VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Operating Systems Lab

Submitted by

DHANUSH H V (1BM21CS052)

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
May-2023 to July-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 Lab**" carried out by **DHANUSH H V (1BM21CS052),** who is bonafide student of **B.M.S. College of Engineering.** 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 academic semester June-2023 to October-2023. The Lab report has been approved as it satisfies the academic requirements in respect of **Operating Systems Lab (22CS4PCOPS)** work prescribed for the said degree.

Name of the Lab-In charge: Dr.Nandhini Vineeth

Professor
Department of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak

Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

9	**Write a C program to simulate the following contiguous memory allocation techniques** a) Worst-fit b) Best-fit c) First-fit	31 - 36
10	Write a C program to simulate paging technique of memory management.	37 - 39
11	Write a C program to simulate page replacement algorithms a) FIFO b) LRU c) Optimal	40 - 43
12	**Write a C program to simulate disk scheduling algorithms** a) FCFS b) SCAN c) C-SCAN	
13	**Write a C program to simulate disk scheduling algorithms** a) SSTF b) LOOK c) c-LOOK	

Course Outcome

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.

Q. Write a C program to simulate the following non-preemptive CPU scheduling algorithm to find Turnaround time and waiting time.

- FCFS

- SJF (preemptive & Non- preemptive)

FCFS

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

int main()
{
   int n, i;
   float waitingTime, turnAroundTime;
   printf("Enter the number of processes: ");
   scanf("%d", &n);
   float *bt = (float *)malloc(n * sizeof(float));
   float *wt = (float *)malloc(n * sizeof(float));
   float *tt = (float *)malloc(n * sizeof(float));
   printf("Enter the burst times of %d processes: ", n);
   for (i = 0; i < n; i++)
   {
      scanf("%f", &bt[i]);
   }
   printf("\nThe details of the processes are as below:\nProcess\tBurst Time\tTurn Around Time\tWaiting Time\n");
   for (i = 0; i < n; i++)
   {
      if (i == 0)
      {
         wt[0] = 0;
      }
      else
      {
         wt[i] = bt[i - 1] + wt[i - 1];
      }
      tt[i] = bt[i] + wt[i];
      printf("%d\t%f\t%f\t%f\n", i + 1, bt[i], tt[i], wt[i]);
      waitingTime += wt[i];
```

```
        turnAroundTime += tt[i];
    }
    printf("The average waiting time is: %f", waitingTime/(float)n);
    printf("\nThe average turn around time is: %f", turnAroundTime / n);
    return 0;
}
```

Output:

```
PS D:\BMS study\SEM IV\OS\All codes\OS_LAB> cd 'd:\BMS study\SEM IV\OS\All codes\OS_LAB\output'
PS D:\BMS study\SEM IV\OS\All codes\OS_LAB\output> & .\'fcfs_sheduling.exe'
Enter the number of processes: 3
Enter the burst times of 3 processes: 12 5 7

The details of the processes are as below:
Process Burst Time      Turn Around Time        Waiting Time
1       12.000000       12.000000       0.000000
2       5.000000        17.000000       12.000000
3       7.000000        24.000000       17.000000
The average waiting time is: 9.666667
The average turn around time is: 17.666666
```

SJF (Non-Preemptive)

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

int main()
{
   int n,i,j,index;
   float WT,TurnAroundTime,temp;
   printf("Enter the number of processes: ");
   scanf("%d", &n);
   float *bt = (float *)malloc(n * sizeof(float));
   float *wt = (float *)malloc(n * sizeof(float));
   float *tt = (float *)malloc(n * sizeof(float));
   printf("Enter the burst times of %d processes: ", n);
   for (i = 0; i < n; i++)
   {
      scanf("%f", &bt[i]);
   }

   for(i = 0; i < n-1; i++){
      // index = i;
      for(j=0; j < n-i-1; j++){

         if(bt[j]>bt[j+1]){
            temp = bt[j];
            bt[j] = bt[j+1];
            bt[j+1] = temp;
         }



      }
   }



 printf("\nThe details of the processes are as below:\nProcess\tBurst Time\tTurn Around Time\tWaiting Time\n");
   for (i = 0; i < n; i++)
```

```
    {
        if (i == 0)
        {
            wt[0] = 0;
        }
        else
        {
            wt[i] = bt[i - 1] + wt[i - 1];
        }
        tt[i] = bt[i] + wt[i];
        printf("%d\t%f\t%f\t\t%f\n", i + 1, bt[i], tt[i], wt[i]);
        WT = WT + wt[i];
        TurnAroundTime = TurnAroundTime + tt[i];
    }
    printf("The average waiting time is: %f", WT/(float)n);
    printf("\nThe average turn around time is: %f", TurnAroundTime/n);
    return 0;
}
```

Output:

```
Enter the number of processes: 3
Enter the burst times of 3 processes: 12 5 7

The details of the processes are as below:
Process Burst Time      Turn Around Time        Waiting Time
1       5.000000        5.000000                0.000000
2       7.000000        12.000000               5.000000
3       12.000000       24.000000               12.000000
The average waiting time is: 5.666667
The average turn around time is: 13.666667
```

SJF (Pre-Emptive)

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

struct Process
{
   int pid;
   int bt;
   int art;
};

void findWaitingTime(struct Process proc[], int n, int wt[])
{
   int rt[n];
   for (int i = 0; i < n; i++)
   {
      rt[i] = proc[i].bt;
   }

   int complete = 0, t = 0, minm = 99999;
   int shortest = 0, finish_time;
   bool check = false;

   while (complete != n)
   {

      for (int j = 0; j < n; j++)
      {
         if ((proc[j].art <= t) &&
            (rt[j] < minm) && rt[j] > 0)
         {
            minm = rt[j];
            shortest = j;
            check = true;
         }
      }

      if (check == false)
      {
         t++;
         continue;
      }

      rt[shortest]--;
      minm = rt[shortest];
```

```
        if (minm == 0)
            minm = 99999;

        if (rt[shortest] == 0)
        {

            complete++;
            check = false;
            finish_time = t + 1;
            wt[shortest] = finish_time - proc[shortest].bt - proc[shortest].art;
            if (wt[shortest] < 0)
                wt[shortest] = 0;
        }
        t++;
    }
}

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

void findavgTime(struct Process proc[], int n)
{
    int wt[n], tat[n], total_wt = 0, total_tat = 0;
    findWaitingTime(proc, n, wt);
    findTurnAroundTime(proc, n, wt, tat);
    printf("Processes\tBurst time\tWaiting time\tTurn around time\n");

    for (int i = 0; i < n; i++)
    {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        printf(" %d\t\t%d\t\t%d\t\t%d\t\t\n", proc[i].pid, proc[i].bt, wt[i], tat[i]);
    }

    printf("Average waiting time = %f", (float)total_wt / (float)n);
    printf("\nAverage turn around time = %f", (float)total_tat / (float)n);
}

int main()
{
```

```
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process proc[n];

    printf("Enter the burst times and arrival times of %d processes: ", n);
    for (int i = 0; i < n; i++)
    {
        scanf("%d %d", &proc[i].bt, &proc[i].art);
        proc[i].pid = i + 1;
    }

    findavgTime(proc, n);
    return 0;
}
```

Output:

```
Enter the number of processes: 3
Enter the burst times and arrival times of 3 processes: 12 5
4 0
6 7
Processes       Burst time      Waiting time    Turn around time
 1              12              6               18
 2              4               0               4
 3              6               0               6
Average waiting time = 2.000000
Average turn around time = 9.333333
```

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

- Priority

- Round Robin

Priority

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

struct process {
   int proc_id;
   int bt;
   int priority;
   int wt;
   int tat;
};

void find_wt(struct process[], int, int[]);
void find_tat(struct process[], int, int[], int[]);

void find_average_time(struct process[], int);

void priority_scheduling(struct process[], int);

int main()
{
   int n, i;
   struct process proc[10];

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

for(i = 0; i< n; i++)
   {
printf("\nEnter the process ID: ");
scanf("%d", &proc[i].proc_id);

printf("Enter the burst time: ");
```

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

printf("Enter the priority: ");
scanf("%d", &proc[i].priority);
   }

priority_scheduling(proc, n);
   return 0;
}

void find_wt(struct process proc[], int n, int wt[])
{
   int i;
   wt[0] = 0;

for(i = 1; i< n; i++)
   {
   wt[i] = proc[i - 1].bt + wt[i - 1];
   }
}

void find_tat(struct process proc[], int n, int wt[], int tat[])
{
   int i;
for(i = 0; i< n; i++)
   {
tat[i] = proc[i].bt + wt[i];
   }
}

void find_average_time(struct process proc[], int n)
{
   int wt[10], tat[10], total_wt = 0, total_tat = 0, i;

find_wt(proc, n, wt);
find_tat(proc, n, wt, tat);

printf("\nProcess ID\tBurst Time\tPriority\tWaiting Time\tTurnaround Time");

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

```
    {
total_wt = total_wt + wt[i];
total_tat = total_tat + tat[i];
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", proc[i].proc_id, proc[i].bt, proc[i].priority, wt[i],
tat[i]);
    }

printf("\n\nAverage Waiting Time = %f", (float)total_wt/n);
printf("\nAverage Turnaround Time = %f\n", (float)total_tat/n);
}

void priority_scheduling(struct process proc[], int n)
{
int i, j, pos;
struct process temp;
for(i = 0; i< n; i++)
{
    pos = i;
for(j = i + 1; j < n; j++)
    {
        if(proc[j].priority< proc[pos].priority)
            pos = j;
    }

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

find_average_time(proc, n);
}
```

Output:

```
Enter the number of processes: 3

Enter the process ID: 1
Enter the burst time: 3
Enter the priority: 2

Enter the process ID: 2
Enter the burst time: 4
Enter the priority: 1

Enter the process ID: 3
Enter the burst time: 2
Enter the priority: 3

Process ID      Burst Time      Priority        Waiting Time    Turnaround Time
2               4               1               0               4
1               3               2               4               7
3               2               3               7               9

Average Waiting Time = 3.666667
Average Turnaround Time = 6.666667
```

Round Robin

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

struct process {
   int proc_id;
   int bt;
   int priority;
   int wt;
   int tat;
};

void find_wt(struct process[], int, int[]);
void find_tat(struct process[], int, int[], int[]);

void find_average_time(struct process[], int);

void priority_scheduling(struct process[], int);

int main()
{
   int n, i;
   struct process proc[10];

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

for(i = 0; i< n; i++)
   {
printf("\nEnter the process ID: ");
scanf("%d", &proc[i].proc_id);

printf("Enter the burst time: ");
scanf("%d", &proc[i].bt);

printf("Enter the priority: ");
scanf("%d", &proc[i].priority);
   }
```

```
priority_scheduling(proc, n);
   return 0;
}

void find_wt(struct process proc[], int n, int wt[])
{
   int i;
   wt[0] = 0;

for(i = 1; i< n; i++)
   {
   wt[i] = proc[i - 1].bt + wt[i - 1];
   }
}

void find_tat(struct process proc[], int n, int wt[], int tat[])
{
   int i;
for(i = 0; i< n; i++)
   {
tat[i] = proc[i].bt + wt[i];
   }
}

void find_average_time(struct process proc[], int n)
{
   int wt[10], tat[10], total_wt = 0, total_tat = 0, i;

find_wt(proc, n, wt);
find_tat(proc, n, wt, tat);

printf("\nProcess ID\tBurst Time\tPriority\tWaiting Time\tTurnaround Time");

for(i = 0; i< n; i++)
   {
total_wt = total_wt + wt[i];
total_tat = total_tat + tat[i];
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", proc[i].proc_id, proc[i].bt, proc[i].priority, wt[i],
tat[i]);
   }
```

```
printf("\n\nAverage Waiting Time = %f", (float)total_wt/n);
printf("\nAverage Turnaround Time = %f\n", (float)total_tat/n);
}

void priority_scheduling(struct process proc[], int n)
{
int i, j, pos;
struct process temp;
for(i = 0; i< n; i++)
{
   pos = i;
for(j = i + 1; j < n; j++)
   {
      if(proc[j].priority< proc[pos].priority)
         pos = j;
   }

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

find_average_time(proc, n);
}
```

Output:

```
Enter number of processes: 3

Enter the Arrival and Burst time of the Process[1]: 2 12

Enter the Arrival and Burst time of the Process[2]: 0 3

Enter the Arrival and Burst time of the Process[3]: 7 8
Enter the time quantum for the process:         2

 Process No              Burst Time              TAT             Waiting Time
Process No[2]            3                       7                       4
Process No[3]            8                       14                      6
Process No[1]            12                      21                      9
 Average Turn Around Time:      6.333333
 Average Waiting Time:  14.000000
```

Q. Write a 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. Use FCFS scheduling for the processes in each queue.

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

int main()
{
   int i,temp,n;
   float wtavg,tatavg;



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

   int pid[n],bt[n],su[n],wt[n],tat[n];



   for (i = 0; i < n; i++)
   {
      pid[i] = i;
      printf("\nEnter the burst time of Process %d :",i+1);
      scanf("%d",&bt[i]);

      printf("For a System Process(0) Else if its a User Process(1):");
      scanf("%d",&su[i]);

   }


   wtavg = wt[0] = 0;
   tatavg = tat[0] = bt[0];


   for(int i=0;i<n-1;i++)
   {
```

```
    for(int j=i+1;j<n;j++)
    {

        if(su[i]>su[j])
        {
            temp = pid[i];
            pid[i] = pid[j];
            pid[j] = temp;

            temp = bt[i];
            bt[i] = bt[j];
            bt[j] = temp;

            temp = su[i];
            su[i] = su[j];
            su[j] = temp;
        }



    }




}

for(i=1;i<n;i++)
{
    wt[i] = wt[i-1] + bt[i-1];
    tat[i] = tat[i-1] + bt[i];
    wtavg += wt[i];
    tatavg += tat[i];
}



printf("\nProcess-ID \t System/User Process \t\t Burst Time \t\t Waiting Time \t\t TAT ");

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

```
        printf("\n%d \t\t\t %d \t\t\t %d \t\t\t %d \t\t\t %d",pid[i]+1,su[i],bt[i],wt[i],tat[i]);
    }

    printf("\nAverage Waiting Time:%0.3f",wtavg/n);
    printf("\nAverage TurnAroundTime:%0.3f",1.0*tatavg/n);


    return 0;
}
```

Output:

```
Enter number of processes: 3

Enter the burst time of Process 1 :12
For a System Process(0) Else if its a User Process(1):0

Enter the burst time of Process 2 :24
For a System Process(0) Else if its a User Process(1):1

Enter the burst time of Process 3 :5
For a System Process(0) Else if its a User Process(1):0

Process-ID      System/User Process             Burst Time              Waiting Time            TAT
1                       0                       12                      0                       12
3                       0                       5                       12                      17
2                       1                       24                      17                      41
Average Waiting Time:9.667
Average TurnAroundTime:23.333
```

Q. Write a C program to simulate Real-Time CPU scheduling algorithms:

a) **Rate Monotonic**
b) **Earliest Deadline First**
c) **Proportional scheduling**

Rate Monotonic

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

int main()
{
   int i,temp,n;
   float wtavg,tatavg;



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

   int pid[n],bt[n],su[n],wt[n],tat[n];



   for (i = 0; i < n; i++)
   {
      pid[i] = i;
      printf("\nEnter the burst time of Process %d :",i+1);
      scanf("%d",&bt[i]);

      printf("For a System Process(0) Else if its a User Process(1):");
      scanf("%d",&su[i]);

   }


   wtavg = wt[0] = 0;
   tatavg = tat[0] = bt[0];
```

```
for(int i=0;i<n-1;i++)
{
   for(int j=i+1;j<n;j++)
   {

      if(su[i]>su[j])
      {
         temp = pid[i];
         pid[i] = pid[j];
         pid[j] = temp;

         temp = bt[i];
         bt[i] = bt[j];
         bt[j] = temp;

         temp = su[i];
         su[i] = su[j];
         su[j] = temp;
      }



   }




}

for(i=1;i<n;i++)
{
   wt[i] = wt[i-1] + bt[i-1];
   tat[i] = tat[i-1] + bt[i];
   wtavg += wt[i];
   tatavg += tat[i];
}
```

```
    printf("\nProcess-ID \t System/User Process \t\t Burst Time \t\t Waiting Time \t\t TAT ");

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


       printf("\n%d \t\t\t %d \t\t\t %d \t\t\t %d \t\t\t %d",pid[i]+1,su[i],bt[i],wt[i],tat[i]);
    }

    printf("\nAverage Waiting Time:%0.3f",wtavg/n);
    printf("\nAverage TurnAroundTime:%0.3f",1.0*tatavg/n);


    return 0;
}
```

Output:

```
Enter the number of tasks: 3
Enter the execution time and period for task 1: 1 10
Enter the execution time and period for task 2: 2 5
Enter the execution time and period for task 3: 3 20

Task2 -> starts:0.000, ends:2.000, execution time:2.000, period:5.000
Task1 -> starts:2.000, ends:3.000, execution time:1.000, period:10.000
Task3 -> starts:3.000, ends:6.000, execution time:3.000, period:20.000
```

Earliest Deadline First

```
#include <stdio.h>
#include<stdlib.h>
#define arrival                 0
#define execution       1
#define deadline        2
#define period                  3
#define abs_arrival             4
#define execution_copy  5
#define abs_deadline    6


typedef struct
{
    int T[7],instance,alive;

}task;

#define IDLE_TASK_ID 1023
#define ALL 1
#define CURRENT 0

void get_tasks(task *t1,int n);
int hyperperiod_calc(task *t1,int n);
float cpu_util(task *t1,int n);
int gcd(int a, int b);
int lcm(int *a, int n);
int sp_interrupt(task *t1,int tmr,int n);
int min(task *t1,int n,int p);
void update_abs_arrival(task *t1,int n,int k,int all);
void update_abs_deadline(task *t1,int n,int all);
void copy_execution_time(task *t1,int n,int all);


int timer = 0;

int main()
{
    task *t;
```

```
int n, hyper_period, active_task_id;
float cpu_utilization;
printf("Enter number of tasks:");
scanf("%d", &n);
t = (task*)malloc(n * sizeof(task));
get_tasks(t, n);
cpu_utilization = cpu_util(t, n);
printf("CPU Utilization %f\n", cpu_utilization);

if (cpu_utilization < 1)
        printf("Tasks can be scheduled\n");
else
        printf("Schedule is not feasible\n");

hyper_period = hyperperiod_calc(t, n);
copy_execution_time(t, n, ALL);
update_abs_arrival(t, n, 0, ALL);
update_abs_deadline(t, n, ALL);

while (timer <= hyper_period)
{

        if (sp_interrupt(t, timer, n))
        {
                active_task_id = min(t, n, abs_deadline);
        }

        if (active_task_id == IDLE_TASK_ID)
        {
                printf("%d  Idle\n", timer);
        }

        if (active_task_id != IDLE_TASK_ID)
        {

                if (t[active_task_id].T[execution_copy] != 0)
                {
                        t[active_task_id].T[execution_copy]--;
                        printf("%d  Task %d\n", timer, active_task_id + 1);
                }
```

```
                    if (t[active_task_id].T[execution_copy] == 0)
                    {
                        t[active_task_id].instance++;
                        t[active_task_id].alive = 0;
                        copy_execution_time(t, active_task_id, CURRENT);
                        update_abs_arrival(t,  active_task_id,  t[active_task_id].instance,
CURRENT);
                        update_abs_deadline(t, active_task_id, CURRENT);
                        active_task_id = min(t, n, abs_deadline);
                    }
                }
                ++timer;
        }
        free(t);
        return 0;
}
void get_tasks(task *t1, int n)
{
        int i = 0;
        while (i < n)
        {
                printf("Enter Task %d parameters\n", i + 1);
                printf("Arrival time: ");
                scanf("%d", &t1->T[arrival]);
                printf("Execution time: ");
                scanf("%d", &t1->T[execution]);
                printf("Deadline time: ");
                scanf("%d", &t1->T[deadline]);
                printf("Period: ");
                scanf("%d", &t1->T[period]);
                t1->T[abs_arrival] = 0;
                t1->T[execution_copy] = 0;
                t1->T[abs_deadline] = 0;
                t1->instance = 0;
                t1->alive = 0;
                t1++;
                i++;
        }
}
```

```
int hyperperiod_calc(task *t1, int n)
{
    int i = 0, ht, a[10];
    while (i < n)

    {
            a[i] = t1->T[period];
            t1++;
            i++;
    }
    ht = lcm(a, n);

    return ht;
}

int gcd(int a, int b)
{
    if (b == 0)
            return a;
    else
            return gcd(b, a % b);
}

int lcm(int *a, int n)
{
    int res = 1, i;
    for (i = 0; i < n; i++)
    {
            res = res * a[i] / gcd(res, a[i]);
    }
    return res;
}

int sp_interrupt(task *t1, int tmr, int n)
{
    int i = 0, n1 = 0, a = 0;
    task *t1_copy;
    t1_copy = t1;
    while (i < n)
```

```
    {
        if (tmr == t1->T[abs_arrival])
        {
            t1->alive = 1;
            a++;
        }
        t1++;
        i++;
    }

    t1 = t1_copy;
    i = 0;

    while (i < n)
    {
        if (t1->alive == 0)
            n1++;
        t1++;
        i++;
    }

    if (n1 == n || a != 0)
    {
        return 1;
    }

    return 0;
}

void update_abs_deadline(task *t1, int n, int all)
{
    int i = 0;
    if (all)
    {
        while (i < n)
        {
            t1->T[abs_deadline] = t1->T[deadline] + t1->T[abs_arrival];
            t1++;
            i++;
        }
```

```
    }
    else
    {
            t1 += n;
            t1->T[abs_deadline] = t1->T[deadline] + t1->T[abs_arrival];
    }
}

void update_abs_arrival(task *t1, int n, int k, int all)
{
    int i = 0;
    if (all)
    {
            while (i < n)
            {
                    t1->T[abs_arrival] = t1->T[arrival] + k * (t1->T[period]);
                    t1++;
                    i++;
            }
    }
    else
    {
            t1 += n;
            t1->T[abs_arrival] = t1->T[arrival] + k * (t1->T[period]);
    }
}

void copy_execution_time(task *t1, int n, int all)
{
    int i = 0;
    if (all)
    {
            while (i < n)
            {
                    t1->T[execution_copy] = t1->T[execution];
                    t1++;
                    i++;
            }
    }
    else
```

```
    {
        t1 += n;
        t1->T[execution_copy] = t1->T[execution];
    }
}

int min(task *t1, int n, int p)
{
    int i = 0, min = 0x7FFF, task_id = IDLE_TASK_ID;
    while (i < n)
    {
        if (min > t1->T[p] && t1->alive == 1)
        {
            min = t1->T[p];
            task_id = i;
        }
        t1++;
        i++;
    }
    return task_id;
}

float cpu_util(task *t1, int n)
{
    int i = 0;
    float cu = 0;
    while (i < n)
    {
        cu = cu + (float)t1->T[execution] / (float)t1->T[deadline];
        t1++;
        i++;
    }
    return cu;
}
```

Output:

```
Enter number of tasks:3
Enter Task 1 parameters
Arrival time: 0
Execution time: 2
Deadline time: 5
Period: 5
Enter Task 2 parameters
Arrival time: 0
Execution time: 3
Deadline time: 8
Period: 8
Enter Task 3 parameters
Arrival time: 0
Execution time: 1
Deadline time: 10
Period: 10
CPU Utilization 0.875000
```

```
Tasks can be scheduled
0  Task 1
1  Task 1
2  Task 2
3  Task 2
4  Task 2
5  Task 1
6  Task 1
7  Task 3
8  Task 2
9  Task 2
10  Task 1
11  Task 1
12  Task 2
13  Task 3
14  Idle
15  Task 1
16  Task 1
17  Task 2
18  Task 2
19  Task 2
20  Task 1
21  Task 1
22  Task 3
23  Idle
24  Task 2
25  Task 1
26  Task 1
27  Task 2
28  Task 2
29  Idle
30  Task 1
31  Task 1
32  Task 2
33  Task 2
34  Task 2
35  Task 1
36  Task 1
37  Task 3
38  Idle
39  Idle
40  Task 1
```

Proportional Scheduling

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

int main() {

    srand(time(0));

    int numbers[5];
    int i;


    for (i = 0; i < 5; i++) {
        numbers[i] = rand() % 10 + 1;
    }

    printf("Initial Numbers: ");
    for (i = 0; i < 5; i++) {
        printf("%d ", numbers[i]);
    }
    printf("\n");

    while (1) {

        int all_zero = 1;
        for (i = 0; i < 5; i++) {
            if (numbers[i] > 0) {
                all_zero = 0;
                break;
            }
        }

        if (all_zero) {
            break;
        }

        int selected_index;
        do {
```

```
            selected_index = rand() % 5;
        } while (numbers[selected_index] == 0);

        numbers[selected_index]--;
        printf("Decrementing number at index %d: ", selected_index);
        for (i = 0; i < 5; i++) {
            printf("%d ", numbers[i]);
        }
        printf("\n");
    }

    printf("All numbers reached 0.\n");

    return 0;
}
```

Output:

```
Initial Numbers: 3 4 6 5 4
Decrementing number at index 4: 3 4 6 5 3
Decrementing number at index 0: 2 4 6 5 3
Decrementing number at index 4: 2 4 6 5 2
Decrementing number at index 2: 2 4 5 5 2
Decrementing number at index 1: 2 3 5 5 2
Decrementing number at index 0: 1 3 5 5 2
Decrementing number at index 2: 1 3 4 5 2
Decrementing number at index 1: 1 2 4 5 2
Decrementing number at index 1: 1 1 4 5 2
Decrementing number at index 2: 1 1 3 5 2
Decrementing number at index 4: 1 1 3 5 1
Decrementing number at index 3: 1 1 3 4 1
Decrementing number at index 2: 1 1 2 4 1
Decrementing number at index 2: 1 1 1 4 1
Decrementing number at index 3: 1 1 1 3 1
Decrementing number at index 0: 0 1 1 3 1
Decrementing number at index 2: 0 1 0 3 1
Decrementing number at index 1: 0 0 0 3 1
Decrementing number at index 4: 0 0 0 3 0
Decrementing number at index 3: 0 0 0 2 0
Decrementing number at index 3: 0 0 0 1 0
Decrementing number at index 3: 0 0 0 0 0
All numbers reached 0.
```

Q. Write a C program to simulate Producer-Consumer problem using semaphores.

```
#include <stdio.h>
#include <semaphore.h>
#include <pthread.h>
#include <stdlib.h>
#include <windows.h>
#include <time.h>

pthread_mutex_t mutex;
sem_t empty, full;
int in=0, out=0, buffer[5];

void *producer(void *pno){
   for(int i=0;i<5;i++){
      sem_wait(&empty);
      pthread_mutex_lock(&mutex);
      int x = rand()%100;
      buffer[in]=x;
      in = (in+1)%5;
      printf("Producer %d has put %d in buffer\n",*((int*)pno), x);
      pthread_mutex_unlock(&mutex);
      sem_post(&full);
   }
}

void *consumer(void* cno){
   for(int i=0;i<5;i++){
      sem_wait(&full);
      pthread_mutex_lock(&mutex);
      int x = buffer[out];
      out = (out+1)%5;
      printf("Comsumer %d has consumed %d\n",*((int*)cno), x);
      pthread_mutex_unlock(&mutex);
      sem_post(&empty);
   }
}

void main(){
```

```
    pthread_t prod[5], con[5];
    sem_init(&empty,0,10);
    sem_init(&full,0,0);
    pthread_mutex_init(&mutex,NULL);

    int a[] = {1,2,3,4,5};

    for(int i=0;i<5;i++){
        pthread_create(&prod[i],NULL,(void*)producer, (void*)&a[i]);
        pthread_create(&con[i],NULL,(void*)consumer, (void*)&a[i]);
    }

    for(int i=0;i<5;i++){
        pthread_join(prod[i],NULL);
        pthread_join(con[i],NULL);
    }
    pthread_mutex_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);
}
```

Output:

```
Producer produces item: 1
Consumer consumes item: 1
Producer produces item: 1
Consumer consumes item: 1
Producer produces item: 1
Consumer consumes item: 1
Producer produces item: 1
Consumer consumes item: 1
Producer produces item: 1
Consumer consumes item: 1
Producer produces item: 1
Consumer consumes item: 1
Producer produces item: 1
```

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

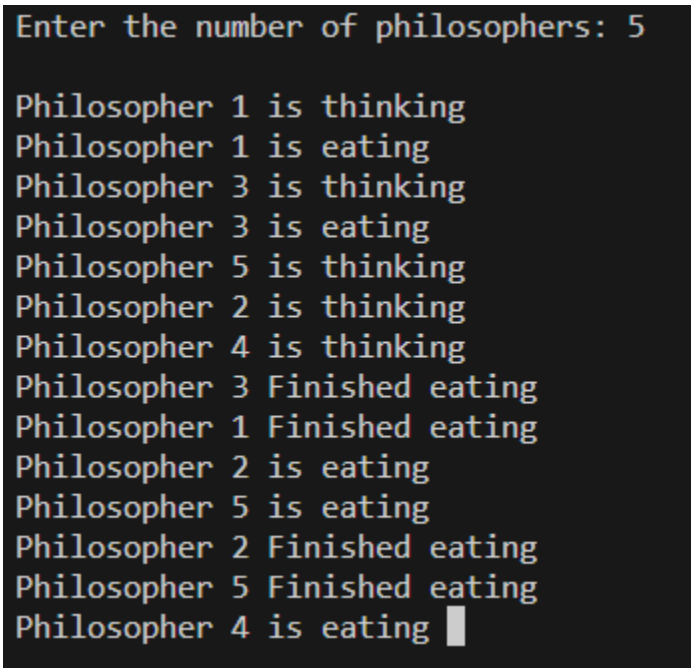
```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>


#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };

sem_t mutex;
sem_t S[N];

void test(int phnum)
{
    if (state[phnum] == HUNGRY
            && state[LEFT] != EATING
            && state[RIGHT] != EATING) {
            // state that eating
            state[phnum] = EATING;

            sleep(2);

            printf("Philosopher %d takes fork %d and %d\n",
                                    phnum + 1, LEFT + 1, phnum + 1);

            printf("Philosopher %d is Eating\n", phnum + 1);

            // sem_post(&S[phnum]) has no effect
            // during takefork
            // used to wake up hungry philosophers
            // during putfork
```

```
            sem_post(&S[phnum]);
        }
    }

    // take up chopsticks
    void take_fork(int phnum)
    {

        sem_wait(&mutex);

        // state that hungry
        state[phnum] = HUNGRY;

        printf("Philosopher %d is Hungry\n", phnum + 1);

        // eat if neighbours are not eating
        test(phnum);

        sem_post(&mutex);

        // if unable to eat wait to be signalled
        sem_wait(&S[phnum]);

        sleep(1);
    }

    // put down chopsticks
    void put_fork(int phnum)
    {

        sem_wait(&mutex);

        // state that thinking
        state[phnum] = THINKING;

        printf("Philosopher %d putting fork %d and %d down\n",
            phnum + 1, LEFT + 1, phnum + 1);
        printf("Philosopher %d is thinking\n", phnum + 1);

        test(LEFT);
```

```
    test(RIGHT);

    sem_post(&mutex);
}

void* philosopher(void* num)
{

    while (1) {

        int* i = num;

        sleep(1);

        take_fork(*i);

        sleep(0);

        put_fork(*i);
    }
}

int main()
{

    int i;
    pthread_t thread_id[N];

    // initialize the semaphores
    sem_init(&mutex, 0, 1);

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

        sem_init(&S[i], 0, 0);

    for (i = 0; i < N; i++) {

        // create philosopher processes
        pthread_create(&thread_id[i], NULL,
                       philosopher, &phil[i]);
```

```
            printf("Philosopher %d is thinking\n", i + 1);
        }

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

            pthread_join(thread_id[i], NULL);
}
```

Output:



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

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


    int n, m, i, j, k;

   printf("Enter the no of Process and Resources:");
   scanf("%d %d",&n,&m);

   int *avail = (int*)malloc(m*sizeof(int));

   printf("Enter the available Resources:");

   for(i=0;i<m;i++){
      scanf("%d",&avail[i]);
   }


    int **alloc = (int**)malloc(n*sizeof(int*));
   printf("Enter the allocation matrix:");
    for(i=0;i<n;i++){
      alloc[i] = (int*)malloc(m*sizeof(int));

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

    int **max = (int**)malloc(n*sizeof(int*));


     printf("Enter the Max matrix:");
    for(i=0;i<n;i++){
      max[i] = (int*)malloc(m*sizeof(int));
```

```
   for(int j=0;j<m;j++){
      scanf("%d",&max[i][j]);
   }
}



 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] - alloc[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] > avail[j]){
                                              flag = 1;
                                              break;
                                     }
                            }

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

```
            }
        }
        int flag = 1;
        for(int i=0;i<n;i++)
        {
        if(f[i]==0)
        {
                flag=0;
                printf("The following system is not safe");
                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", ans[n - 1]);
        }
        return (0);
}
```

Output:

```
Enter the no of Process and Resources:5 3
Enter the available Resources:2 3 0
Enter the allocation matrix:0 1 0
3 0 2
3 0 2
2 1 1
0 0 2
Enter the Max matrix:7 4 3
0 2 0
6 0 0
0 1 1
4 3 1
Following is the SAFE Sequence
 P1 -> P2 -> P3 -> P4 -> P0
```

Q. Write a C program to simulate deadlock detection.

```
#include <stdio.h>

int main()
{
   int i, j, np, nr;

   printf("Enter the number of the process and resources: ");
   scanf("%d%d", &np, &nr);
   int alloc[np][nr], request[np][nr], avail[nr], r[nr], w[nr], mark[np];

   printf("Enter the total amount of each resource available: ");
   for (i = 0; i < nr; i++)
   {
      scanf("%d", &r[i]);
      avail[j] = r[j];
   }

   printf("Enter the request matrix:\n");
   for (i = 0; i < np; i++)
   {
      for (j = 0; j < nr; j++)
      {
         scanf("%d", &request[i][j]);
      }
   }

   printf("Enter the allocation matrix:\n");
   for (i = 0; i < np; i++)
   {
      for (j = 0; j < nr; j++)
      {
         scanf("%d", &alloc[i][j]);
         avail[j] -= alloc[i][j];
      }
   }

   // marking processes with zero allocation
   for (i = 0; i < np; i++)
   {
```

```
    int count = 0;
    for (j = 0; j < nr; j++)
    {
        if (alloc[i][j] == 0)
        {
            count++;
        }
        else
        {
            break;
        }
    }
    if (count == nr)
    {
        mark[i] = 1;
    }
}

// initialize W with avail
for (j = 0; j < nr; j++)
{
    w[j] = avail[j];
}

// mark processes with request less than or equal to W
for (i = 0; i < np; i++)
{
    int canbeprocessed = 0;
    if (mark[i] != 1)
    {
        for (j = 0; j < nr; j++)
        {
            if (request[i][j] <= w[j])
                canbeprocessed = 1;
            else
            {
                canbeprocessed = 0;
                break;
            }
        }
```

```
            if (canbeprocessed)
            {
                mark[i] = 1;

                for (j = 0; j < nr; j++)
                {
                    w[j] += alloc[i][j];
                }
            }
        }
    }

    // checking for unmarked processes
    int deadlock = 0;
    for (i = 0; i < np; i++)
    {
        if (mark[i] != 1)
        {
            deadlock = 1;
        }
    }

    if (deadlock)
    {
        printf("\nDeadlock detected\n");
    }
    else
    {
        printf("\nNo Deadlock detected\n");
    }
}
```

Output:

```
Enter the number of the process and resources: 3 3
Enter the total amount of each resource available: 1 2 4
Enter the request matrix:
1 0 2
2 0 9
1 1 0
Enter the allocation matrix:
0 0 1
1 3 6
9 5 1

No Deadlock detected
```

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

a) Worst Fit

b) Best Fit

c) First Fit

Worst fit

```
#include <stdio.h>

void worstFit(int blockSize[], int m, int processSize[], int n)
{
   int allocation[n];
   memset(allocation, -1, sizeof(allocation));

   for (int i = 0; i < n; i++)
   {
      int wstIdx = -1;
      for (int j = 0; j < m; j++)
      {
         if (blockSize[j] >= processSize[i])
         {
            if (wstIdx == -1)
               wstIdx = j;
            else if (blockSize[wstIdx] < blockSize[j])
               wstIdx = j;
         }
      }

      if (wstIdx != -1)
      {
         allocation[i] = wstIdx;
         blockSize[wstIdx] -= processSize[i];
      }
   }

   printf("\nProcess No.\tProcess Size\tBlock no.\n");
   for (int i = 0; i < n; i++)
   {
      printf("   %d\t\t%d\t\t", i + 1, processSize[i]);
      if (allocation[i] != -1)
         printf("%d", allocation[i] + 1);
      else
         printf("Not Allocated");
      printf("\n");
```

```
    }
}

int main()
{
    printf("Enter the number of blocks: ");
    int m;
    scanf("%d", &m);
    int blockSize[m];
    printf("Enter the block sizes: ");
    for (int i = 0; i < m; i++)
        scanf("%d", &blockSize[i]);

    printf("Enter the number of processes: ");
    int n;
    scanf("%d", &n);
    int processSize[n];
    printf("Enter the process sizes: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &processSize[i]);

    worstFit(blockSize, m, processSize, n);

    return 0;
}
```

Output:

```
Enter the number of blocks: 5
Enter the block sizes: 100 500 200 300 600
Enter the number of processes: 4
Enter the process sizes: 212 417 112 426
The memory allocation is as:
Process-1:  212  5
Process-2:  417  2
Process-3:  112  5
Process-4:  426  Not Allocated
```

Best Fit

```
#include <stdio.h>

void bestFit(int blockSize[], int m, int processSize[], int n)
{
    int allocation[n];

    for (int i = 0; i < n; i++)
        allocation[i] = -1;

    for (int i = 0; i < n; i++)
    {
        int bestIdx = -1;

        for (int j = 0; j < m; j++)
        {
            if (blockSize[j] >= processSize[i])
            {
                if (bestIdx == -1)
                    bestIdx = j;
                else if (blockSize[bestIdx] > blockSize[j])
                    bestIdx = j;
            }
        }

        if (bestIdx != -1)
        {
            allocation[i] = bestIdx;
            blockSize[bestIdx] -= processSize[i];
        }
    }

    printf("\nProcess No.\tProcess Size\tBlock no.\n");
    for (int i = 0; i < n; i++)
    {
        printf(" %d \t\t %d \t\t", i + 1, processSize[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
        printf("\n");
    }
}

int main()
```

```
{
    printf("Enter the number of blocks: ");
    int m;
    scanf("%d", &m);
    int blockSize[m];
    printf("Enter the block sizes: ");
    for (int i = 0; i < m; i++)
        scanf("%d", &blockSize[i]);

    printf("Enter the number of processes: ");
    int n;
    scanf("%d", &n);
    int processSize[n];
    printf("Enter the process sizes: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &processSize[i]);

    bestFit(blockSize, m, processSize, n);

    return 0;
}
```

Output:

```
Enter the number of blocks: 5
Enter the block sizes: 100 500 200 300 600
Enter the number of processes: 5
Enter the process sizes: 212 417 112 426 121

Process No.     Process Size    Block no.
 1               212            4

 2               417            2

 3               112            3

 4               426            5

 5               121            5
```

First Fit

```
#include <stdio.h>

void firstFit(int blockSize[], int m, int processSize[], int n)
{
   int i, j;
   int allocation[n];

   for (i = 0; i < n; i++)
   {
      allocation[i] = -1;
   }

   for (i = 0; i < n; i++)
   {
      for (j = 0; j < m; j++)
      {
         if (blockSize[j] >= processSize[i])
         {
            allocation[i] = j;
            blockSize[j] -= processSize[i];
            break;
         }
      }
   }

   printf("\nProcess No.\tProcess Size\tBlock no.\n");
   for (int i = 0; i < n; i++)
   {
      printf(" %i\t\t\t", i + 1);
      printf("%i\t\t\t\t", processSize[i]);
      if (allocation[i] != -1)
         printf("%i", allocation[i] + 1);
      else
         printf("Not Allocated");
      printf("\n");
   }
}

int main()
{
   int m, n;
   printf("Enter the number of blocks: ");
   scanf("%d", &m);
   int blockSize[m];
```

```
    printf("Enter the block sizes: ");
    for (int i = 0; i < m; i++)
        scanf("%d", &blockSize[i]);

    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int processSize[n];
    for (int i = 0; i < n; i++)
        scanf("%d", &processSize[i]);

    firstFit(blockSize, m, processSize, n);

    return 0;
}
```

Output:

```
Enter the number of blocks: 4
Enter the block sizes: 100 400 200 300
Enter the number of processes: 3
Enter the process Sizes:250 50 210

Process No.     Process Size    Block no.
 1                      250                     2
 2                      50                      1
 3                      210                     4
```

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

```
#include <stdio.h>

int main(void)
{
   int ms, ps, nop, np, rempages, i, j, x, y, pa, offset;

   printf("Enter the memory size : ");
   scanf("%d", &ms);

   printf("Enter the page size : ");
   scanf("%d", &ps);

   nop = ms / ps;
   printf("The no. of pages available in memory are : %d ", nop);

   printf("Enter number of processes : ");
   scanf("%d", &np);
   int s[np], fno[np][20];
   rempages = nop;
   for (i = 1; i <= np; i++)

   {
      printf("\nEnter no. of pages required for p[%d] : ", i);
      scanf("%d", &s[i]);

      if (s[i] > rempages)
      {
         printf("\nMemory is full!");
         break;
      }
      rempages = rempages - s[i];

      printf("\nEnter pagetable for p[%d] :  ", i);
      for (j = 0; j < s[i]; j++)
         scanf("%d", &fno[i][j]);
   }

   printf("\nEnter Logical Address to find Physical Address : ");
   printf("Enter process no. and pagenumber and offset : ");

   scanf("%d %d %d", &x, &y, &offset);

   if (x > np || y >= s[i] || offset >= ps)
      printf("\nInvalid Process or Page Number or offset!");
```

```
    else
    {
        pa = fno[x][y] * ps + offset;
        printf("\nThe Physical Address is : %d", pa);
    }
}
```

Output:

```
Enter the memory size -- 1000

Enter the page size -- 100

The no. of pages available in memory are -- 10
Enter number of processes -- 3

Enter no. of pages required for p[1]-- 4

Enter pagetable for p[1] --- 8 6 9 5

Enter no. of pages required for p[2]-- 5

Enter pagetable for p[2] --- 1 4 5 7 3

Enter no. of pages required for p[3]-- 5

Memory is Full
Enter Logical Address to find Physical Address
Enter process no. and pagenumber and offset -- 2 3 60

The Physical Address is -- 760
```

Q. Write a C program to simulate the following Page Replacement algorithms

a) FIFO

b) LRU

c) Optimal

FIFO

```
#include <stdio.h>

#define FRAME_SIZE 3


int findPageInFrames(int frames[], int page, int frameCount) {
   for (int i = 0; i < frameCount; i++) {
      if (frames[i] == page) {
         return 1;
      }
   }
   return 0;
}

int main() {
   int referenceString[] = {2, 3, 4, 2, 1, 3, 7, 5, 4, 3};
   int referenceLength = sizeof(referenceString) / sizeof(referenceString[0]);
   int frames[FRAME_SIZE] = {-1};
   int frameIndex = 0;

   int pageFaults = 0;

   for (int i = 0; i < referenceLength; i++) {
      int currentPage = referenceString[i];


      if (!findPageInFrames(frames, currentPage, FRAME_SIZE)) {

         frames[frameIndex] = currentPage;
         frameIndex = (frameIndex + 1) % FRAME_SIZE;
         pageFaults++;
      }


      printf("Frames: ");
      for (int j = 0; j < FRAME_SIZE; j++) {
```

```
            if (frames[j] != -1) {
                printf("%d ", frames[j]);
            } else {
                printf("- ");
            }
        }
        printf("\n");
    }

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

    return 0;
}
```

Output:

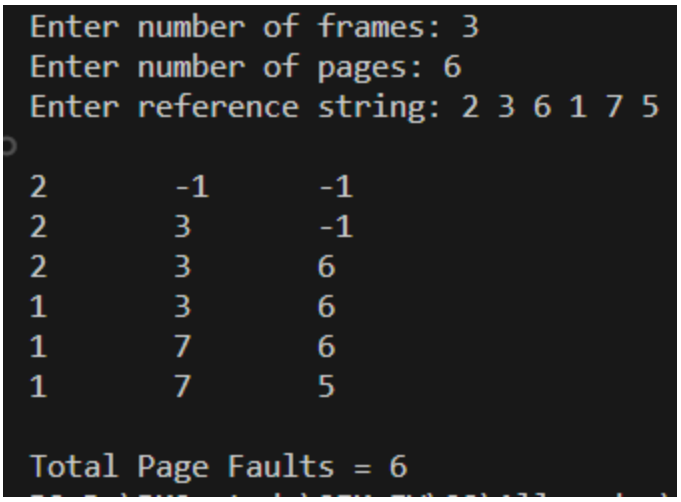
```
Frames: 2 0 0
Frames: 2 3 0
Frames: 2 3 4
Frames: 2 3 4
Frames: 1 3 4
Frames: 1 3 4
Frames: 1 7 4
Frames: 1 7 5
Frames: 4 7 5
Frames: 4 3 5
Total Page Faults: 8
```

LRU

```
#include <stdio.h>

int findLRU(int time[], int n)
{
   int i, minimum = time[0], pos = 0;

   for (i = 1; i < n; ++i)
   {
      if (time[i] < minimum)
      {
         minimum = time[i];
         pos = i;
      }
   }
   return pos;
}

int main(void)
{
   int no_of_frames, no_of_pages, counter = 0, flag1, flag2, i, j, pos, faults = 0;
   printf("Enter number of frames: ");
   scanf("%d", &no_of_frames);
   int frames[no_of_frames];

   printf("Enter number of pages: ");
   scanf("%d", &no_of_pages);
   int pages[no_of_pages];

   int time[no_of_frames];
   printf("Enter reference string: ");
   for (i = 0; i < no_of_pages; ++i)
   {
      scanf("%d", &pages[i]);
   }

   for (i = 0; i < no_of_frames; ++i)
   {
      frames[i] = -1;
   }

   for (i = 0; i < no_of_pages; ++i)
   {
      flag1 = flag2 = 0;
```

```
      for (j = 0; j < no_of_frames; ++j)
      {
         if (frames[j] == pages[i])
         {
            counter++;
            time[j] = counter;
            flag1 = flag2 = 1;
            break;
         }
      }

      if (flag1 == 0)
      {
         for (j = 0; j < no_of_frames; ++j)
         {
            if (frames[j] == -1)
            {
               counter++;
               faults++;
               frames[j] = pages[i];
               time[j] = counter;
               flag2 = 1;
               break;
            }
         }
      }

      if (flag2 == 0)
      {
         pos = findLRU(time, no_of_frames);
         counter++;
         faults++;
         frames[pos] = pages[i];
         time[pos] = counter;
      }
      printf("\n");

      for (j = 0; j < no_of_frames; ++j)
      {
         printf("%d\t", frames[j]);
      }
   }
   printf("\n\nTotal Page Faults = %d", faults);
}
```

Output:


Enter number of frames: 3
Enter number of pages: 6
Enter reference string: 2 3 6 1 7 5
2 -1 -1
2 3 -1
2 3 6
1 3 6
1 7 6
1 7 5
Total Page Faults = 6

Optimal

```
#include <stdio.h>

int main(void)
{
   int no_of_frames, no_of_pages, temp[10], flag1, flag2, flag3, i, j, k, pos, max, faults = 0;
   printf("Enter number of frames: ");
   scanf("%d", &no_of_frames);
   int frames[no_of_frames];

   printf("Enter number of pages: ");
   scanf("%d", &no_of_pages);
   int pages[no_of_pages];

   printf("Enter page reference string: ");

   for (i = 0; i < no_of_pages; ++i)
   {
      scanf("%d", &pages[i]);
   }

   for (i = 0; i < no_of_frames; ++i)
   {
      frames[i] = -1;
   }

   for (i = 0; i < no_of_pages; ++i)
   {
      flag1 = flag2 = 0;

      for (j = 0; j < no_of_frames; ++j)
      {
         if (frames[j] == pages[i])
         {
            flag1 = flag2 = 1;
            break;
         }
      }

      if (flag1 == 0)
      {
         for (j = 0; j < no_of_frames; ++j)
         {
            if (frames[j] == -1)
            {
               faults++;
               frames[j] = pages[i];
               flag2 = 1;
               break;
```

```
        }
    }
}

if (flag2 == 0)
{
    flag3 = 0;

    for (j = 0; j < no_of_frames; ++j)
    {
        temp[j] = -1;

        for (k = i + 1; k < no_of_pages; ++k)
        {
            if (frames[j] == pages[k])
            {
                temp[j] = k;
                break;
            }
        }
    }

    for (j = 0; j < no_of_frames; ++j)
    {
        if (temp[j] == -1)
        {
            pos = j;
            flag3 = 1;
            break;
        }
    }

    if (flag3 == 0)
    {
        max = temp[0];
        pos = 0;

        for (j = 1; j < no_of_frames; ++j)
        {
            if (temp[j] > max)
            {
                max = temp[j];
                pos = j;
            }
        }
    }
    frames[pos] = pages[i];
    faults++;
}

printf("\n");
```

```
      for (j = 0; j < no_of_frames; ++j)
      {
         if (frames[j] == -1)
            printf("-\t");
         else
            printf("%d\t", frames[j]);
      }
   }

   printf("\n\nTotal Page Faults = %d", faults);
}
```

Output:

```
Frames: 2 0 0
Frames: 2 3 0
Frames: 2 3 4
Frames: 2 3 4
Frames: 1 3 4
Frames: 1 3 4
Frames: 7 3 4
Frames: 5 3 4
Frames: 5 3 4
Frames: 5 3 4
Total Page Faults: 6
```

Q. Write a C program to simulate the disk scheduling algorithms

a) FCFS
b) SCAN
c) C-SCAN

FCFS

```
#include<stdio.h>
int absoluteValue(int);

void main()
{
   int queue[25],n,headposition,i,j,k,seek=0, maxrange,
   difference,temp,queue1[20],queue2[20],temp1=0,temp2=0;
   float averageSeekTime;

   printf("Enter the maximum range of Disk: ");
   scanf("%d",&maxrange);

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

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

   printf("Enter the disk positions to be read(queue): ");
   for(i=1;i<=n;i++)
   {
      scanf("%d",&temp);

      if(temp>headposition)
      {
         queue1[temp1]=temp;
         temp1++;
      }
      else
      {
         queue2[temp2]=temp;
         temp2++;
      }
   }

   for(i=0;i<temp1-1;i++)
   {
      for(j=i+1;j<temp1;j++)
```

```
    {
        if(queue1[i]>queue1[j])
        {
            temp=queue1[i];
            queue1[i]=queue1[j];
            queue1[j]=temp;
        }
    }
}

for(i=0;i<temp2-1;i++)
{
    for(j=i+1;j<temp2;j++)
    {
        if(queue2[i]<queue2[j])
        {
            temp=queue2[i];
            queue2[i]=queue2[j];
            queue2[j]=temp;
        }
    }
}

for(i=1,j=0;j<temp1;i++,j++) {
    queue[i]=queue1[j];
}

queue[i]=maxrange;

for(i=temp1+2,j=0;j<temp2;i++,j++) {
    queue[i]=queue2[j];
}

queue[i]=0;

queue[0]=headposition;


for(j=0; j<=n; j++) {
    difference = absoluteValue(queue[j+1]-queue[j]);

    seek = seek + difference;

    printf("Disk head moves from position %d to %d with Seek %d \n",
    queue[j], queue[j+1], difference);
}
```

```
    averageSeekTime = seek/(float)n;

    printf("Total Seek Time= %d\n", seek);
    printf("Average Seek Time= %f\n", averageSeekTime);
}

int absoluteValue(int x) {
    if(x>0) {
        return x;
    }
    else {
        return x*-1;
    }
}
```

Output:

```
Enter the maximum range of Disk: 200
Enter the number of queue requests: 7
Enter the initial head position: 50
Enter the disk positions to be read(queue): 82 170 43 140 24 16 190
Disk head moves from position 50 to 82 with Seek 32
Disk head moves from position 82 to 140 with Seek 58
Disk head moves from position 140 to 170 with Seek 30
Disk head moves from position 170 to 190 with Seek 20
Disk head moves from position 190 to 200 with Seek 10
Disk head moves from position 200 to 43 with Seek 157
Disk head moves from position 43 to 24 with Seek 19
Disk head moves from position 24 to 16 with Seek 8
Total Seek Time= 334
Average Seek Time= 47.714287
```

SCAN

```
#include<stdio.h>
int absoluteValue(int);

void main()
{
   int queue[25],n,headposition,i,j,k,seek=0, maxrange,
   difference,temp,queue1[20],queue2[20],temp1=0,temp2=0;
   float averageSeekTime;

   printf("Enter the maximum range of Disk: ");
   scanf("%d",&maxrange);

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

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

   printf("Enter the disk positions to be read(queue): ");
   for(i=1;i<=n;i++)
   {
      scanf("%d",&temp);

      if(temp>headposition)
      {
         queue1[temp1]=temp;
         temp1++;
      }
      else
      {
         queue2[temp2]=temp;
         temp2++;
      }
   }

   for(i=0;i<temp1-1;i++)
   {
      for(j=i+1;j<temp1;j++)
      {
         if(queue1[i]>queue1[j])
         {
            temp=queue1[i];
            queue1[i]=queue1[j];
            queue1[j]=temp;
```

```
            }
        }
    }

    for(i=0;i<temp2-1;i++)
    {
        for(j=i+1;j<temp2;j++)
        {
            if(queue2[i]<queue2[j])
            {
                temp=queue2[i];
                queue2[i]=queue2[j];
                queue2[j]=temp;
            }
        }
    }

    for(i=1,j=0;j<temp1;i++,j++) {
        queue[i]=queue1[j];
    }

    queue[i]=maxrange;

    for(i=temp1+2,j=0;j<temp2;i++,j++) {
        queue[i]=queue2[j];
    }

    queue[i]=0;

    queue[0]=headposition;


    for(j=0; j<=n; j++) {
        difference = absoluteValue(queue[j+1]-queue[j]);

        seek = seek + difference;

        printf("Disk head moves from position %d to %d with Seek %d \n",
        queue[j], queue[j+1], difference);
    }

    averageSeekTime = seek/(float)n;

    printf("Total Seek Time= %d\n", seek);
    printf("Average Seek Time= %f\n", averageSeekTime);
}
```

```
int absoluteValue(int x) {
   if(x>0) {
      return x;
   }
   else {
      return x*-1;
   }
}
```

Output:

```
Enter the maximum range of Disk: 200
Enter the number of queue requests: 7
Enter the initial head position: 50
Enter the disk positions to be read(queue): 82 170 43 140 24 16 190
Disk head moves from position 50 to 82 with Seek 32
Disk head moves from position 82 to 140 with Seek 58
Disk head moves from position 140 to 170 with Seek 30
Disk head moves from position 170 to 190 with Seek 20
Disk head moves from position 190 to 200 with Seek 10
Disk head moves from position 200 to 43 with Seek 157
Disk head moves from position 43 to 24 with Seek 19
Disk head moves from position 24 to 16 with Seek 8
Total Seek Time= 334
Average Seek Time= 47.714287
```

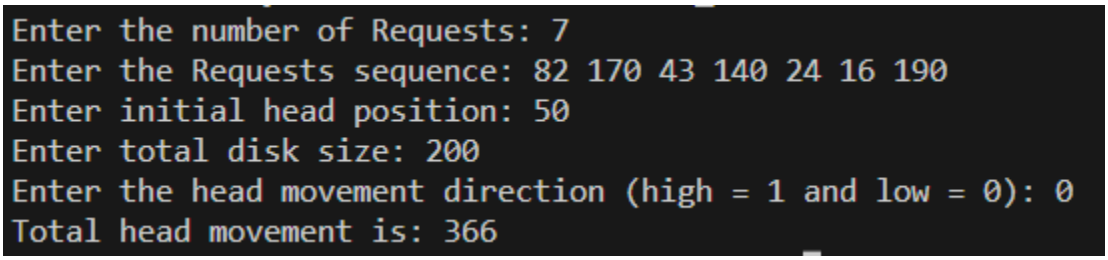
C-SCAN

```
#include<stdio.h>
int absoluteValue(int);

void main()
{
   int queue[25],n,headposition,i,j,k,seek=0, maxrange,
   difference,temp,queue1[20],queue2[20],temp1=0,temp2=0;
   float averageSeekTime;

   printf("Enter the maximum range of Disk: ");
   scanf("%d",&maxrange);

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

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

   printf("Enter the disk positions to be read(queue): ");
   for(i=1;i<=n;i++)
   {
      scanf("%d",&temp);

      if(temp>headposition)
      {
         queue1[temp1]=temp;
         temp1++;
      }
      else
      {
         queue2[temp2]=temp;
         temp2++;
      }
   }

   for(i=0;i<temp1-1;i++)
   {
      for(j=i+1;j<temp1;j++)
      {
         if(queue1[i]>queue1[j])
         {
            temp=queue1[i];
            queue1[i]=queue1[j];
            queue1[j]=temp;
```

```
            }
        }
    }

    for(i=0;i<temp2-1;i++)
    {
        for(j=i+1;j<temp2;j++)
        {
            if(queue2[i]<queue2[j])
            {
                temp=queue2[i];
                queue2[i]=queue2[j];
                queue2[j]=temp;
            }
        }
    }

    for(i=1,j=0;j<temp1;i++,j++) {
        queue[i]=queue1[j];
    }

    queue[i]=maxrange;

    for(i=temp1+2,j=0;j<temp2;i++,j++) {
        queue[i]=queue2[j];
    }

    queue[i]=0;

    queue[0]=headposition;


    for(j=0; j<=n; j++) {
        difference = absoluteValue(queue[j+1]-queue[j]);

        seek = seek + difference;

        printf("Disk head moves from position %d to %d with Seek %d \n",
        queue[j], queue[j+1], difference);
    }

    averageSeekTime = seek/(float)n;

    printf("Total Seek Time= %d\n", seek);
    printf("Average Seek Time= %f\n", averageSeekTime);
}
```

```
int absoluteValue(int x) {
   if(x>0) {
      return x;
   }
   else {
      return x*-1;
   }
}
```

Output:



Q. Write a C program to simulate the disk scheduling algorithms

a) SSTF

b) LOOK

c) C-LOOK

SSTF

```
#include<stdio.h>
int absoluteValue(int);

void main()
{
   int queue[25],n,headposition,i,j,k,seek=0, maxrange,
   difference,temp,queue1[20],queue2[20],temp1=0,temp2=0;
   float averageSeekTime;

   printf("Enter the maximum range of Disk: ");
   scanf("%d",&maxrange);

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

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

   printf("Enter the disk positions to be read(queue): ");
   for(i=1;i<=n;i++)
   {
      scanf("%d",&temp);

      if(temp>headposition)
      {
         queue1[temp1]=temp;
         temp1++;
      }
      else
      {
         queue2[temp2]=temp;
         temp2++;
      }
   }

   for(i=0;i<temp1-1;i++)
   {
      for(j=i+1;j<temp1;j++)
```

```
    {
        if(queue1[i]>queue1[j])
        {
            temp=queue1[i];
            queue1[i]=queue1[j];
            queue1[j]=temp;
        }
    }
}

for(i=0;i<temp2-1;i++)
{
    for(j=i+1;j<temp2;j++)
    {
        if(queue2[i]<queue2[j])
        {
            temp=queue2[i];
            queue2[i]=queue2[j];
            queue2[j]=temp;
        }
    }
}

for(i=1,j=0;j<temp1;i++,j++) {
    queue[i]=queue1[j];
}

queue[i]=maxrange;

for(i=temp1+2,j=0;j<temp2;i++,j++) {
    queue[i]=queue2[j];
}

queue[i]=0;

queue[0]=headposition;


for(j=0; j<=n; j++) {
    difference = absoluteValue(queue[j+1]-queue[j]);

    seek = seek + difference;

    printf("Disk head moves from position %d to %d with Seek %d \n",
    queue[j], queue[j+1], difference);
}
```

```
    averageSeekTime = seek/(float)n;

    printf("Total Seek Time= %d\n", seek);
    printf("Average Seek Time= %f\n", averageSeekTime);
}

int absoluteValue(int x) {
    if(x>0) {
        return x;
    }
    else {
        return x*-1;
    }
}
```

Output:

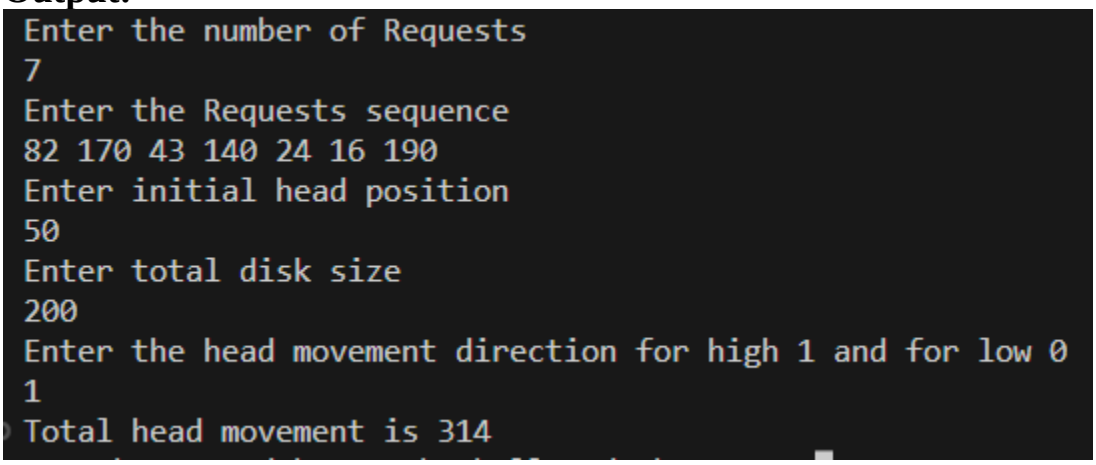
```
Enter the number of Requests: 7
Enter the Requests sequence: 82 170 43 140 24 16 190
Enter initial head position: 50
Total head movement is: 208
```

LOOK

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;
   printf("Enter the number of Requests\n");
   scanf("%d",&n);
   printf("Enter the Requests sequence\n");
   for(i=0;i<n;i++)
    scanf("%d",&RQ[i]);
   printf("Enter initial head position\n");
   scanf("%d",&initial);
   printf("Enter total disk size\n");
   scanf("%d",&size);
   printf("Enter the head movement direction for high 1 and for low 0\n");
   scanf("%d",&move);

   // logic for look disk scheduling

      /*logic for sort the request array */
   for(i=0;i<n;i++)
   {
      for(j=0;j<n-i-1;j++)
      {
         if(RQ[j]>RQ[j+1])
         {
            int temp;
            temp=RQ[j];
            RQ[j]=RQ[j+1];
            RQ[j+1]=temp;
         }

      }
   }

   int index;
   for(i=0;i<n;i++)
   {
      if(initial<RQ[i])
      {
         index=i;
         break;
      }
   }
```

```
    // if movement is towards high value
    if(move==1)
    {
       for(i=index;i<n;i++)
       {
          TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
          initial=RQ[i];
       }

       for(i=index-1;i>=0;i--)
       {
           TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
           initial=RQ[i];

       }
    }
    // if movement is towards low value
    else
    {
       for(i=index-1;i>=0;i--)
       {
          TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
          initial=RQ[i];
       }

       for(i=index;i<n;i++)
       {
           TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
           initial=RQ[i];

       }
    }

    printf("Total head movement is %d",TotalHeadMoment);
    return 0;
}
```

Output:

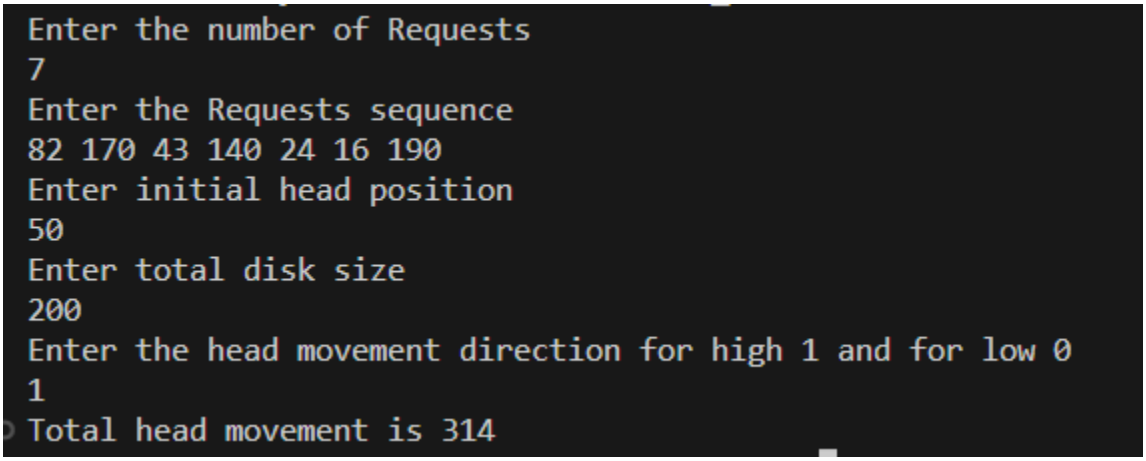

Enter the number of Requests
7
Enter the Requests sequence
82 170 43 140 24 16 190
Enter initial head position
50
Enter total disk size
200
Enter the head movement direction for high 1 and for low 0
1
Total head movement is 314

C-LOOK

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;
   printf("Enter the number of Requests\n");
   scanf("%d",&n);
   printf("Enter the Requests sequence\n");
   for(i=0;i<n;i++)
    scanf("%d",&RQ[i]);
   printf("Enter initial head position\n");
   scanf("%d",&initial);
   printf("Enter total disk size\n");
   scanf("%d",&size);
   printf("Enter the head movement direction for high 1 and for low 0\n");
   scanf("%d",&move);

   // logic for look disk scheduling

      /*logic for sort the request array */
   for(i=0;i<n;i++)
   {
      for(j=0;j<n-i-1;j++)
      {
         if(RQ[j]>RQ[j+1])
         {
            int temp;
            temp=RQ[j];
            RQ[j]=RQ[j+1];
            RQ[j+1]=temp;
         }

      }
   }

   int index;
   for(i=0;i<n;i++)
   {
      if(initial<RQ[i])
      {
         index=i;
         break;
      }
   }
```

```
    // if movement is towards high value
    if(move==1)
    {
       for(i=index;i<n;i++)
       {
          TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
          initial=RQ[i];
       }

       for(i=index-1;i>=0;i--)
       {
           TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
           initial=RQ[i];

       }
    }
    // if movement is towards low value
    else
    {
       for(i=index-1;i>=0;i--)
       {
          TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
          initial=RQ[i];
       }

       for(i=index;i<n;i++)
       {
           TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
           initial=RQ[i];

       }
    }

    printf("Total head movement is %d",TotalHeadMoment);
    return 0;
}
```

Output:


Enter the number of Requests
7
Enter the Requests sequence
82 170 43 140 24 16 190
Enter initial head position
50
Enter total disk size
200
Enter the head movement direction for high 1 and for low 0
1
Total head movement is 314