VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

Operating Systems

(22CS4PCOPS)

Submitted by:

Dhavan SK (1BM21CS054)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)

BENGALURU-560019

June 2023 - August 2023

B. M. S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "**Operating Systems**" carried out by **Dhavan SK(1BM21CS054),** who is bonafide student of **B. M. S. College ofEngineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University,Belgaum during the year 2022-23. The Lab report has been approved as it satisfies the academic requirements in respect of **Operating Systems - (22CS4PCOPS)** work prescribed for the said degree.

Dr Nandhini Vineeth
Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak
Professor and Head
Department of CSE
BMSCE, Bengaluru

Table Of Contents

1. Course Outcomes

CO1: Apply the different concepts and functionalities of Operating System.

CO2: Analyse various Operating system strategies and techniques.

CO3: Demonstrate the different functionalities of Operating System.

CO4: Conduct practical experiments to implement the functionalities of Operating system.

2. Experiments

2.1 Experiment - 1

2.1.1 Question:

Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time.

(a) FCFS

(b) SJF

2.1.2 Code:

(a) FCFS

```
#include <stdio.h>

int main() {

   int n;

   printf("Enter the number of processes: ");

   scanf("%d", &n);

   int pid[n], arrival[n], burst[n], waiting[n], turnaround[n];

   printf("Enter the process ids:\n");

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

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

// Input process details

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

       printf("Enter arrival time and burst time for process %d: ", i + 1);

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

   }

// Sort processes based on arrival time and then burst time
```

```
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arrival[j] == arrival[j + 1] && burst[j] > burst[j + 1]) {
                int temp = burst[j];
                burst[j] = burst[j + 1];
                burst[j + 1] = temp;

                temp = arrival[j];
                arrival[j] = arrival[j + 1];
                arrival[j + 1] = temp;

                temp = pid[j];
                pid[j] = pid[j + 1];
                pid[j + 1] = temp;
}
            else if (arrival[j] > arrival[j + 1]) {
                int temp = arrival[j];
                arrival[j] = arrival[j + 1];
                arrival[j + 1] = temp;

                temp = burst[j];
                burst[j] = burst[j + 1];
                burst[j + 1] = temp;

                temp = pid[j];
                pid[j] = pid[j + 1];
                pid[j + 1] = temp;
            }
        }
    }
    waiting[0] = 0;
    turnaround[0] = burst[0];
```

```
    // Calculate waiting and turnaround times
    for (int i = 1; i < n; i++) {
        waiting[i] = turnaround[i - 1] + arrival[i - 1] - arrival[i];
        if (waiting[i] < 0)
            waiting[i] = 0;
        turnaround[i] = waiting[i] + burst[i];
    }
    float totalWaiting = 0, totalTurnaround = 0;
    // Calculate total waiting and turnaround times
    for (int i = 0; i < n; i++) {
        totalWaiting += waiting[i];
        totalTurnaround += turnaround[i];
    }
    float avgWaiting = totalWaiting / n;
    float avgTurnaround = totalTurnaround / n;
    printf("\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", pid[i], arrival[i], burst[i], waiting[i], turnaround[i]);
    }
    printf("\nAverage Waiting Time: %.2f\n", avgWaiting);
    printf("Average Turnaround Time: %.2f\n", avgTurnaround);
    return 0;
}
```

(b) SJF

```
#include <stdio.h>
void main()
{
   int n,pid[10],bt[10],at[10],swap,tat[10],wt[10],comp=0,min,j,count=0,k;
   float t_tat=0,t_wt=0;
   printf("Enter the number of processes:\n");
   scanf("%d",&n);
   printf("Enter the process id:\n");
```

```
for (int i = 0; i < n; i++)
{
  scanf("%d", &pid[i]);
}
printf("Enter the arrival time of the processes:\n");
for (int i = 0; i < n; i++)
{
  scanf("%d", &at[i]);
}
printf("Enter the burst time of the processes:\n");
for (int i = 0; i < n; i++)
{
  scanf("%d", &bt[i]);
}
//sort based on burst time
for(int i=0;i<n-1;i++)
{
  for(int j=0;j<n-i-1;j++)
  {
    if(bt[j]>bt[j+1])
    {
      swap = pid[j];
      pid[j] = pid[j+1];
      pid[j+1] = swap;

      swap= bt[j];
      bt[j] = bt[j+1];
      bt[j+1] = swap;

      swap = at[j];
      at[j] = at[j+1];
      at[j+1] = swap;
```

```
        }
    }
}
for(int i=0;i<n;i++)
{
    tat[i]=-1;
}
//find the process which has minimum arrival time because the arrays are sorted
min=at[0];
for(int i=1;i<n;i++)
{
    if(at[i]<min)
    {
        min=at[i];
        j=i;
    }
}
comp+=at[j]+bt[j];
tat[j]=comp-at[j];
wt[j]=tat[j]-bt[j];
count++;
k=0;
while(count!=n)
{
    if(tat[k]==-1 && at[k]<=comp)
    {
        comp+=bt[k];
        tat[k]=comp-at[k];
        wt[k]=tat[k]-bt[k];
        count++;
        k=(k+1)%n;
    }
```

```
    else if(tat[k]!=-1 || at[k]>comp)
    {
      k=(k+1)%n;
    }
  }
  for(int i=0;i<n;i++)
  {
    t_tat+=tat[i];
    t_wt+=wt[i];
  }
  printf("Pid\tArrivalTime\tBurstTime\tTAT\tWaitingTime\n");
  for(int m=0;m<n;m++)
  {
    printf("%d\t\t%d\t\t%d\t%d\t%d\n", pid[m],at[m], bt[m],tat[m], wt[m]);
  }
  printf("Average turn around time:%0.2f\n", (t_tat) / n);
  printf("Average waiting time:%0.2f\n", (t_wt) / n);
}
```

2.1.3 Output:

(a) FCFS

```
Enter the number of processes: 4
Enter the process ids:
1 2 3 4
Enter arrival time and burst time for process 1: 0 3
Enter arrival time and burst time for process 2: 1 6
Enter arrival time and burst time for process 3: 4 4
Enter arrival time and burst time for process 4: 6 2

Process Arrival Time    Burst Time      Waiting Time    Turnaround Time
1       0               3               0               3
2       1               6               2               8
3       4               4               5               9
4       6               2               7               9

Average Waiting Time: 3.50
Average Turnaround Time: 7.25
```

(b) SJF

```
Enter the number of processes:
4
Enter the process id:
1 2 3 4
Enter the arrival time of the processes:
0 1 4 6
Enter the burst time of the processes:
3 6 4 2
Pid     ArrivalTime     BurstTime       TAT     WaitingTime
4               6               2       5       3
1               0               3       3       0
3               4               4       11      7
2               1               6       8       2
Average turn around time:6.75
Average waiting time:3.00
```

2.2 Experiment - 2

2.2.1 Question:

Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time.

(a) Priority (Non-pre-emptive)

(b) Round Robin (Experiment with different quantum sizes for RR algorithm)

2.2.2 Code:

(a) <u>Priority (Non-pre-emptive)</u>

```
#include <stdio.h>
void main()
{
  int n,pid[10],bt[10],at[10],pr[10],swap,tat[10],wt[10],comp=0,min,j,count=0,k;
  float t_tat=0,t_wt=0;
  printf("Enter the number of processes:\n");
  scanf("%d",&n);
  printf("Enter the process id:\n");
  for (int i = 0; i < n; i++)
  {
    scanf("%d", &pid[i]);
  }
  printf("Enter the arrival time of the processes:\n");
  for (int i = 0; i < n; i++)
  {
    scanf("%d", &at[i]);
  }
  printf("Enter the burst time of the processes:\n");
  for (int i = 0; i < n; i++)
  {
    scanf("%d", &bt[i]);
  }
  printf("Enter the priority of processes:\n");
  for (int i = 0; i < n; i++)
  {
    scanf("%d", &pr[i]);
  }

  // sorting based on priority, higher number means higher priority, so sorting in descending order
  for(int i=0;i<n-1;i++)
  {
    for(int j=0;j<n-i-1;j++)
    {
      if(pr[j]<=pr[j+1])
      {
        swap = pr[j];
        pr[j] = pr[j+1];
        pr[j+1] = swap;
```

```
        swap = pid[j];
        pid[j] = pid[j+1];
        pid[j+1] = swap;

        swap= bt[j];
        bt[j] = bt[j+1];
        bt[j+1] = swap;

        swap = at[j];
        at[j] = at[j+1];
        at[j+1] = swap;
      }
    }
}
for(int i=0;i<n;i++)
{
  tat[i]=-1;
}
//to find which process has arrived first because we have sorted the array based on priority
min=at[0];
j=0;
for(int i=1;i<n;i++)
{
  if(at[i]<min)
  {
    min=at[i];
    j=i;
  }
  else if(at[i]==min) //if arrival time is the same, check which has higher priority
  {
    if(pr[i]>pr[j])
    {
      j=i;
    }
    else if(pr[i]==pr[j]) //if priorities are also same, check whcih one has lesser burst time.
    {
      if(bt[i]<bt[j])
      {
        j=i;
      }
    }
  }
}

//j is the index/process which has arrived first, so compute tat for that first
comp+=at[j]+bt[j];
tat[j]=comp-at[j];
wt[j]=tat[j]-bt[j];

count++; //keeps track of number of processes computed for tat
k=0;
while(count!=n)
```

```
  {
    if(tat[k]==-1 && at[k]<=comp) //if tat is not yet computed and arrival time is less than completion time, then only we can compute tat
    {
      comp+=bt[k]; //update completion time
      tat[k]=comp-at[k];
      wt[k]=tat[k]-bt[k];
      k=(k+1)%n;// if the process has not arrrived, we are not computing for this process rn, so we need to come back to check for those not computed
      count++;
    }
    else if(tat[k]!=-1 || at[k]>comp)
    {
      k=(k+1)%n; // if tat already computed or the process has not yet arrived, just circularly increment
    }
  }
  for(int i=0;i<n;i++)
  {
    t_tat+=tat[i];
    t_wt+=wt[i];
  }
  printf("Pid\tArrivalTime\tBurstTime\tPriority\tTAT\tWaitingTime\n");
  for(int m=0;m<n;m++)
  {
    printf("%d\t\t%d\t\t%d\t\t%d\t%d\t\t%d\n", pid[m],at[m], bt[m],pr[m],tat[m], wt[m]);
  }
  printf("Average turn around time:%0.1f\n", (t_tat) / n);
  printf("Average waiting time:%0.1f\n", (t_wt) / n);
}
```

(b) <u>Round Robin (Non-pre-emptive)</u>

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

int turnarroundtime(int processes[], int n, int bt[], int wt[], int tat[]) {
  for (int i = 0; i < n ; i++)
  tat[i] = bt[i] + wt[i];
  return 1;
}

int waitingtime(int processes[], int n, int bt[], int wt[], int quantum)
{
  int rem_bt[n];
  for (int i = 0 ; i < n ; i++)
  rem_bt[i] = bt[i];
  int t = 0;

   while (1)
   {
      bool done = true;

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

```
        {
            if (rem_bt[i] > 0)
            {
                done = false;
                if (rem_bt[i] > quantum)
                {
                    t += quantum;
                    rem_bt[i] -= quantum;
                }

                else
                {
                    t = t + rem_bt[i];
                    wt[i] = t - bt[i];
                    rem_bt[i] = 0;
                }
            }
        }
        if (done == true)
          break;
    }
    return 1;
}

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

    waitingtime(processes, n, bt, wt, quantum);
    turnarroundtime(processes, n, bt, wt, tat);

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

    printf("\nAverage waiting time = %f", (float)total_wt / (float)n);
    printf("\nAverage turnaround time = %f", (float)total_tat / (float)n);
    return 1;
}

int main()
{
   int n, processes[n], burst_time[n], quantum;
   printf("Enter the Number of Processes: ");
   scanf("%d",&n);

   printf("\nEnter the quantum time: ");
   scanf("%d",&quantum);
```

```
  int i=0;
  for(i=0;i<n;i++)
  {
     printf("\nEnter the process: ");
     scanf("%d",&processes[i]);
     printf("Enter the Burst Time:");
     scanf("%d",&burst_time[i]);
 }

 findavgTime(processes, n, burst_time, quantum);
 return 0;
}
```

2.2.3 Output:

(a) Priority (Non-pre-emptive)

```
Enter the number of processes:
4
Enter the process id:
1 2 3 4
Enter the arrival time of the processes:
0 1 2 3
Enter the burst time of the processes:
4 3 3 5
Enter the priority of processes:
3 4 6 5
Min:0
j:3
Pid     ArrivalTime     BurstTime       Priority        TAT     WaitingTime
3               2               3               6       5               2
4               3               5               5       9               4
2               1               3               4       14              11
1               0               4               3       4               0
Average turn around time:8.0
Average waiting time:4.3
```

(b) Round Robin (Non-pre-emptive)

```
Enter the Number of Processes: 3

Enter the quantum time: 2

Enter the process: 1
Enter the Burst Time:4

Enter the process: 2
Enter the Burst Time:3

Enter the process: 3
Enter the Burst Time:5


Processes               Burst Time              Waiting Time            turnaround time

        1                       4                       4                       8

        2                       3                       6                       9

        3                       5                       7                       12

Average waiting time = 5.666667
Average turnaround time = 9.666667
```

2.3 Experiment - 3

2.3.1 Question:

Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories – system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling for the processes in each queue.

2.3.2 Code:

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

struct process {
    int pid;
    int arrival_time;
    int burst_time;
    int priority;
    int waiting_time;
    int turnaround_time;
};

void FCFS(struct process *queue, int n) {
    int i, j;
    struct process temp;
    for (i = 0; i < n; i++) {
        for (j = i + 1; j < n; j++) {
            if (queue[i].arrival_time > queue[j].arrival_time) {
                temp = queue[i];
                queue[i] = queue[j];
                queue[j] = temp;
            }
        }
    }
}

int main() {
    int n, i;
    struct process *system_queue, *user_queue;
    int system_n = 0, user_n = 0;
    float avg_waiting_time = 0, avg_turnaround_time = 0;

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

    system_queue = (struct process *) malloc(n * sizeof(struct process));
    user_queue = (struct process *) malloc(n * sizeof(struct process));

    for (i = 0; i < n; i++) {
        struct process p;
        printf("Enter arrival time, burst time, and priority (0-System/1-User) for process %d: ", i + 1);
```

```
        scanf("%d %d %d", &p.arrival_time, &p.burst_time, &p.priority);
        p.pid = i + 1;
        p.waiting_time = 0;
        p.turnaround_time = 0;
        if (p.priority == 0) {
            system_queue[system_n++] = p;
        } else {
            user_queue[user_n++] = p;
        }
    }

    FCFS(system_queue, system_n);
    FCFS(user_queue, user_n);

    int time = 0;
    int s=0,u=0;
    while(s<system_n || u<user_n){
        if(system_queue[s].arrival_time <= time){
            if(user_queue[u].arrival_time <= time && user_queue[u].arrival_time <
system_queue[s].arrival_time){
                user_queue[u].waiting_time = time - user_queue[u].arrival_time;
                time += user_queue[u].burst_time;
                user_queue[u].turnaround_time = user_queue[u].waiting_time + user_queue[u].burst_time;
                avg_waiting_time += user_queue[u].waiting_time;
                avg_turnaround_time += user_queue[u].turnaround_time;
                u++;
            }
            else{
                system_queue[s].waiting_time = time - system_queue[s].arrival_time;
                time += system_queue[s].burst_time;
                system_queue[s].turnaround_time = system_queue[s].waiting_time + system_queue[s].burst_time;
                avg_waiting_time += system_queue[s].waiting_time;
                avg_turnaround_time += system_queue[s].turnaround_time;
                s++;
            }
        }
        else if(user_queue[u].arrival_time <= time){
            user_queue[u].waiting_time = time - user_queue[u].arrival_time;
                time += user_queue[u].burst_time;
                user_queue[u].turnaround_time = user_queue[u].waiting_time + user_queue[u].burst_time;
                avg_waiting_time += user_queue[u].waiting_time;
                avg_turnaround_time += user_queue[u].turnaround_time;
                u++;
        }
        else{
            if(system_queue[s].arrival_time <= user_queue[u].arrival_time){
                time = system_queue[s].arrival_time;
            }
            else{
                time = user_queue[u].arrival_time;
            }
        }
```

```
    }

    avg_waiting_time /= n;
    avg_turnaround_time /= n;

    printf("PID\tBurst Time\tPriority\tQueue Type\tWaiting Time\tTurnaround Time\n");
    for (i = 0; i < system_n; i++) {
        printf("%d\t%d\t\t%d\t\tSystem\t\t%d\t\t%d\n", system_queue[i].pid, system_queue[i].burst_time,
system_queue[i].priority, system_queue[i].waiting_time, system_queue[i].turnaround_time);
    }
    for (i = 0; i < user_n; i++) {
        printf("%d\t%d\t\t%d\t\tUser\t\t%d\t\t%d\n", user_queue[i].pid, user_queue[i].burst_time,
user_queue[i].priority, user_queue[i].waiting_time, user_queue[i].turnaround_time);
    }

    printf("Average Waiting Time: %.2f\n", avg_waiting_time);
    printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);

    free(system_queue);
    free(user_queue);

    return 0;
}
```

2.3.3 Output:

```
Enter the number of processes: 4
Enter arrival time, burst time, and priority (0-System/1-User) for process 1: 0 3 0
Enter arrival time, burst time, and priority (0-System/1-User) for process 2: 1 3 1
Enter arrival time, burst time, and priority (0-System/1-User) for process 3: 8 3 0
Enter arrival time, burst time, and priority (0-System/1-User) for process 4: 8 3 1
PID     Burst Time      Priority        Queue Type      Waiting Time    Turnaround Time
1       3               0               System          0               3
3       3               0               System          0               3
2       3               1               User            2               5
4       3               1               User            3               6
Average Waiting Time: 1.25
Average Turnaround Time: 4.25
```

2.4 Experiment – 4

2.4.1 Question:

Write a C program to simulate Real-Time CPU Scheduling algorithms:

(a) Rate- Monotonic

(b) Earliest-deadline First

(c) Proportional scheduling

2.4.2 Code:

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

#define MAX_PROCESS 10

typedef struct {
   int id;
   int burst_time;
   float priority;
} Task;

int num_of_process;
int execution_time[MAX_PROCESS], period[MAX_PROCESS], remain_time[MAX_PROCESS],
deadline[MAX_PROCESS], remain_deadline[MAX_PROCESS];

void get_process_info(int selected_algo)
{
   printf("Enter total number of processes (maximum %d): ", MAX_PROCESS);
   scanf("%d", &num_of_process);
   if (num_of_process < 1)
   {
      exit(0);
   }

   for (int i = 0; i < num_of_process; i++)
   {
      printf("\nProcess %d:\n", i + 1);
      printf("➔ Execution time: ");
      scanf("%d", &execution_time[i]);
      remain_time[i] = execution_time[i];
      if (selected_algo == 2)
      {
         printf("➔ Deadline: ");
         scanf("%d", &deadline[i]);
      }
      else
      {
         printf("➔ Period: ");
         scanf("%d", &period[i]);
      }
   }
```

```
}

int max(int a, int b, int c)
{
   int max;
   if (a >= b && a >= c)
      max = a;
   else if (b >= a && b >= c)
      max = b;
   else if (c >= a && c >= b)
      max = c;
   return max;
}

int get_observation_time(int selected_algo)
{
   if (selected_algo == 1)
   {
      return max(period[0], period[1], period[2]);
   }
   else if (selected_algo == 2)
   {
      return max(deadline[0], deadline[1], deadline[2]);
   }
}

void print_schedule(int process_list[], int cycles)
{
   printf("\nScheduling:\n\n");
   printf("Time: ");
   for (int i = 0; i < cycles; i++)
   {
      if (i < 10)
         printf("| 0%d ", i);
      else
         printf("| %d ", i);
   }
   printf("|\n");
   for (int i = 0; i < num_of_process; i++)
   {
      printf("P[%d]: ", i + 1);
      for (int j = 0; j < cycles; j++)
      {
         if (process_list[j] == i + 1)
            printf("|####");
         else
            printf("|    ");
      }
      printf("|\n");
   }
}
```

```
void rate_monotonic(int time)
{
    int process_list[100] = {0}, min = 999, next_process = 0;
    float utilization = 0;
    for (int i = 0; i < num_of_process; i++)
    {
        utilization += (1.0 * execution_time[i]) / period[i];
    }
    int n = num_of_process;
    int m = (float) (n * (pow(2, 1.0 / n) – 1));
    if (utilization > m)
    {
        printf("\nGiven problem is not schedulable under the said scheduling algorithm.\n");
    }
    for (int i = 0; i < time; i++)
    {
        min = 1000;
        for (int j = 0; j < num_of_process; j++)
        {
            if (remain_time[j] > 0)
            {
                if (min > period[j])
                {
                    min = period[j];
                    next_process = j;
                }
            }
        }
        if (remain_time[next_process] > 0)
        {
            process_list[i] = next_process + 1;
            remain_time[next_process] -= 1;
        }
        for (int k = 0; k < num_of_process; k++)
        {
            if ((i + 1) % period[k] == 0)
            {
                remain_time[k] = execution_time[k];
                next_process = k;
            }
        }
    }
    print_schedule(process_list, time);
}

void earliest_deadline_first(int time){
    float utilization = 0;
    for (int i = 0; i < num_of_process; i++){
        utilization += (1.0*execution_time[i])/deadline[i];
    }
    int n = num_of_process;
```

```
int process[num_of_process];
int max_deadline, current_process=0, min_deadline,process_list[time];
bool is_ready[num_of_process];

for(int i=0; i<num_of_process; i++){
    is_ready[i] = true;
    process[i] = i+1;
}

max_deadline=deadline[0];
for(int i=1; i<num_of_process; i++){
    if(deadline[i] > max_deadline)
        max_deadline = deadline[i];
}

for(int i=0; i<num_of_process; i++){
    for(int j=i+1; j<num_of_process; j++){
        if(deadline[j] <  deadline[i]){
            int temp = execution_time[j];
            execution_time[j] = execution_time[i];
            execution_time[i] = temp;
            temp = deadline[j];
            deadline[j] = deadline[i];
            deadline[i] = temp;
            temp = process[j];
            process[j] = process[i];
            process[i] = temp;
        }
    }
}

for(int i=0; i<num_of_process; i++){
    remain_time[i] = execution_time[i];
    remain_deadline[i] = deadline[i];
}

for (int t = 0; t < time; t++){
    if(current_process != -1){
        --execution_time[current_process];
        process_list[t] = process[current_process];
    }
    else
        process_list[t] = 0;

    for(int i=0;i<num_of_process;i++){
        --deadline[i];
        if((execution_time[i] == 0) && is_ready[i]){
            deadline[i] += remain_deadline[i];
            is_ready[i] = false;
        }
        if((deadline[i] <= remain_deadline[i]) && (is_ready[i] == false)){
            execution_time[i] = remain_time[i];
```

```
            is_ready[i] = true;
        }
    }

    min_deadline = max_deadline;
    current_process = -1;
    for(int i=0;i<num_of_process;i++){
        if((deadline[i] <= min_deadline) && (execution_time[i] > 0)){
            current_process = i;
            min_deadline = deadline[i];
        }
    }
  }
  print_schedule(process_list, time);
}

void proportionalScheduling() {
  int n;
  printf(“Enter the number of tasks: “);
  scanf(“%d”, &n);

  Task tasks[n];
  printf(“Enter burst time and priority for each task:\n”);
  for (int i = 0; i < n; i++) {
    tasks[i].id = i + 1;
    printf(“Task %d – Burst Time: “, tasks[i].id);
    scanf(“%d”, &tasks[i].burst_time);
    printf(“Task %d – Priority: “, tasks[i].id);
    scanf(“%f”, &tasks[i].priority);
  }

  // Sort tasks based on priority (ascending order)
  for (int i = 0; i < n – 1; i++) {
    for (int j = 0; j < n – i – 1; j++) {
      if (tasks[j].priority > tasks[j + 1].priority) {
        // Swap tasks
        Task temp = tasks[j];
        tasks[j] = tasks[j + 1];
        tasks[j + 1] = temp;
      }
    }
  }

  printf(“\nProportional Scheduling:\n”);

  int total_burst_time = 0;
  float total_priority = 0.0;

  for (int i = 0; i < n; i++) {
    total_burst_time += tasks[i].burst_time;
    total_priority += tasks[i].priority;
  }
```

```
    for (int i = 0; i < n; i++) {
        float time_slice = (tasks[i].priority / total_priority) * total_burst_time;
        printf("Task %d executes for %.2f units of time\n", tasks[i].id, time_slice);
    }
}


int main()
{
    int option;
    int observation_time;

    while (1)
    {
    printf("\n1. Rate Monotonic\n2. Earliest Deadline first\n3. Proportional Scheduling\n\nEnter your choice:
");
    scanf("%d", &option);
    switch(option)
    {
        case 1: get_process_info(option);
                observation_time = get_observation_time(option);
                rate_monotonic(observation_time);
                break;
        case 2: get_process_info(option);
                observation_time = get_observation_time(option);
                earliest_deadline_first(observation_time);
                break;
        case 3: proportionalScheduling();
                break;
        case 4: exit (0);
        default: printf("\nInvalid Statement");
    }
    }
    return 0;
}
```

2.4.3 Output:

(a) Rate Monotonic:

```
1. Rate Monotonic
2. Earliest Deadline first
3. Proportional Scheduling

Enter your choice: 1
Enter total number of processes (maximum 10): 3

Process 1:
==> Execution time: 3
==> Period: 20

Process 2:
==> Execution time: 2
==> Period: 5

Process 3:
==> Execution time: 2
==> Period: 10

Scheduling:

Time: | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
P[1]: |    |    |    |    |####|    |    |####|####|    |    |    |    |    |    |    |    |    |    |    |
P[2]: |####|####|    |    |    |####|####|    |    |    |####|####|    |    |    |####|####|    |    |    |
P[3]: |    |    |####|####|    |    |    |    |    |    |    |    |####|####|    |    |    |    |    |    |
```

(b) Earliest Deadline First:

```
1. Rate Monotonic
2. Earliest Deadline first
3. Proportional Scheduling

Enter your choice: 2
Enter total number of processes (maximum 10): 3

Process 1:
==> Execution time: 3
==> Deadline: 7

Process 2:
==> Execution time: 2
==> Deadline: 4

Process 3:
==> Execution time: 2
==> Deadline: 8

Scheduling:

Time: | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
P[1]: |    |    |####|####|####|    |    |    |
P[2]: |####|####|    |    |    |    |    |####|
P[3]: |    |    |    |    |    |####|####|    |
```

© Proportional Scheduling:

```
1. Rate Monotonic
2. Earliest Deadline first
3. Proportional Scheduling

Enter your choice: 3
Enter the number of tasks: 3
Enter burst time and priority for each task:
Task 1 - Burst Time: 4
Task 1 - Priority: 2
Task 2 - Burst Time: 6
Task 2 - Priority: 3
Task 3 - Burst Time: 5
Task 3 - Priority: 1

Proportional Scheduling:
Task 3 executes for 2.50 units of time
Task 1 executes for 5.00 units of time
Task 2 executes for 7.50 units of time
```

2.5 Experiment – 5

2.5.1 Question:

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

2.5.2 Code:

```
#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>


#define BUFFER_SIZE 10

#define NUM_ITEMS 20


int buffer[BUFFER_SIZE];

int fill = 0; // Index to add data by producer

int use = 0; // Index to consume data by consumer

int count = 0; // Number of items in the buffer


sem_t empty; // Semaphore to track empty slots in the buffer

sem_t full; // Semaphore to track the number of items available for consumption


void put(int value) {

    buffer[fill] = value;

    fill = (fill + 1) % BUFFER_SIZE;

    count++;

}


int get() {

    int tmp = buffer[use];

    use = (use + 1) % BUFFER_SIZE;

    count--;

    return tmp;

}
```

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < NUM_ITEMS; i++) {
        sem_wait(&empty); // Wait for an empty slot
        put(i);
        printf("Produced: %d\n", i);
        sem_post(&full); // Signal that an item is produced
    }
    pthread_exit(NULL);
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < NUM_ITEMS; i++) {
        sem_wait(&full); // Wait for an item to be produced
        int value = get();
        printf("Consumed: %d\n", value);
        sem_post(&empty); // Signal that an empty slot is available
    }
    pthread_exit(NULL);
}

int main() {
    // Initialize semaphores
    sem_init(&empty, 0, BUFFER_SIZE); // Set empty slots to BUFFER_SIZE
    sem_init(&full, 0, 0); // No items available initially

    pthread_t producer_thread, consumer_thread;

    // Create 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;
}
```

2.5.3 Output:

```
Produced:0
Produced:1
Produced:2
Produced:3
Produced:4
Coonsumed:0
Coonsumed:1
Coonsumed:2
Coonsumed:3
Coonsumed:4
Produced:5
Produced:6
Produced:7
Produced:8
Produced:9
Coonsumed:5
Coonsumed:6
Coonsumed:7
Coonsumed:8
Coonsumed:9
```

2.6 Experiment – 6

2.6.1 Question:

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

2.6.2 Code:

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

sem_t room;
sem_t chopstick[5];

void * philosopher(void *);
void eat(int);
int main()
{
int i,a[5];
pthread_t tid[5];

sem_init(&room,0,4);

for(i=0;i<5;i++)
sem_init(&chopstick[i],0,1);

for(i=0;i<5;i++){
a[i]=i;
pthread_create(&tid[i],NULL,philosopher,(void *)&a[i]);
}
for(i=0;i<5;i++)
pthread_join(tid[i],NULL);
}

void * philosopher(void * num)
{
int phil=*(int *)num;

sem_wait(&room);
printf("\nPhilosopher %d has entered room",phil);
sem_wait(&chopstick[phil]);
sem_wait(&chopstick[(phil+1)%5]);

eat(phil);
sleep(2);
printf("\nPhilosopher %d has finished eating",phil);

sem_post(&chopstick[(phil+1)%5]);
sem_post(&chopstick[phil]);
sem_post(&room);
}
```

void eat(int phil)
{
printf("\nPhilosopher %d is eating",phil);
}

2.6.3 Output:

```
Philo 4 has entered the room.
Philo 4 has started eating.
Philo 3 has entered the room.
Philo 2 has entered the room.
Philo 1 has entered the room.
Philo 4 has finished eating.
Philo 0 has entered the room.
Philo 3 has started eating.
Philo 3 has finished eating.
Philo 2 has started eating.
Philo 2 has finished eating.
Philo 1 has started eating.
Philo 1 has finished eating.
Philo 0 has started eating.
Philo 0 has finished eating.
```

2.7 Experiment – 7

2.7.1 Question:

Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

2.7.2 Code:

```
#include <stdio.h>

int main()
{
   int n, m, i, j, k;
   printf("Enter the number of processes: ");
   scanf("%d", &n);
   printf("Enter the number of resources: ");
   scanf("%d", &m);

   int allocation[n][m];
   printf("Enter the Allocation Matrix:\n");
   for (i = 0; i < n; i++)
   {
      for (j = 0; j < m; j++)
      {
         scanf("%d", &allocation[i][j]);
      }
   }

   int max[n][m];
   printf("Enter the MAX Matrix:\n");
   for (i = 0; i < n; i++)
   {
      for (j = 0; j < m; j++)
      {
         scanf("%d", &max[i][j]);
      }
   }

   int available[m];
   printf("Enter the Available Resources:\n");
   for (i = 0; i < m; i++)
   {
      scanf("%d", &available[i]);
   }

   int f[n], ans[n], ind = 0;
   for (k = 0; k < n; k++)
   {
      f[k] = 0;
   }

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

```
    for (j = 0; j < m; j++)
    {
        need[i][j] = max[i][j] – allocation[i][j];
    }
}

int y = 0;
for (k = 0; k < n; k++)
{
    for (i = 0; i < n; i++)
    {
        if (f[i] == 0)
        {
            int flag = 0;
            for (j = 0; j < m; j++)
            {
                if (need[i][j] > available[j])
                {
                    flag = 1;
                    break;
                }
            }

            if (flag == 0)
            {
                ans[ind++] = i;
                for (y = 0; y < m; y++)
                {
                    available[y] += allocation[i][y];
                }
                f[i] = 1;
            }
        }
    }
}

int flag = 1;
for (i = 0; i < n; i++)
{
    if (f[i] == 0)
    {
        flag = 0;
        printf("The following system is not safe\n");
        break;
    }
}

if (flag == 1)
{
    printf("Following is the SAFE Sequence\n");
    for (i = 0; i < n – 1; i++)

{
```

```
            printf(" P%d ->", ans[i]);
        }
        printf(" P%d\n", ans[n – 1]);
    }
    return 0;
}
```

2.7.3 Output:

```
Enter the number of processes: 5
Enter the number of resources: 3
Enter the Allocation Matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter the MAX Matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter the Available Resources:
3 3 2
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P2
```

```
Enter the number of processes: 5
Enter the number of resources: 3
Enter the Allocation Matrix:
0 2 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter the MAX Matrix:
8 4 6
3 5 7
3 6 7
9 5 3
2 5 7
Enter the Available Resources:
3 2 2
The following system is not safe
```

2.8 Experiment – 8

2.8.1 Question:

Write a C program to simulate deadlock detection.

2.8.2 Code:

```
#include<stdio.h>

int max[100][100];
int allocation[100][100];
int need[100][100];
int available[100];
int n,r;

int main()
{
   int i,j;
   printf("Deadlock Detection\n");
   input();
   show();
   cal();
   return 0;
}

void input()
{
   int i,j;
   printf("Enter the no of Processes: ");
   scanf("%d",&n);
   printf("Enter the no of resource instances: ");
   scanf("%d",&r);
   printf("Enter the Max Matrix:\n");
   for(i=0;i<n;i++)
   {
      for(j=0;j<r;j++)
      {
         scanf("%d",&max[i][j]);
      }
   }
   printf("Enter the Allocation Matrix:\n");
   for(i=0;i<n;i++)
   {
      for(j=0;j<r;j++)
      {
         scanf("%d",&allocation[i][j]);
      }
   }
   printf("Enter the available Resources:\n");
   for(j=0;j<r;j++)
   {
      scanf("%d",&available[j]);
   }
```

```
}

void show()
{
   int i,j;
   printf("Process\t Allocation\t Max\t Available\t");
   for(i=0;i<n;i++)
   {
      printf("\nP%d\t ",i+1);
      for(j=0;j<r;j++)
      {
         printf("%d ",allocation[i][j]);
      }
      printf("\t");
      for(j=0;j<r;j++)
      {
         printf("%d ",max[i][j]);
      }
      printf("\t");
      if(i==0)
      {
         for(j=0;j<r;j++)
         printf("%d ",available[j]);
      }
   }
}

void cal()
{
   int finish[100],temp,need[100][100],flag=1,k,c1=0;
   int dead[100];
   int  safe[100];
   int i,j;
   for(i=0;i<n;i++)
   {
      finish[i]=0;
   }

   for(i=0;i<n;i++)
   {
      for(j=0;j<r;j++)
      {
         need[i][j]=max[i][j]-allocation[i][j];
      }
   }
   while(flag)
   {
      flag=0;
      for(i=0;i<n;i++)
      {
         int c=0;
         for(j=0;j<r;j++)
```

```
      {
        if((finish[i]==0)&&(need[i][j]<=available[j]))
        {
          c++;
          if(c==r)
          {
            for(k=0;k<r;k++)
            {
              available[k]+=allocation[i][j];
              finish[i]=1;
              flag=1;
            }
            if(finish[i]==1)
            {
              i=n;
            }
          }
        }
      }
    }
  }

  j=0;
  flag=0;
  for(i=0;i<n;i++)
  {
    if(finish[i]==0)
    {
      dead[j]=i;
      j++;
      flag=1;
    }
  }
  if(flag==1)
  {
    printf("\n\nSystem is in Deadlock and the Deadlock process are\n");
    for(i=0;i<n;i++)
    {
      printf("P%d\t",dead[i]);
    }
  }
  else
  {
    printf("\nNo Deadlock Occur");
  }
}
```

2.8.3 Output:

```
Deadlock Detection
Enter the no of Processes: 3
Enter the no of resource instances: 3
Enter the Max Matrix:
3 6 8
4 3 3
3 4 4
Enter the Allocation Matrix:
3 3 3
2 0 4
1 2 4
Enter the available Resources:
1 2 0
Process  Allocation      Max     Available
P0          3 3 3        3 6 8    1 2 0
P1          2 0 4        4 3 3
P2          1 2 4        3 4 4

System is in Deadlock and the Deadlock process are
P0       P1       P2
```

```
Deadlock Detection
Enter the no of Processes: 5
Enter the no of resource instances: 3
Enter the Max Matrix:
0 0 0
2 0 2
0 0 0
1 0 0
0 0 2
Enter the Allocation Matrix:
0 1 0
2 0 0
3 0 3
3 1 1
0 0 2
Enter the available Resources:
0 0 0
Process  Allocation      Max     Available
P0          0 1 0        0 0 0    0 0 0
P1          2 0 0        2 0 2
P2          3 0 3        0 0 0
P3          3 1 1        1 0 0
P4          0 0 2        0 0 2
No Deadlock Occur
```

2.9 Experiment – 9

2.9.1 Question:

Write a C program to simulate the following contiguous memory allocation techniques:

(a) Worst-fit

(b) Best-fit

(c) First-fit

2.9.2 Code:

(a) Worst-fit

```
#include<stdio.h>
void main()
{
   int n,m,i,j;
   printf("Enter the number of processes and number of blocks:\n");
   scanf("%d %d",&n,&m);
   int all[n],blockSize[m],processSize[n];
   printf("Enter %d process sizes:\n",n);
   for(i=0;i<n;i++)
   {
      scanf("%d",&processSize[i]);
      all[i]=-1;
   }
   printf("Enter %d block sizes:\n",m);
   for(j=0;j<m;j++)
   {
      scanf("%d",&blockSize[j]);
   }

   //Since this is worst fit, the largest available partition should be allocated. So we can sort the block sizes
in descending order.
   for(i=0;i<m-1;i++)
   {
      for(j=0;j<m-i-1;j++)
      {
         if(blockSize[j]<=blockSize[j+1])
         {
            int temp=blockSize[j];
            blockSize[j]=blockSize[j+1];
            blockSize[j+1]=temp;
         }
      }
   }
   for(i=0;i<n;i++)
   {
      for(j=0;j<m;j++)
      {
         if(blockSize[j]>=processSize[i])
         {
```

```
            all[i]=blockSize[j];
            blockSize[j]=-1;
            break;
         }
      }
   }
   printf("****Worst fit memory allocation****\n");
   printf("ProcessID\tProcess_Size\tBlock_size_allocated\n");
   for(i=0;i<n;i++)
   {
      printf("P%d\t\t",(i+1));
      printf("%d\t\t",processSize[i]);
      if(all[i]==-1)
      printf("Not allocated\n");
      else
      printf("%d\n",all[i]);
   }
}
```

(b) Best-fit

```
#include<stdio.h>
void main()
{
   int n,m,i,j;
   printf("Enter the number of processes and number of blocks:\n");
   scanf("%d %d",&n,&m);
   int all[n],blockSize[m],processSize[n];
   printf("Enter %d process sizes:\n",n);
   for(i=0;i<n;i++)
   {
      scanf("%d",&processSize[i]);
      all[i]=-1;
   }
   printf("Enter %d block sizes:\n",m);
   for(j=0;j<m;j++)
   {
      scanf("%d",&blockSize[j]);
   }

   /*Since this is best fit, the smallest partition which is adequate is allocated to the processes. So we can
sort the blockSizes
   in ascending order. */
   for(i=0;i<m-1;i++)
   {
      for(j=0;j<m-i-1;j++)
      {
         if(blockSize[j]>blockSize[j+1])
         {
            int temp=blockSize[j];
            blockSize[j]=blockSize[j+1];
            blockSize[j+1]=temp;
```

```
        }
    }
  }
  for(i=0;i<n;i++)
  {
    for(j=0;j<m;j++)
    {
      if(blockSize[j]>=processSize[i])
      {
        all[i]=blockSize[j];
        blockSize[j]=-1;
        break;
      }
    }
  }
  printf("****Best fit memory allocation****\n");
  printf("ProcessID\tProcess_Size\tBlock_size_allocated\n");
  for(i=0;i<n;i++)
  {
    printf("P%d\t\t",(i+1));
    printf("%d\t\t",processSize[i]);
    if(all[i]==-1)
    printf("Not allocated\n");
    else
    printf("%d\n",all[i]);
  }
}
```

(c) First-fit

```
#include<stdio.h>
void main()
{
  int n,m,i,j,c=0;
  printf("Enter the number of processes and number of blocks:\n");
  scanf("%d %d",&n,&m);
  int all[n];
  for(int i=0;i<n;i++)
  {
    all[i]=-1;
  }
  int blockSize[m],processSize[n];
  printf("Enter the %d block sizes:\n",m);
  for(j=0;j<m;j++)
  {
    scanf("%d",&blockSize[j]);
  }
  printf("Enter the %d process sizes:\n",n);
  {
    for(i=0;i<n;i++)
    {
      scanf("%d",&processSize[i]);
    }
```

```
    }
    for(i=0;i<n;i++)
    {
        for(j=0;j<m;j++)
        {
            if(blockSize[j]>=processSize[i])
            {
                all[i]=blockSize[j];
                blockSize[j]=-1;
                break;
            }
        }
    }
    printf("****First fit memory allocation****\n");
    printf("ProcessId\tProcessSize\tBlock_Size_allocated\n");
    for(i=0;i<n;i++)
    {
        printf("P%d\t\t",(i+1));
        printf("%d\t\t",processSize[i]);
        if(all[i]!=-1)
        printf("%d\n",all[i]);
        else
        printf("Not allocated\n");
    }
}
```

2.9.3 Output:

(a) Worst-fit

```
Enter the number of processes and number of blocks:
4 5
Enter 4 process sizes:
212 417 112 426
Enter 5 block sizes:
100 500 200 300 600
****Worst fit memory allocation****
ProcessID        Process_Size     Block_size_allocated
P1               212              600
P2               417              500
P3               112              300
P4               426              Not allocated
```

(b) Best-fit

```
Enter the number of processes and number of blocks:
4 5
Enter 4 process sizes:
212 417 112 426
Enter 5 block sizes:
100 500 200 300 600
****Best fit memory allocation****
ProcessID        Process_Size     Block_size_allocated
P1               212              300
P2               417              500
P3               112              200
P4               426              600
```

(c) First-fit

```
Enter the number of processes and number of blocks:
4 5
Enter the 5 block sizes:
100 500 200 300 600
Enter the 4 process sizes:
212 417 112 426
****First fit memory allocation****
ProcessId        ProcessSize      Block_Size_allocated
P1               212              500
P2               417              600
P3               112              200
P4               426              Not allocated
```

2.10 Experiment – 10

2.10.1 Question:

Write a C program to simulate paging technique of memory management.

2.10.2 Code:

```
#include<stdio.h>
#define MAX 50
int main()
{
   int page[MAX],i,n,f,ps,off,pno;
   int choice=0;
   printf("Enter the number of pages in memory: ");
   scanf("%d",&n);
   printf("\nEnter Page size: ");
   scanf("%d",&ps);

   printf("\nEnter number of frames: ");
   scanf("%d",&f);
   for(i=0;i<n;i++)
      page[i]=-1;

   printf("\nEnter the Page Table\n");
   printf("(Enter frame no as -1 if that page is not present in any frame)\n\n");

   printf("\nPage No\t\tFrame No\n-------\t\t ------");
   for(i=0;i<n;i++)
   {
      printf("\n\n%d\t\t",i);
      scanf("%d",&page[i]);
   }

   do
   {
      printf("\n\nEnter the logical address(i.e,page no & offset):");
      scanf("%d%d",&pno,&off);

      if(page[pno]==-1)
         printf("\n\nThe required page is not available in any of frames");
      else
         printf("\nPhysical address(i.e,frame no & offset):%d,%d",page[pno],off);

      printf("\n\nDo you want to continue(1/0)?:");
      scanf("%d",&choice);
   }while(choice==1);

   return 1;
}
```

2.10.3 Output:

```
Enter the number of pages in memory: 4

Enter Page size: 10

Enter number of frames: 4

Enter the Page Table
(Enter frame no as -1 if that page is not present in any frame)


Page No         Frame No
-------         -------

0               -1


1               8


2               5


3               2
```

```
Enter the logical address(i.e,page no & offset):0 100


The required page is not available in any of frames

Do you want to continue(1/0)?:1


Enter the logical address(i.e,page no & offset):1 25

Physical address(i.e,frame no & offset):8,25

Do you want to continue(1/0)?:1


Enter the logical address(i.e,page no & offset):2 352

Physical address(i.e,frame no & offset):5,352

Do you want to continue(1/0)?:1


Enter the logical address(i.e,page no & offset):3 20

Physical address(i.e,frame no & offset):2,20

Do you want to continue(1/0)?:0
```

2.11 Experiment – 11

2.11.1 Question:

Write a C program to simulate page replacement algorithms:

(a) FIFO

(b) LRU

(c) Optimal

2.11.2 Code:

(a) FIFO

```
#include<stdio.h>
int isHit(int fr[], int pg, int m)
{
   int hit=0;
   for(int i=0;i<m;i++)
   {
      if(fr[i]==pg)
      {
         hit=1;
         break;
      }
   }
   return hit;
}
void main()
{
   int n,m,k=0,pagefault=0;
   printf("Enter the length of reference sequence:\n");
   scanf("%d",&n);
   int ref[n];
   printf("Enter the page reference sequence:\n");
   for(int i=0;i<n;i++)
   {
```

```
        scanf("%d",&ref[i]);
    }
    printf("Enter the number of frames:\n");
    scanf("%d",&m);
    int fr[m];
    for(int i=0;i<m;i++)
    {
        fr[i]=-1;
    }
    for(int i=0;i<n;i++)
    {
        //if it is not a hit

        if(isHit(fr,ref[i],m)==0)
        {
            fr[k]=ref[i];
            k=(k+1)%m; //since this is first come first serve.
            pagefault++;
            printf("%d:Page Fault\n",ref[i]);
        }
        else
        printf("%d:No page fault\n",ref[i]);
    }
    printf("Total number of page faults:%d\n",pagefault);
}
```

(b) Optimal

```
#include<stdio.h>

int isHit(int fr[], int pg, int m)

{

    int hit=0;

    for(int i=0;i<m;i++)

    {

        if(fr[i]==pg)

        {

            hit=1;

            break;

        }

    }

    return hit;

}

void main()

{

    int i,n,m,k,j,pagefault=0,max=-1,x,y,flag=0,count=0,u;

    printf("Enter the length of reference sequence:\n");

    scanf("%d",&n);

    int ref[n];

    printf("Enter the page reference sequence:\n");

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

    {

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

    }

    printf("Enter the number of frames:\n");

    scanf("%d",&m);

    int fr[m];

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

    {

        fr[i]=-1;
```

```
}
u=0;
y=0;
while(count<m)
{
    if(isHit(fr,ref[u],m)==0)
    {
        fr[y]=ref[u];
        printf("%d:Page fault\n",ref[u]);
        u++;
        y++;
        count++;
        pagefault++;
    }
    else
    {
        printf("%d:No page fault\n",ref[u]);
        u++;
    }
}
for(i=u;i<n;i++)
{
    if(isHit(fr,ref[i],m)==0)
    {
        for(j=0;j<m;j++)
        {
            for(k=i+1;k<n;k++)
            {
                if(fr[j]==ref[k])//as soon as match happens, break.
                {
                    flag=1;
                    break;
```

```
                }
}               else if(k==n-1 && fr[j]!=ref[k])//if there is no demand of a particular page in future, just
replace that.
                {
                    flag=-1;
                    fr[j]=ref[i];
                    break;
                }
            }
            if(flag==-1)//if there is no demand, directly replaced, no need to check other pages in the frames.
            break;
            else if(flag==1 && k>max)
            {
                max=k;
                x=j;
            }
        }
        max=-1; //reset max for other iterations
        if(flag!=-1
        {
            fr[x]=ref[i];
        }
        pagefault++;
        printf("%d:Page fault\n",ref[i]);
    }
    else
    {
        printf("%d:No page fault\n",ref[i]);
    }
  }
  printf("Total no of page faults:%d\n",pagefault);
}
```

(c) LRU

```
#include<stdio.h>
int isHit(int fr[], int pg, int m)
{
    int hit=0;
    for(int i=0;i<m;i++)
    {
        if(fr[i]==pg)
        {
            hit=1;
            break;
        }
    }
    return hit;
}
void main()
{
    int i,n,m,k,j,pagefault=0,min=999,x,y,count=0,u=0;
    printf("Enter the length of reference sequence:\n");
    scanf("%d",&n);
    int ref[n];
    printf("Enter the page reference sequence:\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&ref[i]);
    }
    printf("Enter the number of frames:\n");
    scanf("%d",&m);
    int fr[m];
    for(i=0;i<m;i++)
    {
        fr[i]=-1;
```

```
    }
    y=0;
    u=0;
    while(count<m)
    {
        if(isHit(fr,ref[u],m)==0)
        {
            fr[y]=ref[u];
            printf("%d:Page Fault\n",ref[u]);
            y++;
            u++;
            pagefault++;
            count++;


        }
        else
        {
            printf("%d:No page fault\n",ref[u]);
            u++;
        }
    }
    for(i=u;i<n;i++)
    {
        if(isHit(fr,ref[i],m)==0)
        {
            for(j=0;j<m;j++)//for every element in the frames, check which index is the least.
            {
                for(k=i-1;k>=0;k--)//to check which index is the least, for each number in the frame, we need to
start checking from i-1 only.
                {
                    if(fr[j]==ref[k])
                    {
```

```
                    break;
                }
            }
            if(k<min) /*agar pg no ki index min se kam ho, iska matlab ye hai ki uski demand sabse pehele hua tha,
            sirf tabhi min ko update karna*/
            {
                x=j;
                min=k;
            }
        }
        min=999; //reset min for other iterations
        fr[x]=ref[i];
        pagefault++;
        printf("%d:Page fault\n",ref[i]);
    }
    else
    {
        printf("%d:No page fault\n",ref[i]);
    }
  }
  printf("Total number page faults:%d\n",pagefault);
}
```

2.11.3 Output:
(a) FIFO:

```
Enter the length of reference sequence:
20
Enter the page reference sequence:
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
Enter the number of frames:
4
7:Page Fault
0:Page Fault
1:Page Fault
2:Page Fault
0:No page fault
3:Page Fault
0:No page fault
4:Page Fault
2:No page fault
3:No page fault
0:Page Fault
3:No page fault
2:No page fault
1:Page Fault
2:Page Fault
0:No page fault
1:No page fault
7:Page Fault
0:No page fault
1:No page fault
Total number of page faults:10
```

(b) OPTIMAL:

```
Enter the length of reference sequence:
20
Enter the page reference sequence:
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
Enter the number of frames:
4
7:Page fault
0:Page fault
1:Page fault
2:Page fault
0:No page fault
3:Page fault
0:No page fault
4:Page fault
2:No page fault
3:No page fault
0:No page fault
3:No page fault
2:No page fault
1:Page fault
2:No page fault
0:No page fault
1:No page fault
7:Page fault
0:No page fault
1:No page fault
Total no of page faults:8
```

(c) LRU:

```
Enter the length of reference sequence:
20
Enter the page reference sequence:
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
Enter the number of frames:
4
7:Page Fault
0:Page Fault
1:Page Fault
2:Page Fault
0:No page fault
3:Page fault
0:No page fault
4:Page fault
2:No page fault
3:No page fault
0:No page fault
3:No page fault
2:No page fault
1:Page fault
2:No page fault
0:No page fault
1:No page fault
7:Page fault
0:No page fault
1:No page fault
Total number page faults:8
```

2.12 Experiment - 12

2.12.1 Question:

Write a C program to simulate disk scheduling algorithms:

(a) FCFS

(b) SCAN

(c) c-SCAN

2.12.2 Code:

(a) FCFS:

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
   int tr,n,total=0,curr;
   printf("Enter the total no of tracks in the disk:\n");
   scanf("%d",&tr);
   printf("Enter the number of requests in the request queue:\n");
   scanf("%d",&n);
   int arr[n];
   printf("Enter the request sequence:\n");
   for(int i=0;i<n;i++)
   {
      scanf("%d",&arr[i]);
   }
   printf("Enter the current head positon of the disk arm:\n");
   scanf("%d",&curr);
   for(int i=0;i<n;i++)
   {
      printf("The head moves from track %d to %d with seek time %d units\n",curr,arr[i], abs(arr[i]-curr));
      total+=abs(arr[i]-curr);
      curr=arr[i];
   }
   printf("The total head movements using FCFS scheduling are:%d\n",total);
}
```

(b) SCAN:

```
#include<stdio.h>
#include<stdlib.h>
void sortAsc(int arr[], int s,int e)
{
   int temp;
   for(int i=s;i<e-1;i++)
   {
      for(int j=s;j<e-i-1;j++)
      {
         if(arr[j]>arr[j+1])
         {
            temp=arr[j];
            arr[j]=arr[j+1];
            arr[j+1]=temp;
```

```
            }
        }
    }
}
void sortDesc(int arr[], int s,int e)
{
    int temp;
    for(int i=s;i<e-1;i++)
    {
        for(int j=s;j<e-i-1;j++)
        {
            if(arr[j]<arr[j+1])
            {
                temp=arr[j];
                arr[j]=arr[j+1];
                arr[j+1]=temp;
            }
        }
    }
}
void main()
{
    int  tr,n,total=0,curr,dir,min,max,i,j,k;
    printf("Enter the total no of tracks in the disk:\n");
    scanf("%d",&tr);
    printf("Enter the number of requests in the request queue:\n");
    scanf("%d",&n);
    int arr[n],seek[n+1];
    printf("Enter the request sequence:\n");
    for(int i=0;i<n;i++)
    {
        scanf("%d",&arr[i]);
    }
    printf("Enter the current head positon of the disk arm:\n");
    scanf("%d",&curr);

    printf("Enter head movement direction(1 for High and 0 for Low):\n");
    scanf("%d",&dir);
    switch(dir)
    {
        case 1:
        //disk fulfills all the higher requests first, so the head reaches the higher end of disk and then changes direction.
        //That is why, we need to find the lower most request track.
        min=arr[0];
        for(i=1;i<n;i++)
        {
            if(arr[i]<min)
            min=arr[i];
        }
        for(i=0;i<=n;i++)
        {
```

```
        seek[i]=arr[i];
      }
      seek[n]=curr;
      printf("Seek sequence:\n");
      sortDesc(seek,0,n+1); //sort in descending order
      for(i=0;i<=n;i++)
      {
        if(seek[i]==curr)
        k=i;
      }
      sortAsc(seek,0,k);

      for(i=0;i<=n;i++)
      {
        printf("%d ",seek[i]);
      }
      printf("\n");
      total=(tr-1-curr)+(tr-1-min);
      printf("Total head movements using SCAN scheduling are:%d\n",total);
      break;

      case 0:
      //disk fulfills all the lower requests first, so the head reaches the lower end of disk and then changes direction.
      //That is why, we need to find the max request track.
      max=arr[0];
      for(i=1;i<n;i++)
      {
        if(arr[i]>max)
        max=arr[i];
      }
      total=(curr-0)+(max-0);//0 is the lower most track
      printf("Total head movements using SCAN scheduling are:%d\n",total);
      break;

      default:
      printf("Invalid choice:\n");
  }
}
```

(c) c-SCAN:

```
#include<stdio.h>

#include<stdlib.h>

#include<limits.h>

void main()

{

  int tr,n,total=0,curr,dir,min,max;

  printf("Enter the total no of tracks in the disk:\n");
```

```
    scanf("%d",&tr);
    printf("Enter the number of requests in the request queue:\n");
    scanf("%d",&n);
    int arr[n];
    printf("Enter the request sequence:\n");
    for(int i=0;i<n;i++)
    {
        scanf("%d",&arr[i]);
    }
    printf("Enter the current head positon of the disk arm:\n");
    scanf("%d",&curr);
    printf("Enter head movement direction(1 for High and 0 for Low):\n");
    scanf("%d",&dir);
    switch(dir)
    {
        case 1:
        //head first moves to the higher end of disk while the disk fulfills all the higher requests, changes 
direction to
        //reach the lower end when th disk does not fulill any lower request. After reaching the lower end, the 
head again
        //changes direction when the disk starts fulfilling lower requests.
        //So we need to find max request less than curr head position

        max=INT_MIN;
        for(int i=0;i<n;i++)
        {
            if(arr[i]<curr && arr[i]>max)
            {
                max=arr[i];
            }
        }
        total=(tr-1-curr)+(tr-1-0)+(max-0);
        printf("The total head movements using C-SCAN scheduling are:%d\n",total);
```

```
        break;


        case 0:
        //reverse of case 1


        min=INT_MAX;
        for(int i=0;i<n;i++)
        {
            if(arr[i]>50 && arr[i]<min)
            {
                min=arr[i];
            }
        }
        printf("Min:%d\n",min);
        total=(curr-0)+(tr-1-0)+(tr-1-min);
        printf("The total head movements using C-SCAN scheduling are:%d\n",total);
        break;


        default:
        printf("Invalid choice!\n");
    }
}
```

2.12.3 Output:

(a) FCFS:

```
Enter the total no of tracks in the disk:
200
Enter the number of requests in the request queue:
7
Enter the request sequence:
82 170 43 140 24 16 190
Enter the current head positon of the disk arm:
50
The total head movements are using FCFS scheduling:642
```

(b) SCAN:

```
Enter the total no of tracks in the disk:
200
Enter the number of requests in the request queue:
7
Enter the request sequence:
82 170 43 140 24 16 190
Enter the current head positon of the disk arm:
50
Enter head movement direction(1 for High and 0 for Low):
1
Seek sequence:
82 140 170 190 50 43 24 16
Total head movements using SCAN scheduling are:332
```

(c) C-SCAN:

```
Enter the total no of tracks in the disk:
200
Enter the number of requests in the request queue:
7
Enter the request sequence:
82 170 43 140 24 16 190
Enter the current head positon of the disk arm:
50
Enter head movement direction(1 for High and 0 for Low):
1
The total head movements using C-SCAN scheduling are:391
```

2.13 Experiment - 13

2.13.1 Question:

Write a C program to simulate disk scheduling algorithms:

(a) SSTF

(b) LOOK

(c) C-LOOK

2.13.2 Code:

(a) SSTF:

```
#include<stdio.h>
#include<stdlib.h>
#include<limits.h>
void main()
{
   int tr,n,total=0,curr,min,count=0,d,ind,i,j=0;
   printf("Enter the total no of tracks in the disk:\n");
   scanf("%d",&tr);
   printf("Enter the number of requests in the request queue:\n");
   scanf("%d",&n);
   int arr[n],seek[n];
   printf("Enter the request sequence:\n");
   for(int i=0;i<n;i++)
   {
      scanf("%d",&arr[i]);
   }
   printf("Enter the current head positon of the disk arm:\n");
   scanf("%d",&curr);
   while(count!=n)
   {
      min=1000;
      for(i=0;i<n;i++)
      {
         if(abs(arr[i]-curr)<min)
         {
            min=abs(arr[i]-curr);
            ind=i;
         }
      }
      seek[j]=arr[ind];
      total+=min;
      curr=arr[ind];
      arr[ind]=1000;
      count++;
      j++;
   }
   printf("Safe sequence is:\n");
   for(i=0;i<n;i++)
   {
      printf("%d ",seek[i]);
```

```
    }
    printf("\n");
    printf("Total number of movements using SSTF are:%d\n",total);
}
```

(b) LOOK:

```
#include<stdio.h>

#include<stdlib.h>

void main()

{

    int tr,n,total=0,curr,min,max,i,j=0,index;

    printf("Enter the total no of tracks in the disk:\n");

    scanf("%d",&tr);

    printf("Enter the number of requests in the request queue:\n");

    scanf("%d",&n);

    int arr[n],seek[n];

    printf("Enter the request sequence:\n");

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

    {

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

    }

    printf("Enter the current head positon of the disk arm:\n");

    scanf("%d",&curr);

    //direction considered- towards larger values first

    max=arr[0];

    min=arr[0];

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

    {

        if(arr[i]>max)

        {

            max=arr[i];

        }

        if(arr[i]<min)

        {

            min=arr[i];
```

```
        }
    }
    total=(max-curr)+(max-min);
    printf("Total number of movements using LOOK scheduling:%d\n",total);
}
```

(c) c-LOOK:

```
#include<stdio.h>
#include<stdlib.h>
#include<limits.h>
void main()
{
    int  tr,n,total=0,curr,min,sec_max,i,j=0,index,max;
    printf("Enter the total no of tracks in the disk:\n");
    scanf("%d",&tr);
    printf("Enter the number of requests in the request queue:\n");
    scanf("%d",&n);
    int arr[n],seek[n];
    printf("Enter the request sequence:\n");
    for(int i=0;i<n;i++)
    {
        scanf("%d",&arr[i]);
    }
    printf("Enter the current head positon of the disk arm:\n");
    scanf("%d",&curr);
    //direction considered- towards larger values first
    max=arr[0];
    min=arr[0];
    for(i=1;i<n;i++)
    {
        if(arr[i]>max)
        {
            max=arr[i];
```

```
        }
        if(arr[i]<min)
        {
            min=arr[i];
        }
    }
    sec_max=INT_MIN;
    for(i=0;i<n;i++)
    {
        if(arr[i]<curr && arr[i]>sec_max)
        {
            sec_max=arr[i];
        }
    }
    printf("sec_max:%d\n",sec_max);
    total=(max-curr)+(max-min)+(sec_max-min);

    printf("Total number of movements using C-LOOK scheduling are:%d\n",total);
}
```

2.13.3 Output:

(a) SSTF:

```
Enter the total no of tracks in the disk:
200
Enter the number of requests in the request queue:
7
Enter the request sequence:
82 170 43 140 24 16 190
Enter the current head positon of the disk arm:
50
Safe sequence is:
43 24 16 82 140 170 190
Total number of movements using SSTF scheduling are:208
```

(b) LOOK:

```
Enter the total no of tracks in the disk:
200
Enter the number of requests in the request queue:
7
Enter the request sequence:
82 170 43 140 24 16 190
Enter the current head positon of the disk arm:
50
Total number of movements using LOOK scheduling are:314
```

(c) c-LOOK:

```
Enter the total no of tracks in the disk:
200
Enter the number of requests in the request queue:
7
Enter the request sequence:
82 170 43 140 24 16 190
Enter the current head positon of the disk arm:
50
sec_max:43
Total number of movements using C-LOOK scheduling are:341
```