where the records of the file are stored one after another both physically and logically. A record of the file can only be accessed by reading all the previous records. Design a C program to simulate the file allocation strategy.

AIM: Consider a file system where the records of the file are stored one after another both physically and logically. A record of the file can only be accessed by reading all the previous records. Design a C program to simulate the file allocation strategy.

ALGORITHM:

1. Define the structure of a record that will be stored in the file.
2. Create a file to represent the sequential file.
3. Write records to the file sequentially, one after the other.
4. To read a specific record:
5. Prompt the user for the record number they want to access.
6. Read and display all records from the beginning of the file up to the requested record.
7. Continue this process until the user decides to exit.

PROGRAM:

```
#include <stdio.h>

#include <stdlib.h>


// Structure to represent a record struct

Record { int

    recordNumber;

    char data[256]; // Adjust the size as needed for your records

};


int main() { FILE *file;

    struct Record record;

    int recordNumber;


    // Open or create a file in write mode (for writing records) file

    = fopen("sequential_file.txt", "w"); if (file == NULL) {

    printf("Error opening the file.\n");

        return 1;

    }


    // Write records sequentially to the file

    printf("Enter records (Enter '0' as record number to exit):\n"); while (1)
```

```
{ printf("Record Number: "); scanf("%d",
    &record.recordNumber); if
    (record.recordNumber == 0) {
        break;
    }

    // Input data for the record
    printf("Data: ");
    scanf(" %[^\n]", record.data);


    // Write the record to the file
    fwrite(&record, sizeof(struct Record), 1, file);
}

fclose(file);

// Reopen the file in read mode (for reading records) file
= fopen("sequential_file.txt", "r"); if (file == NULL) {
printf("Error opening the file.\n"); return 1;
}

// Read a specific record from the filew hile
(1) { printf("Enter the record number to read (0 to exit):
    "); scanf("%d", &recordNumber); if (recordNumber
    == 0) {
        break;
    }

    // Read and display records up to the requested record while
    (fread(&record, sizeof(struct Record), 1, file)) {
        printf("Record Number: %d\n", record.recordNumber);
        printf("Data: %s\n", record.data); if
        (record.recordNumber == recordNumber) { break; }
    }
```

```
        rewind(file); // Reset the file pointer to the beginning of the file
    }

    fclose(file);
    return 0;
}
```

OUTPUT:

```
Enter records (Enter '0' as record number to exit):
Record Number: 389
Data: JASWANTH
Record Number: 0
Enter the record number to read (0 to exit): 389
Record Number: 389
Data: JASWANTH
Enter the record number to read (0 to exit): 0

--------------------------------
Process exited after 29.88 seconds with return value 0
Press any key to continue . . .
```

35. Consider a file system that brings all the file pointers together into an index block. The ith entry in the index block points to the ith block of the file. Design a C program to simulate the file allocation strategy.

AIM: Consider a file system that brings all the file pointers together into an index block. The ith entry in the index block points to the ith block of the file. Design a C program to simulate the file allocation strategy.

ALGORITHM:

1. Define the structure of a block that will be stored in the file.
2. Create a file to represent the indexed file.
3. Initialize an index block that contains pointers to data blocks.
4. To write a new block:
5. Prompt the user for the block number and the data to be written to the block.
6. Update the corresponding entry in the index block to point to the new data block.
7. Write the data block to the file.
8. To read a specific block:
9. Prompt the user for the block number they want to access.
10. Use the index block to find the pointer to the requested data block. 11.Read and display the data in the requested data block.

12.Continue this process until the user decides to exit.

PROGRAM:

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


// Structure to represent a block
struct      Block      {      int
blockNumber;
    char data[256]; // Adjust the size as needed for your blocks
};


int main() { FILE *file;
    struct   Block   block;
    int blockNumber;

    // Create an index block that contains pointers to data blocks int
    indexBlock[100] = {0}; // Adjust the size as needed

    // Open or create a file in write mode (for writing blocks) file =
```

```
fopen("indexed_file.txt", "w"); if
(file == NULL) { printf("Error
opening the file.\n"); return 1;
}

// Write blocks and update the index block
printf("Enter blocks (Enter '0' as block number to exit):\n"); while (1)
{ printf("Block Number: "); scanf("%d",
    &block.blockNumber);i f
    (block.blockNumber == 0) {
        break;
    }

    // Input data for the block
    printf("Data: ");
    scanf(" %[^\n]", block.data);

    // Write the block to the file
    fwrite(&block, sizeof(struct Block), 1, file);


    // Update the index block with the pointer to the data block indexBlock[block.blockNumber] =
    ftell(file) - sizeof(struct Block);
}


fclose(file);

// Reopen the file in read mode (for reading blocks) file
= fopen("indexed_file.txt", "r"); if (file == NULL) {
printf("Error opening the file.\n"); return 1;
}

// Read a specific block from the file while
```

```
    (1) { printf("Enter the block number to read (0 to exit):
        "); scanf("%d", &blockNumber); if (blockNumber
        == 0) {
            break;
        }

        if (indexBlock[blockNumber] == 0) {
            printf("Block not found.\n");
        } else {
            // Seek to the data block using the index block fseek(file,
            indexBlock[blockNumber], SEEK_SET); fread(&block,
            sizeof(struct Block), 1, file);


            printf("Block Number: %d\n", block.blockNumber);
            printf("Data: %s\n", block.data);
        }
    }


    fclose(file);
    return 0;
}
```

OUTPUT:

```
Enter blocks (Enter '0' as block number to exit):
Block Number: 39
Data: JSAWNTH
Block Number: 43
Data: SAI
Block Number: 12
Data: FRIEND
Block Number: 0
Enter the block number to read (0 to exit): 12
Block Number: 12
Data: FRIEND
Enter the block number to read (0 to exit): 0

--------------------------------
Process exited after 34.15 seconds with return value 0
Press any key to continue . . .
```

36. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block. Design a C program to simulate the file allocation strategy.

AIM: With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block. Design a C program to simulate the file allocation strategy.

ALGORITHM:

1. Define the structure of a block that will be stored in the file. Each block contains a pointer to the next block.
2. Create a file to represent the linked allocation.
3. Create a directory entry for the file containing a pointer to the first and last blocks.
4. To write a new block:
5. Prompt the user for the block data.
6. Allocate a new block in the file.
7. If it's the first block, update the directory entry to point to it as both the first and last block.
8. If it's not the first block, update the previous block to point to the new block.
9. Update the new block's pointer to the next block (usually NULL for the last block).
10. To read a specific block:
11. Prompt the user for the block number they want to access. 12.Use the directory entry to find the first block of the file.

13.Traverse the linked list of blocks until you reach the desired block. 14.Read and display the data in the requested block.

15. Continue this process until the user decides to exit.

PROGRAM:

```
#include <stdio.h>

#include <stdlib.h>

// Structure to represent a block struct Block { char data[256]; //

Adjust the size as needed for your blocks struct Block* next;

};


int main() { struct Block* firstBlock = NULL; // Pointer to the first block in the

    linked list struct Block* lastBlock = NULL; // Pointer to the last block in the

    linked list


    int blockCount = 0; // Count of blocks in the linked list
```

```
int blockNumber;
char data[256]; char
choice;

printf("Linked Allocation Simulation\n");

while (1) { printf("Enter 'W' to write a block, 'R' to read a block, or 'Q' to quit:
    "); scanf(" %c", &choice);

    if (choice == 'Q' || choice == 'q') { break;
    }

    if (choice == 'W' || choice == 'w') {
        printf("Enter data for the block: "); scanf("
        %[^\n]", data);

        // Create a new block
        struct Block* newBlock = (struct Block*)malloc(sizeof(struct Block)); for (int i =
        0; i < 256; i++) { newBlock-
            >data[i] = data[i];
        }
        newBlock->next = NULL;

        if (blockCount == 0) {
            // This is the first block firstBlock
              =          newBlock; lastBlock =
            newBlock;
        } else {
            // Link the new block to the last block lastBlock-
            >next = newBlock; lastBlock = newBlock;
        }

        blockCount++;
```

```
} else if (choice == 'R' || choice == 'r') { printf("Enter the block
    number to read (1-%d): ", blockCount); scanf("%d",
    &blockNumber);

    if (blockNumber < 1 || blockNumber > blockCount) { printf("Invalid
        block number. The valid range is 1-%d.\n",
blockCount);
    } else { struct Block* currentBlock =
        firstBlock; for
        (int i = 1; i < blockNumber; i++) {
            currentBlock = currentBlock->next;
        }

            printf("Block %d Data: %s\n", blockNumber, currentBlock->data);
    }
  }
}

// Free the allocated memory for blocks before exiting
struct Block* currentBlock = firstBlock; while
(currentBlock != NULL) { struct Block* nextBlock =
currentBlock->next; free(currentBlock); currentBlock =
nextBlock;
}

return 0;
}
```

OUTPUT:

```
Linked Allocation Simulation
Enter 'W' to write a block, 'R' to read a block, or 'Q' to quit: W
Enter data for the block: SAI IS WORST
Enter 'W' to write a block, 'R' to read a block, or 'Q' to quit: W
Enter data for the block: JASWANTH IS GOOD
Enter 'W' to write a block, 'R' to read a block, or 'Q' to quit: R
Enter the block number to read (1-2): 2
Block 2 Data: JASWANTH IS GOOD
Enter 'W' to write a block, 'R' to read a block, or 'Q' to quit: R
Enter the block number to read (1-2): 1
Block 1 Data: SAI IS WORST
Enter 'W' to write a block, 'R' to read a block, or 'Q' to quit: Q

--------------------------------
Process exited after 46.7 seconds with return value 0
Press any key to continue . . .
```

37.Construct a C program to simulate the First Come First Served disk scheduling algorithm.

AIM:- Construct a C program to simulate the First Come First Served disk scheduling algorithm.

ALGORITHM:-

1. Start at the current position of the disk head.
2. For each disk request in the queue:

- Move the disk head to the requested track.
- Calculate the seek time as the absolute difference between the new position of the disk head and the previous position.
- Add the seek time to the total seek time.
- Update the previous position of the disk head to the current position.

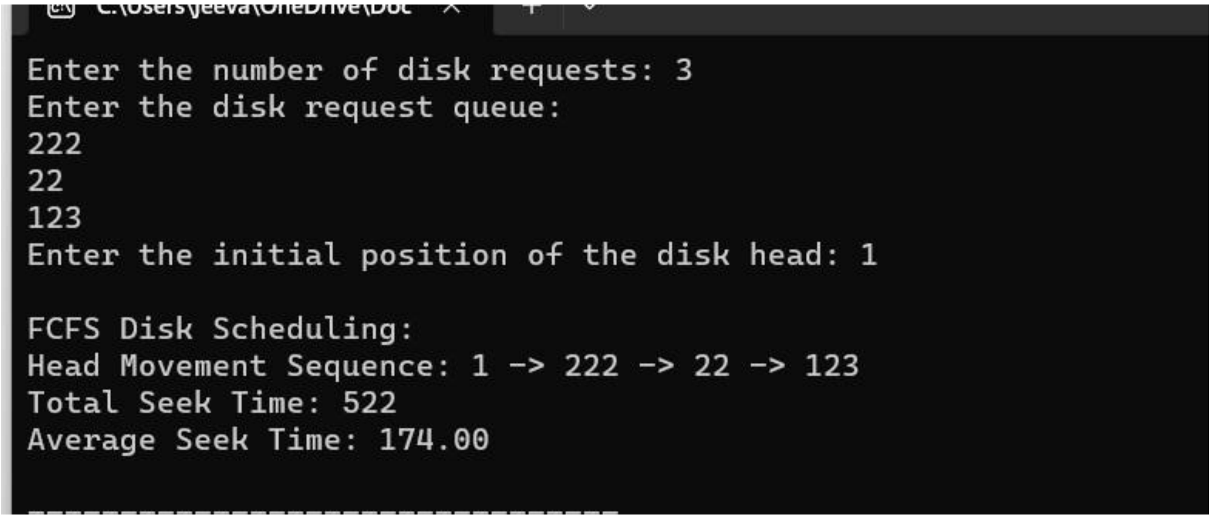
3. Repeat step 2 for all disk requests in the queue.
4. After serving all the requests, calculate and display the total seek time.
5. Calculate and display the average seek time, which is the total seek time divided by the number of requests.

PROGRAM:-

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

int main() {
    int n, head, seek_time = 0;


    printf("Enter the number of disk requests: ");
    scanf("%d", &n);

    int request_queue[n];


    printf("Enter the disk request queue:\n"); for
    (int i = 0; i < n; i++) { scanf("%d",
    &request_queue[i]);
    }


    printf("Enter the initial position of the disk head: ");
    scanf("%d", &head);

    // FCFS Scheduling
    printf("\nFCFS Disk Scheduling:\n"); printf("Head
    Movement Sequence: %d", head); for (int i = 0; i < n;
    i++) { seek_time += abs(head - request_queue[i]);
    head = request_queue[i]; printf(" -> %d", head);
    }
```

```
printf("\nTotal Seek Time: %d\n", seek_time); printf("Average Seek Time:
%.2f\n", (float) seek_time / n);


return 0;
}
```

OUTPUT:-



```
Enter the number of disk requests: 3
Enter the disk request queue:
222
22
123
Enter the initial position of the disk head: 1

FCFS Disk Scheduling:
Head Movement Sequence: 1 -> 222 -> 22 -> 123
Total Seek Time: 522
Average Seek Time: 174.00
```

38. Design a C program to simulate SCAN disk scheduling algorithm.

AIM:- Design a C program to simulate SCAN disk scheduling algorithm.

ALGORITHM:-

1. Determine the direction of movement (inward or outward) based on the queue of pending requests and the current position.

2. While servicing requests in the selected direction:

- Move the disk head to the next track in the current direction.

- Calculate the seek time as the absolute difference between the new position of the disk head and the previous position.
- Add the seek time to the total seek time.
- Update the previous position of the disk head to the current position.

3. If there are no more requests in the current direction, change direction to the opposite direction.
4. Repeat step 3 until all requests are serviced.
5. After serving all the requests, calculate and display the total seek time.
6. Calculate and display the average seek time, which is the total seek time divided by the number of requests.

PROGRAM:

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

int main() {
    int n, head, seek_time = 0;


    printf("Enter the number of disk requests: ");
    scanf("%d", &n);

    int request_queue[n];


    printf("Enter the disk request queue:\n");
    for (int i = 0; i < n; i++) { scanf("%d",
    &request_queue[i]);
    }

    printf("Enter the initial position of the disk head: ");
    scanf("%d", &head);

    // Sort the request queue to simplify SCAN algorithm for (int i
    = 0; i < n - 1; i++) { for (int j =
        i + 1; j < n; j++) {
```

```
            if (request_queue[i] > request_queue[j]) { int
                temp = request_queue[i]; request_queue[i] =
                request_queue[j]; request_queue[j] = temp;
            }
        }
    }

    // SCAN (Elevator) Scheduling
    printf("\nSCAN (Elevator) Disk Scheduling:\n"); int start
    = 0; int end =
    n - 1;
    int current_direction = 1; // 1 for moving right, -1 for moving left

    while (start <= end) {
        if (current_direction == 1) { for
            (int i = start; i <= end; i++) {
                if (request_queue[i] >= head) {
                    seek_time += abs(head - request_queue[i]); head
                    = request_queue[i]; start = i + 1; break;
                }
            }
            current_direction = -1; // Change direction
        } else { for (int i = end; i >= start;
            i--) {
                if (request_queue[i] <= head) {
                    seek_time += abs(head - request_queue[i]); head
                    = request_queue[i]; end = i - 1;
                    break;
                }
            }
            current_direction = 1; // Change direction
        }
    }
```

```
    printf("Total Seek Time: %d\n", seek_time); printf("Average Seek Time:
    %.2f\n", (float)seek_time / n);

    return 0;
}
```

Output:-

```
Enter the number of disk requests: 3
Enter the disk request queue:
12
34
45
Enter the initial position of the disk head: 45

SCAN (Elevator) Disk Scheduling:
Total Seek Time: 0
Average Seek Time: 0.00
```

39. Develop a C program to simulate C-SCAN disk scheduling algorithm.

AIM:- Develop a C program to simulate C-SCAN disk scheduling algorithm.

ALGORITHM:-

1. Start at the current position of the disk head.
2. Set the direction of movement to one side (e.g., right).
3. While servicing requests in the selected direction:
   - Move the disk head to the next track in the current direction.
   - Calculate the seek time as the absolute difference between the new position of the disk head and the previous position.
   - Add the seek time to the total seek time.
   - Update the previous position of the disk head to the current position.
4. If there are no more requests in the current direction:
   - Move the disk head to the end of the disk in the current direction.
   - Change direction to the opposite side (e.g., left).
   - Continue servicing requests in the new direction.
5. Repeat step 3 and step 4 until all requests are serviced.
6. After serving all the requests, calculate and display the total seek time.
7. Calculate and display the average seek time, which is the total seek time divided by the number of requests.

PROGRAM:-

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

int main() { int n, head,
    seek_time = 0;

    printf("Enter the number of disk requests: ");
    scanf("%d", &n);
int request_queue[n];

    printf("Enter the disk request queue:\n"); for
    (int i = 0; i < n; i++) { scanf("%d",
    &request_queue[i]);
    }

    printf("Enter the initial position of the disk head: ");
    scanf("%d", &head);

    // Sort the request queue for simplicity for
    (int i = 0; i < n - 1; i++) { for (int j = i + 1; j < n; j++)
        { if (request_queue[i] > request_queue[j]) { int
        temp = request_queue[i]; request_queue[i] =
        request_queue[j]; request_queue[j] = temp;
```

```
            }
        }
    }

    // C-SCAN Scheduling
    printf("\nC-SCAN Disk Scheduling:\n"); int
    start = 0;
    int end = n - 1;

    while (start <= end) { for (int i =
        start; i <= end; i++) { if
            (request_queue[i] >= head) { seek_time += abs(head
                - request_queue[i]); head = request_queue[i]; start
                = i + 1;
            }
        }
        // Move the head to the end in the current
        direction seek_time += abs(head - 0); head = 0;
        // Change direction to the opposite side seek_time +=
        abs(head - request_queue[end]); head =
        request_queue[end];
        end = n - 2; // Exclude the last request, as it has already been served

    }

    printf("Total Seek Time: %d\n", seek_time); printf("Average Seek Time:
    %.2f\n", (float)seek_time / n);

    return 0;
}
```

OUTPUT:-

```
Enter the number of disk requests: 3
Enter the disk request queue:
12
13
14
Enter the initial position of the disk head: 5

C-SCAN Disk Scheduling:
Total Seek Time: 37
Average Seek Time: 12.33
```

40. Illustrate the various File Access Permission and different types users in Linux.

AIM: Illustrate the various File Access Permission and different types users in Linux.

ALGORITHM:

1. Create a file or identify an existing file to demonstrate permissions and users.
2. View the file's permissions using the ls -l command. The output will look something like this: .txt

- The first character (-) represents the file type (a dash indicates a regular file).
- The next three characters (rw-) represent the permissions for the file's owner (Read and Write, no Execute).
- The next three characters (r--) represent the permissions for the file's group (Read, no Write or Execute).
- The last three characters (r--) represent the permissions for others (Read, no Write or Execute).
- The number 1 represents the number of hard links to the file.
- owner is the username of the file's owner.
- group is the name of the file's group.
- 1234 is the file's size in bytes.
- Oct 19 10:30 is the last modification timestamp.
- file.txt is the file name.

3. Use the chmod command to change the file's permissions. For example, to give the group write permission, use chmod g+w file.txt.
4. Re-run ls -l to confirm the updated permissions.
5. You can also change the file's owner and group using the chown and chgrp commands, respectively.
6. To create and manage user accounts, you can use the useradd and passwd commands.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

int main() { char filename[] =
    "file.txt";
    int new_permissions = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH; // rw-
rwr--
```

```
    if (chmod(filename, new_permissions) == 0) { printf("File permissions
        changed successfully.\n");
    }          else          {
        perror("chmod")
        ; return 1;
    }

    return 0;
}
```

OUTPUT:

1. Compile the C program (assuming it's saved in a file named `change_permissions.c`):

```bash
gcc -o change_permissions change_permissions.c
```

1. Run the program:

```bash
./change_permissions
```

Output:

If the program executes successfully, it should display the following output:

```arduino
File permissions changed successfully.
```