

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT

on

Operating Systems (23CS4PCOPS)

Submitted by:

Sarthak Gupta (1BM22CS352)

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 2024 - August 2024

**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 **Sarthak Gupta (1BM22CS352)**, 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 year 2022-23. The Lab report has been approved as it satisfies the academic requirements in respect of **Operating Systems - (23CS4PCOPS)** work prescribed for the said degree.

Sowmya T
Assistant Professor
Department of CSE
BMSCE, Bengaluru

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

Table Of Contents

Lab Program No.	Program Details	Page No.
1	FCFS AND SJF	1-5
2	PRIORITY AND ROUND ROBIN	6-13
3	RATE-MONOTONIC AND EARLIEST DEADLINE FIRST	14-19
4	PRODUCER-CONSUMER PROBLEM	20-21
5	DINERS-PHILOSOPHERS PROBLEM	22-24
6	BANKERS ALGORITHM(DEADLOCK AVOIDANCE)	25-27
7	DEADLOCK DETECTION	28-30
8	CONTIGUOUS MEMORY ALLOCATION(FIRST, BEST, WORST FIT)	31-34
9	PAGE REPLACEMENT(FIFO, LRU, OPTIMAL)	35-40
10	DISK SCHEDULING ALGORITHMS(FCFS, SCAN, C-SCAN)	41-46

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.

LAB - 1

Question 1:

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

CODE:

```
#include <stdio.h>
int n, i, j, pos, temp, choice, total = 0;
int Burst_time[20], Arrival_time[20], Waiting_time[20], Turn_around_time[20], process[20];
float avg_Turn_around_time = 0, avg_Waiting_time = 0;

void FCFS() {
    int total_waiting_time = 0, total_turnaround_time = 0;
    int current_time = 0;

    for (i = 0; i < n - 1; i++) {
        for (j = i + 1; j < n; j++) {
            if (Arrival_time[i] > Arrival_time[j]) {
                temp = Arrival_time[i];
                Arrival_time[i] = Arrival_time[j];
                Arrival_time[j] = temp;

                temp = Burst_time[i];
                Burst_time[i] = Burst_time[j];
                Burst_time[j] = temp;

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

    Waiting_time[0] = 0;
    current_time = Arrival_time[0] + Burst_time[0];

    for (i = 1; i < n; i++) {
        if (current_time < Arrival_time[i]) {
            current_time = Arrival_time[i];
        }
        Waiting_time[i] = current_time - Arrival_time[i];
        current_time += Burst_time[i];
        total_waiting_time += Waiting_time[i];
    }

    printf("\nProcess\t\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time");
```

```

for (i = 0; i < n; i++) {
    Turn_around_time[i] = Burst_time[i] + Waiting_time[i];
    total_turnaround_time += Turn_around_time[i];
    printf("\nP[%d]\t\t%d\t\t%d\t\t%d\t\t%d", process[i], Arrival_time[i], Burst_time[i], Waiting_time[i],
Turn_around_time[i]);
}

avg_Waiting_time = (float)total_waiting_time / n;
avg_Turn_around_time = (float)total_turnaround_time / n;
printf("\nAverage Waiting Time: %.2f", avg_Waiting_time);
printf("\nAverage Turnaround Time: %.2f\n", avg_Turn_around_time);
}

void SJF() {
    int total_waiting_time = 0, total_turnaround_time = 0;
    int completed = 0, current_time = 0, min_index;
    int is_completed[20] = {0};

    while (completed != n) {
        int min_burst_time = 9999;
        min_index = -1;

        for (i = 0; i < n; i++) {
            if (Arrival_time[i] <= current_time && is_completed[i] == 0) {
                if (Burst_time[i] < min_burst_time) {
                    min_burst_time = Burst_time[i];
                    min_index = i;
                }
                if (Burst_time[i] == min_burst_time) {
                    if (Arrival_time[i] < Arrival_time[min_index]) {
                        min_burst_time = Burst_time[i];
                        min_index = i;
                    }
                }
            }
        }

        if (min_index != -1) {
            Waiting_time[min_index] = current_time - Arrival_time[min_index];
            current_time += Burst_time[min_index];
            Turn_around_time[min_index] = current_time - Arrival_time[min_index];
            total_waiting_time += Waiting_time[min_index];
            total_turnaround_time += Turn_around_time[min_index];
            is_completed[min_index] = 1;
            completed++;
        } else {
            current_time++;
        }
    }

    printf("\nProcess\t\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time");
    for (i = 0; i < n; i++) {

```

```

        printf("\nP[%d]\t\t%d\t\t%d\t\t%d\t\t%d", process[i], Arrival_time[i], Burst_time[i], Waiting_time[i],
Turn_around_time[i]);
    }

    avg_Waiting_time = (float)total_waiting_time / n;
    avg_Turn_around_time = (float)total_turnaround_time / n;
    printf("\n\nAverage Waiting Time = %.2f", avg_Waiting_time);
    printf("\n\nAverage Turnaround Time = %.2f\n", avg_Turn_around_time);
}

int main() {
    printf("Enter the total number of processes: ");
    scanf("%d", &n);
    printf("\nEnter Arrival Time and Burst Time:\n");
    for (i = 0; i < n; i++) {
        printf("P[%d] Arrival Time: ", i + 1);
        scanf("%d", &Arrival_time[i]);
        printf("P[%d] Burst Time: ", i + 1);
        scanf("%d", &Burst_time[i]);
        process[i] = i + 1;
    }
    while (1) {
        printf("\n-----MAIN MENU-----\n");
        printf("1. FCFS Scheduling\n2. SJF Scheduling\n");
        printf("\nEnter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1: FCFS();
                    break;
            case 2: SJF();
                    break;
            default: printf("Invalid Input!!!\n");
        }
    }
    return 0;
}

```

OUTPUTS:

a.

Enter the total number of processes: 5

Enter Arrival Time and Burst Time:

P[1] Arrival Time: 0

P[1] Burst Time: 10

P[2] Arrival Time: 0

P[2] Burst Time: 1

P[3] Arrival Time: 3

P[3] Burst Time: 2

P[4] Arrival Time: 5

P[4] Burst Time: 1

P[5] Arrival Time: 10

P[5] Burst Time: 5

-----MAIN MENU-----

1. FCFS Scheduling

2. SJF Scheduling

Enter your choice: 1

Process	Arrival Time	Burst Time	Waiting Time	Turnaround Time
P[1]	0	10	0	10
P[2]	0	1	10	11
P[3]	3	2	8	10
P[4]	5	1	8	9
P[5]	10	5	4	9

Average Waiting Time: 6.00

Average Turnaround Time: 9.80

b.

Enter the total number of processes: 4

Enter Arrival Time and Burst Time:

P[1] Arrival Time: 0

P[1] Burst Time: 3

P[2] Arrival Time: 1

P[2] Burst Time: 6

P[3] Arrival Time: 4

P[3] Burst Time: 4

P[4] Arrival Time: 6

P[4] Burst Time: 2

-----MAIN MENU-----

1. FCFS Scheduling

2. SJF Scheduling

Enter your choice: 2

Process	Arrival Time	Burst Time	Waiting Time	Turnaround Time
P[1]	0	3	0	3
P[2]	1	6	2	8
P[3]	4	4	7	11
P[4]	6	2	3	5

Average Waiting Time = 3.00

Average Turnaround Time = 6.75

LAB - 2

Question:

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

- (a) Priority (pre-emptive & Non-pre-emptive)
- (b) Round Robin (Experiment with different quantum sizes for RR algorithm)

CODE:

(a) Priority (Non-pre-emptive)

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

struct process {
    int process_id;
    int burst_time;
    int priority;
    int waiting_time;
    int turnaround_time;
};

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].process_id);

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

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

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

```

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

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

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

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

    find_waiting_time(proc, n, wt);
    find_turnaround_time(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%d\t%d\t%d\t%d", proc[i].process_id, proc[i].burst_time, proc[i].priority, wt[i],
            tat[i]);
    }
    printf("\n\nAverage Waiting Time = %f", (float)total_wt/n);
    printf("\n\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: 5

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

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

Enter the process ID: 3
Enter the burst time: 1
Enter the priority: 4

Enter the process ID: 4
Enter the burst time: 5
Enter the priority: 5

Enter the process ID: 5
Enter the burst time: 2
Enter the priority: 5

Process ID      Burst Time      Priority      Waiting Time      Turnaround Time
1                4                2              0                  4
2                3                3              4                  7
3                1                4              7                  8
4                5                5              8                 13
5                2                5             13                 15

Average Waiting Time = 6.400000
Average Turnaround Time = 9.400000

```

Priority (Pre-emptive):

CODE:

```

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

struct process {
    int process_id;
    int burst_time;
    int priority;
    int arrival_time;
    int remaining_time;
    int waiting_time;
    int turnaround_time;
    int is_completed;
};

```

```
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].process_id);

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

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

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

        proc[i].remaining_time = proc[i].burst_time;
        proc[i].is_completed = 0;
    }

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

```
void find_waiting_time(struct process proc[], int n) {
    int time = 0, completed = 0, min_priority, shortest = 0;
    while (completed != n) {
        min_priority = 10000;
        for (int i = 0; i < n; i++) {
            if ((proc[i].arrival_time <= time) && (!proc[i].is_completed) && (proc[i].priority < min_priority)) {
                min_priority = proc[i].priority;
                shortest = i;
            }
        }
    }
}
```

```

    }
    proc[shortest].remaining_time--;
    time++;
    if (proc[shortest].remaining_time == 0) {
        proc[shortest].waiting_time = time - proc[shortest].arrival_time - proc[shortest].burst_time;
        proc[shortest].turnaround_time = time - proc[shortest].arrival_time;
        proc[shortest].is_completed = 1;
        completed++;
    }
}
}

void find_turnaround_time(struct process proc[], int n) {
    // Turnaround time is calculated during the find_waiting_time function
}

void find_average_time(struct process proc[], int n) {
    int total_wt = 0, total_tat = 0;
    find_waiting_time(proc, n);
    find_turnaround_time(proc, n);

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

    for (int i = 0; i < n; i++) {
        total_wt += proc[i].waiting_time;
        total_tat += proc[i].turnaround_time;
        printf("\n%d\t%d\t%d\t%d\t%d\t%d", proc[i].process_id, proc[i].burst_time,
        proc[i].arrival_time, proc[i].priority, proc[i].waiting_time, proc[i].turnaround_time);
    }
    printf("\n\nAverage Waiting Time = %f", (float)total_wt / n);
    printf("\n\nAverage Turnaround Time = %f\n", (float)total_tat / n);
}

void priority_scheduling(struct process proc[], int n) {
    find_average_time(proc, n);
}

```

OUTPUT:

```

Enter the number of processes: 5
Enter the process ID: 5
Enter the burst time: 2
Enter the arrival time: 4
Enter the priority: 5
Enter the process ID: 1
Enter the burst time: 4
Enter the arrival time: 0
Enter the priority: 2
Enter the process ID: 2
Enter the burst time: 3
Enter the arrival time: 1
Enter the priority: 3
Enter the process ID: 3
Enter the burst time: 1
Enter the arrival time: 2
Enter the priority: 4
Enter the process ID: 4
Enter the burst time: 5
Enter the arrival time: 3
Enter the priority: 5

```

Process ID	Burst Time	Arrival Time	Priority	Waiting Time	Turnaround Time
5	2	4	5	4	6
1	4	0	2	0	4
2	3	1	3	3	6
3	1	2	4	5	6
4	5	3	5	7	12

```

Average Waiting Time = 3.800000
Average Turnaround Time = 6.800000

```

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

```

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

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

void findWaitingTime(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 += rem_bt[i];
            wt[i] = t - bt[i];
            rem_bt[i] = 0;
        }
    }
}
if (done == true)
    break;
}
}

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

    findWaitingTime(processes, n, bt, wt, quantum);
    findTurnaroundTime(processes, n, bt, wt, tat);

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

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

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

int main() {
    int n, quantum;

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

    int processes[n], burst_time[n];

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

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

```

```

    findAvgTime(processes, n, burst_time, quantum);
    return 0;
}

```

OUTPUT:

```

Enter the Number of Processes: 5

Enter the quantum time: 2

Enter the process ID: 1
Enter the Burst Time: 5

Enter the process ID: 2
Enter the Burst Time: 3

Enter the process ID: 3
Enter the Burst Time: 1

Enter the process ID: 4
Enter the Burst Time: 2

Enter the process ID: 5
Enter the Burst Time: 3

Process ID      Burst Time      Waiting Time      Turnaround Time
1                5                9                14
2                3                9                12
3                1                4                5
4                2                5                7
5                3                10               13

Average waiting time = 7.400000
Average turnaround time = 10.200000

```


LAB - 3

Question 1:

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

- (a) Rate- Monotonic**
- (b) Earliest-deadline First**

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);
}

int main()
{
    int option;
    int observation_time;

    while (1)
    {
        printf("\n1. Rate Monotonic\n2. Earliest Deadline first\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: exit (0);
            default: printf("\nInvalid Statement");
        }
    }
    return 0;
}

```

Output:

(a) Rate Monotonic:

1. Rate Monotonic
2. Earliest Deadline first

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

Given problem is not schedulable under the said scheduling algorithm.

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

Enter your choice: 2

Enter total number of processes (maximum 10): 2

Process 1:

==> Execution time: 4

==> Deadline: 6

Process 2:

==> Execution time: 3

==> Deadline: 2

Scheduling:

Time:	00	01	02	03	04	05	
P[1]:							
P[2]:	#### ####	#### ####	#### ####	#### ####	#### ####	#### ####	

LAB - 4

Question 1:

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

Code:

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

int mutex=1,full=0,empty=3,x=0;

int main()
{
    int n;
    void producer();
    void consumer();
    int wait(int);
    int signal(int);
    printf("\n1.Producer\n2.Consumer\n3.Exit");
    while(1)
    {
        printf("\nEnter your choice: ");
        scanf("%d",&n);
        switch(n)
        {
            case 1: if((mutex==1)&&(empty!=0))
                    producer();
                    else
                    printf("Buffer is full!!");
                    break;
            case 2: if((mutex==1)&&(full!=0))
                    consumer();
                    else
                    printf("Buffer is empty!!");
                    break;
            case 3: exit(0);
                    break;
        }
    }
    return 0;
}

int wait(int s)
{
    return (--s);
}

int signal(int s)
{
    return(++s);
}

void producer()
```

```

{
    mutex=wait(mutex);
    full=signal(full);
    empty=wait(empty);
    x++;
    printf("\nProducer produces the item %d",x);
    mutex=signal(mutex);
}

```

```

void consumer()
{
    mutex=wait(mutex);
    full=wait(full);
    empty=signal(empty);

    printf("\nConsumer consumes item %d",x);
    x--;
    mutex=signal(mutex);
}

```

OUTPUT:

```

1.Producer
2.Consumer
3.Exit
Enter your choice: 1

Producer produces the item 1
Enter your choice: 1

Producer produces the item 2
Enter your choice: 2

Consumer consumes item 2
Enter your choice: 2

Consumer consumes item 1
Enter your choice: 1

Producer produces the item 1
Enter your choice: 2

Consumer consumes item 1
Enter your choice: 2
Buffer is empty!!
Enter your choice: 3

```


Question 2:

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

CODE:

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

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

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

sem_t mutex;
sem_t S[N];

void test(int i)
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
    {
        state[i] = EATING;

        sleep(2);

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

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

        sem_post(&S[i]);
    }
}

void take_fork(int i)
{
    sem_wait(&mutex);
    state[i] = HUNGRY;
    printf("Philosopher %d is Hungry\n", i + 1);
    test(i);

    sem_post(&mutex);
    sem_wait(&S[i]);
    sleep(1);
}

void put_fork(int i)
{
    sem_wait(&mutex);
    state[i] = THINKING;
```

```

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

    printf("Philosopher %d is thinking\n", i + 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];

    sem_init(&mutex, 0, 1);

    for (i = 0; i < N; i++)
        sem_init(&S[i], 0, 0);

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

```
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 1 is Hungry
Philosopher 2 is Hungry
Philosopher 3 is Hungry
Philosopher 4 is Hungry
Philosopher 5 is Hungry
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 5 putting fork 4 and 5 down
```

LAB 5

Question 1:

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

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;
}

```

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

```

Question 2:

Write a C program to simulate deadlock detection.

CODE:

```
#include<stdio.h>
static int mark[20];
int i,j,np,nr;

int main()
{
int alloc[10][10],request[10][10],avail[10],r[10],w[10];

printf("\nEnter the no of process: ");
scanf("%d",&np);
printf("\nEnter the no of resources: ");
scanf("%d",&nr);
for(i=0;i<nr;i++)
{
printf("\nTotal Amount of the Resource R%d: ",i+1);
scanf("%d",&r[i]);
}

printf("\nEnter the request matrix:");

for(i=0;i<np;i++)
for(j=0;j<nr;j++)
scanf("%d",&request[i][j]);

printf("\nEnter the allocation matrix:");
for(i=0;i<np;i++)
for(j=0;j<nr;j++)
scanf("%d",&alloc[i][j]);

for(j=0;j<nr;j++)
{
avail[j]=r[j];
for(i=0;i<np;i++)
{
avail[j]-=alloc[i][j];
}
}

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;
}

for(j=0;j<nr;j++)
    w[j]=avail[j];

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];
    }
}

int deadlock=0;
for(i=0;i<np;i++)
    if(mark[i]!=1)
        deadlock=1;

if(deadlock)
    printf("\n Deadlock detected");
else
    printf("\n No Deadlock possible");
}

```


OUTPUT:

```
Enter the no of process: 5
Enter the no of resources: 3
Total Amount of the Resource R1: 0
Total Amount of the Resource R2: 0
Total Amount of the Resource R3: 0
Enter the request 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
2 1 1
0 0 2
Deadlock detected
```

LAB 6

Question 1:

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

- (a) **Worst-fit**
- (b) **Best-fit**
- (c) **First-fit**

CODE:

```
#include <stdio.h>

#define max 25

void firstFit(int b[], int nb, int f[], int nf);
void worstFit(int b[], int nb, int f[], int nf);
void bestFit(int b[], int nb, int f[], int nf);

int main()
{
    int b[max], f[max], nb, nf;

    printf("Memory Management Schemes\n");

    printf("\nEnter the number of blocks:");
    scanf("%d", &nb);

    printf("Enter the number of files:");
    scanf("%d", &nf);

    printf("\nEnter the size of the blocks:\n");
    for (int i = 1; i <= nb; i++)
    {
        printf("Block %d:", i);
        scanf("%d", &b[i]);
    }

    printf("\nEnter the size of the files:\n");
    for (int i = 1; i <= nf; i++)
    {
        printf("File %d:", i);
        scanf("%d", &f[i]);
    }

    printf("\nMemory Management Scheme - First Fit");
    firstFit(b, nb, f, nf);

    printf("\n\nMemory Management Scheme - Worst Fit");
    worstFit(b, nb, f, nf);

    printf("\n\nMemory Management Scheme - Best Fit");
    bestFit(b, nb, f, nf);
}
```

```

    return 0;
}

void firstFit(int b[], int nb, int f[], int nf)
{
    int bf[max] = {0};
    int ff[max] = {0};
    int frag[max], i, j;

    for (i = 1; i <= nf; i++)
    {
        for (j = 1; j <= nb; j++)
        {
            if (bf[j] != 1 && b[j] >= f[i])
            {
                ff[i] = j;
                bf[j] = 1;
                frag[i] = b[j] - f[i];
                break;
            }
        }
    }

    printf("\nFile_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment");
    for (i = 1; i <= nf; i++)
    {
        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]);
    }
}

void worstFit(int b[], int nb, int f[], int nf)
{
    int bf[max] = {0};
    int ff[max] = {0};
    int frag[max], i, j, temp, highest = 0;

    for (i = 1; i <= nf; i++)
    {
        for (j = 1; j <= nb; j++)
        {
            if (bf[j] != 1)
            {
                temp = b[j] - f[i];
                if (temp >= 0 && highest < temp)
                {
                    ff[i] = j;
                    highest = temp;
                }
            }
        }
        frag[i] = highest;
    }
}

```

```

        bf[ff[i]] = 1;
        highest = 0;
    }

    printf("\nFile_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment");
    for (i = 1; i <= nf; i++)
    {
        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]);
    }
}

void bestFit(int b[], int nb, int f[], int nf)
{
    int bf[max] = {0};
    int ff[max] = {0};
    int frag[max], i, j, temp, lowest = 10000;

    for (i = 1; i <= nf; i++)
    {
        for (j = 1; j <= nb; j++)
        {
            if (bf[j] != 1)
            {
                temp = b[j] - f[i];
                if (temp >= 0 && lowest > temp)
                {
                    ff[i] = j;
                    lowest = temp;
                }
            }
        }
        frag[i] = lowest;
        bf[ff[i]] = 1;
        lowest = 10000;
    }

    printf("\nFile_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment");
    for (i = 1; i <= nf && ff[i] != 0; i++)
    {
        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]);
    }
}

```

OUTPUT:

Memory Management Schemes

Enter the number of blocks:5

Enter the number of files:5

Enter the size of the blocks:

Block 1:100

Block 2:500

Block 3:200

Block 4:300

Block 5:600

Enter the size of the files:

File 1:212

File 2:415

File 3:63

File 4:200

File 5:255

Memory Management Scheme - First Fit

File_no:	File_size:	Block_no:	Block_size:	Fragment
1	212	2	500	288
2	415	5	600	185
3	63	1	100	37
4	200	3	200	0
5	255	4	300	45

Memory Management Scheme - Worst Fit

File_no:	File_size:	Block_no:	Block_size:	Fragment
1	212	5	600	388
2	415	2	500	85
3	63	4	300	237
4	200	0	0	0
5	255	0	0	0

Memory Management Scheme - Best Fit

File_no:	File_size:	Block_no:	Block_size:	Fragment
1	212	4	300	88
2	415	2	500	85
3	63	1	100	37
4	200	3	200	0
5	255	5	600	345

Question 2:

Write a C program to simulate page replacement algorithms:

- (a) FIFO
- (b) LRU
- (c) Optimal

CODE:

```
#include<stdio.h>
int n, f, i, j, k;
int in[100];
int p[50];
int hit=0;
int pgfaultcnt=0;

void getData()
{
    printf("\nEnter length of page reference sequence:");
    scanf("%d",&n);
    printf("\nEnter the page reference sequence:");
    for(i=0; i<n; i++)
        scanf("%d",&in[i]);
    printf("\nEnter no of frames:");
    scanf("%d",&f);
}

void initialize()
{
    pgfaultcnt=0;
    for(i=0; i<f; i++)
        p[i]=9999;
}

int isHit(int data)
{
    hit=0;
    for(j=0; j<f; j++)
    {
        if(p[j]==data)
        {
            hit=1;
            break;
        }
    }
    return hit;
}

int getHitIndex(int data)
{
    int hitind;
    for(k=0; k<f; k++)
```

```

    {
        if(p[k]==data)
        {
            hitind=k;
            break;
        }
    }
    return hitind;
}

void dispPages()
{
    for (k=0; k<f; k++)
    {
        if(p[k]!=9999)
            printf(" %d",p[k]);
    }
}

void dispPgFaultCnt()
{
    printf("\nTotal no of page faults:%d",pgfaultcnt);
}

void fifo()
{
    getdata();
    initialize();
    for(i=0; i<n; i++)
    {
        printf("\nFor %d :",in[i]);
//not a hit
        if(isHit(in[i])==0)
        {

            for(k=0; k<f-1; k++)
                p[k]=p[k+1];

            p[k]=in[i];
            pgfaultcnt++;
            dispPages();
        }
        else
            printf("No page fault");
    }
    dispPgFaultCnt();
}

void optimal()
{
    initialize();

```

```

int near[50];
for(i=0; i<n; i++)
{

    printf("\nFor %d :",in[i]);

    if(isHit(in[i])==0)
    {

        for(j=0; j<f; j++)
        {
            int pg=p[j];
            int found=0;
            for(k=i; k<n; k++)
            {
                if(pg==in[k])
                {
                    near[j]=k;
                    found=1;
                    break;
                }
                else
                    found=0;
            }
            if(!found)
                near[j]=9999;
        }
        int max=-9999;
        int repindex;
        for(j=0; j<nf; j++)
        {
            if(near[j]>max)
            {
                max=near[j];
                repindex=j;
            }
        }
        p[repindex]=in[i];
        pgfaultcnt++;

        dispPages();
    }
    else
        printf("No page fault");
}
dispPgFaultCnt();
}

void lru()
{
    initialize();

```



```

int least[50];
for(i=0; i<n; i++)
{

    printf("\nFor %d :",in[i]);

    if(isHit(in[i])==0)
    {

        for(j=0; j<nf; j++)
        {
            int pg=p[j];
            int found=0;
            for(k=i-1; k>=0; k--)
            {
                if(pg==in[k])
                {
                    least[j]=k;
                    found=1;
                    break;
                }
                else
                    found=0;
            }
            if(!found)
                least[j]=-9999;
        }
        int min=9999;
        int repindex;
        for(j=0; j<nf; j++)
        {
            if(least[j]<min)
            {
                min=least[j];
                repindex=j;
            }
        }
        p[repindex]=in[i];
        pgfaultcnt++;

        dispPages();
    }
    else
        printf("No page fault!");
}
dispPgFaultCnt();
}

int main()
{
    int choice;
    while(1)

```

```

{
    printf("\nPage Replacement Algorithms\n1.Enter data\n2.FIFO\n3.Optimal\n4.LRU\n5.Exit\nEnter your
choice:");
    scanf("%d",&choice);
    switch(choice)
    {
        case 1: getData();
            break;
        case 2: fifo();
            break;
        case 3: optimal();
            break;
        case 4: lru();
            break;
        default: return 0;
            break;
    }
}
}

```

OUTPUT:

```

Page Replacement Algorithms
1.Enter data
2.FIFO
3.Optimal
4.LRU
5.Exit
Enter your choice:2

Enter length of page reference sequence:12

Enter the page reference sequence:1 2 3 4 1 2 5 1 2 3 4 5

Enter no of frames:3

For 1 : 1
For 2 : 1 2
For 3 : 1 2 3
For 4 : 2 3 4
For 1 : 3 4 1
For 2 : 4 1 2
For 5 : 1 2 5
For 1 :No page fault
For 2 :No page fault
For 3 : 2 5 3
For 4 : 5 3 4
For 5 :No page fault
Total no of page faults:9

```

```

Enter your choice:3

For 1 : 1
For 2 : 1 2
For 3 : 1 2 3
For 4 : 1 2 4
For 1 :No page fault
For 2 :No page fault
For 5 : 1 2 5
For 1 :No page fault
For 2 :No page fault
For 3 : 3 2 5
For 4 : 4 2 5
For 5 :No page fault
Total no of page faults:7
Page Replacement Algorithms
1.Enter data
2.FIFO
3.Optimal
4.LRU
5.Exit
Enter your choice:4

For 1 : 1
For 2 : 1 2
For 3 : 1 2 3
For 4 : 4 2 3
For 1 : 4 1 3
For 2 : 4 1 2
For 5 : 5 1 2
For 1 :No page fault!
For 2 :No page fault!
For 3 : 3 1 2
For 4 : 3 4 2
For 5 : 3 4 5
Total no of page faults:10

```

LAB 7

Question 1:

Write a C program to simulate the disk scheduling algorithms.

(a)FCFS

(b)SCAN

(c)C-SCAN

(a) FCFS:

CODE:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int RQ[100],i,n,TotalHeadMoment=0,initial;
    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);

    for(i=0;i<n;i++)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];
    }

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

OUTPUT:

```
Enter the number of Requests
8
Enter the Requests sequence
98 183 37 122 14 124 65 67
Enter initial head position
53
Total head moment is 640
```

(b) SCAN:

CODE:

```
#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);

    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(move==1)
{
    for(i=index;i<n;i++)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];
    }

    TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);
    initial = size-1;
    for(i=index-1;i>=0;i--)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];
    }
}
else
{
    for(i=index-1;i>=0;i--)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];
    }
    TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);
    initial =0;
    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
8
Enter the Requests sequence
98 183 37 122 14 124 65 67
Enter initial head position
53
Enter total disk size
199
Enter the head movement direction for high 1 and for low 0
0
Total head movement is 236
```

(d) C-SCAN:

CODE:

```
#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);

    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(move==1)
{
    for(i=index;i<n;i++)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];
    }
    TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);
    TotalHeadMoment=TotalHeadMoment+abs(size-1-0);
    initial=0;
    for( i=0;i<index;i++)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];
    }
}
else
{
    for(i=index-1;i>=0;i--)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];
    }
}

```



```

TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);

TotalHeadMoment=TotalHeadMoment+abs(size-1-0);
initial =size-1;
for(i=n-1;i>=index;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
8
Enter the Requests sequence
98 183 37 122 14 124 65 67
Enter initial head position
53
Enter total disk size
199
Enter the head movement direction for high 1 and for low 0
0
Total head movement is 384

```