

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT on

OPERATING SYSTEMS

Submitted by

G SANJANA HEBBAR(1BM21CS062)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING



B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)

BENGALURU-560019

June-2023 to September-2023

B. M. S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering



CERTIFICATE

This is to certify that the Lab work entitled “OPERATING SYSTEMS” carried out by **G Sanjana Hebbar(1BM21CS062)**, 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 September-2023. The Lab report has been approved as it satisfies the academic requirements in respect of a **OPERATING SYSTEMS (22CS4PCOPS)** work prescribed for the said degree.

Name of the Lab-In charge: Dr.Nandhini Vineeth
Designation: Assistant Professor
Department of CSE
BMSCE, Bengaluru

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

Index Sheet

Lab Program No.	Program Details	Page No.
1	Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time. <ul style="list-style-type: none">• FCFS• SJF (pre-emptive & Non-pre-emptive)	6
2	Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time. <ul style="list-style-type: none">• Priority (pre-emptive & Non-pre-emptive)• Round Robin (Experiment with different quantum sizes for RR algorithm)	13
3	Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories – system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling for the processes in each queue.	21
4	Write a C program to simulate Real-Time CPU Scheduling algorithms: <ul style="list-style-type: none">a) Rate- Monotonicb) Earliest-deadline Firstc) Proportional scheduling	29
5	Write a C program to simulate producer-consumer problem using semaphores.	36
6	Write a C program to simulate the concept of Dining-Philosophers problem.	39
7	Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.	45

8	Write a C program to simulate deadlock detection	50
9	Write a C program to simulate the following contiguous memory allocation techniques a) Worst-fit b) Best-fit c) First-fit	54
10	Write a C program to simulate page replacement algorithms a) FIFO b) LRU c) Optimal	60
11	Write a C program to simulate disk scheduling algorithms a) FCFS b) SCAN c) C-SCAN	69
12	Write a C program to simulate disk scheduling algorithms a) SSTF b) LOOK c) c-LOOK	69

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-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time.

- **FCFS**
- **SJF (pre-emptive & Non-pre-emptive)**

CODE:

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

```
// Structure to represent a process
```

```
struct Process {
```

```
    int pid;
```

```
    int arrivalTime;
```

```
    int burstTime;
```

```
    int remainingTime;
```

```
};
```

```
// Function to calculate FCFS scheduling
```

```
void fcfs(struct Process processes[], int n) {
```

```
    int currentTime = 0;
```

```
    int totalWaitingTime = 0;
```

```
    int totalTurnaroundTime = 0;
```

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

```
        if (currentTime < processes[i].arrivalTime) {
```

```
            currentTime = processes[i].arrivalTime;
```

```
        }
```

```
        totalWaitingTime += currentTime - processes[i].arrivalTime;
```

```

        totalTurnaroundTime += currentTime + processes[i].burstTime - processes[i].arrivalTime;
        currentTime += processes[i].burstTime;
    }

    printf("FCFS Scheduling:\n");
    printf("Average Waiting Time: %.2f\n", (float)totalWaitingTime / n);
    printf("Average Turnaround Time: %.2f\n", (float)totalTurnaroundTime / n);
}

// Function to calculate SJF (Non-preemptive) scheduling
void sjfNonPreemptive(struct Process processes[], int n) {
    // Sorting processes based on burst time (shortest first)
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].burstTime > processes[j + 1].burstTime) {
                struct Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
}

int currentTime = 0;
int totalWaitingTime = 0;
int totalTurnaroundTime = 0;

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

```

```

    if (currentTime < processes[i].arrivalTime) {
        currentTime = processes[i].arrivalTime;
    }

    totalWaitingTime += currentTime - processes[i].arrivalTime;
    totalTurnaroundTime += currentTime + processes[i].burstTime - processes[i].arrivalTime;
    currentTime += processes[i].burstTime;
}

printf("SJF (Non-preemptive) Scheduling:\n");
printf("Average Waiting Time: %.2f\n", (float)totalWaitingTime / n);
printf("Average Turnaround Time: %.2f\n", (float)totalTurnaroundTime / n);
}

// Function to calculate SJF (Preemptive) scheduling
void sjfPreemptive(struct Process processes[], int n) {
    int currentTime = 0;
    int totalWaitingTime = 0;
    int totalTurnaroundTime = 0;
    int completed = 0;

    while (completed < n) {
        int shortestIndex = -1;
        int shortestBurst = -1;

        for (int i = 0; i < n; i++) {
            if (processes[i].remainingTime > 0 && processes[i].arrivalTime <= currentTime &&

```



```

        (shortestIndex == -1 || processes[i].remainingTime < shortestBurst)) {
            shortestIndex = i;
            shortestBurst = processes[i].remainingTime;
        }
    }

    if (shortestIndex == -1) {
        currentTime++;
        continue;
    }

    processes[shortestIndex].remainingTime--;
    currentTime++;

    if (processes[shortestIndex].remainingTime == 0) {
        completed++;

        totalWaitingTime += currentTime - processes[shortestIndex].arrivalTime -
processes[shortestIndex].burstTime;

        totalTurnaroundTime += currentTime - processes[shortestIndex].arrivalTime;
    }
}

printf("SJF (Preemptive) Scheduling:\n");
printf("Average Waiting Time: %.2f\n", (float)totalWaitingTime / n);
printf("Average Turnaround Time: %.2f\n", (float)totalTurnaroundTime / n);
}

int main() {

```

```

int n;

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

struct Process processes[n];

for (int i = 0; i < n; i++) {
    processes[i].pid = i + 1;
    printf("Enter arrival time for process %d: ", i + 1);
    scanf("%d", &processes[i].arrivalTime);
    printf("Enter burst time for process %d: ", i + 1);
    scanf("%d", &processes[i].burstTime);
    processes[i].remainingTime = processes[i].burstTime;
}

int choice;
printf("Select scheduling algorithm:\n");
printf("1. FCFS\n");
printf("2. SJF (Non-preemptive)\n");
printf("3. SJF (Preemptive)\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
    case 1:
        fcfs(processes, n);

```

```

        break;
    case 2:
        sjfNonPreemptive(processes, n);
        break;
    case 3:
        sjfPreemptive(processes, n);
        break;
    default:
        printf("Invalid choice.\n");
}

return 0;
}

```

OUTPUT

```

Enter the number of processes: 5
Enter arrival time for process 1: 0
Enter burst time for process 1: 10
Enter arrival time for process 2: 3
Enter burst time for process 2: 5
Enter arrival time for process 3: 5
Enter burst time for process 3: 2
Enter arrival time for process 4: 6
Enter burst time for process 4: 6
Enter arrival time for process 5: 8
Enter burst time for process 5: 4
Select scheduling algorithm:
1. FCFS
2. SJF (Non-preemptive)
3. SJF (Preemptive)
Enter your choice: 1
FCFS Scheduling:
Average Waiting Time: 8.60
Average Turnaround Time: 14.00

Process returned 0 (0x0)   execution time : 18.912 s
Press any key to continue.

```

```
Enter the number of processes: 4
Enter arrival time for process 1: 0
Enter burst time for process 1: 8
Enter arrival time for process 2: 1
Enter burst time for process 2: 4
Enter arrival time for process 3: 2
Enter burst time for process 3: 9
Enter arrival time for process 4: 3
Enter burst time for process 4: 5
Select scheduling algorithm:
1. FCFS
2. SJF (Non-preemptive)
3. SJF (Preemptive)
Enter your choice: 2
SJF (Non-preemptive) Scheduling:
Average Waiting Time: 7.00
Average Turnaround Time: 13.50

Process returned 0 (0x0)   execution time : 26.209 s
Press any key to continue.
```

```
Enter the number of processes: 4
Enter arrival time for process 1: 0
Enter burst time for process 1: 8
Enter arrival time for process 2: 1
Enter burst time for process 2: 4
Enter arrival time for process 3: 2
Enter burst time for process 3: 9
Enter arrival time for process 4: 3
Enter burst time for process 4: 5
Select scheduling algorithm:
1. FCFS
2. SJF (Non-preemptive)
3. SJF (Preemptive)
Enter your choice: 3
SJF (Preemptive) Scheduling:
Average Waiting Time: 6.50
Average Turnaround Time: 13.00

Process returned 0 (0x0)   execution time : 23.352 s
Press any key to continue.
```

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

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

CODE:

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

#define MAX_PROCESSES

struct Process {
    int pid;
    int arrival_time;
    int burst_time;
    int priority;
    int remaining_time;
    int turnaround_time;
    int waiting_time;
};

void priority_nonpreemptive(struct Process processes[], int n) {
    // Sort the processes based on priority in ascending order
    int i,j,count=0,m;
    for(i=0;i<n;i++)
    {
        if(processes[i].arrival_time==0)
            count++;
    }
    if(count==n | count==1)
    {
        if(count==n)
        {
            for (i = 0; i < n - 1; i++) {
                for (j = 0; j < n - i - 1; j++) {
                    if (processes[j].priority > processes[j + 1].priority) {
                        struct Process temp = processes[j];
                        processes[j] = processes[j + 1];
                        processes[j + 1] = temp;
                    }
                }
            }
        }
    }
}
```

```

    }
    }
}

else
{
    for (i = 1; i < n - 1; i++) {
        for (j = 1; j <= n - i - 1; j++) {
            if (processes[j].priority > processes[j + 1].priority) {
                struct Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
}

int total_time = 0;
double total_turnaround_time = 0;
double total_waiting_time = 0;

for (i = 0; i < n; i++) {
    total_time += processes[i].burst_time;
    processes[i].turnaround_time = total_time - processes[i].arrival_time;
    processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;

    total_turnaround_time += processes[i].turnaround_time;
    total_waiting_time += processes[i].waiting_time;
}

printf("Process\tTurnaround Time\tWaiting Time\n");
for (i = 0; i < n; i++) {
    printf("%d\t%d\t\t%d\n", processes[i].pid, processes[i].turnaround_time,
processes[i].waiting_time);
}

printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);
printf("Average Waiting Time: %.2f\n", total_waiting_time / n);
}

```

```

void priority_preemptive(struct Process processes[], int n) {
    int total_time = 0;
    int completed = 0;

    while (completed < n) {
        int highest_priority = -1;
        int next_process = -1;

        for (i = 0; i < n; i++) {
            if (processes[i].arrival_time <= total_time && processes[i].remaining_time > 0) {
                if (highest_priority == -1 || processes[i].priority < highest_priority) {
                    highest_priority = processes[i].priority;
                    next_process = i;
                }
            }
        }

        if (next_process == -1) {
            total_time++;
            continue;
        }

        processes[next_process].remaining_time--;
        total_time++;

        if (processes[next_process].remaining_time == 0) {
            completed++;
            processes[next_process].turnaround_time = total_time -
processes[next_process].arrival_time;
            processes[next_process].waiting_time =
processes[next_process].turnaround_time - processes[next_process].burst_time;
        }
    }

    double total_turnaround_time = 0;
    double total_waiting_time = 0;

    printf("Process\tTurnaround Time\tWaiting Time\n");
    for (i = 0; i < n; i++) {

```

```

        printf("%d\t%d\t\t%d\n", processes[i].pid, processes[i].turnaround_time,
processes[i].waiting_time);

        total_turnaround_time += processes[i].turnaround_time;
        total_waiting_time += processes[i].waiting_time;
    }

    printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);
    printf("Average Waiting Time: %.2f\n", total_waiting_time / n);
}

void round_robin(struct Process processes[], int n, int quantum) {
    int i, total_time = 0, completed = 0;

    while (completed < n) {
        for (i = 0; i < n; i++) {
            if (processes[i].remaining_time > 0) {
                if (processes[i].remaining_time > quantum) {
                    total_time += quantum;
                    processes[i].remaining_time -= quantum;
                } else {
                    total_time += processes[i].remaining_time;
                    processes[i].remaining_time = 0;
                    processes[i].turnaround_time = total_time - processes[i].arrival_time;
                    processes[i].waiting_time = processes[i].turnaround_time -
processes[i].burst_time;
                    completed++;
                }
            }
        }
    }
}

double total_turnaround_time = 0;
double total_waiting_time = 0;

printf("Process\tTurnaround Time\tWaiting Time\n");
for (i = 0; i < n; i++) {
    printf("%d\t%d\t\t\t%d\n", processes[i].pid, processes[i].turnaround_time,
processes[i].waiting_time);
    total_turnaround_time += processes[i].turnaround_time;
    total_waiting_time += processes[i].waiting_time;
}

```



```

    }

    printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);
    printf("Average Waiting Time: %.2f\n", total_waiting_time / n);
}

void main() {
    int n, quantum, i, choice;
    struct Process processes[MAX_PROCESSES];

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

    for (i = 0; i < n; i++) {
        printf("Process %d\n", i + 1);
        printf("Enter arrival time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Enter burst time: ");
        scanf("%d", &processes[i].burst_time);
        printf("Enter priority: ");
        scanf("%d", &processes[i].priority);
        processes[i].pid = i + 1;
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].turnaround_time = 0;
        processes[i].waiting_time = 0;
    }

    while (1) {
        printf("\nSelect a scheduling algorithm:\n");
        printf("1. Priority (Non-preemptive)\n");
        printf("2. Priority (Preemptive)\n");
        printf("3. Round Robin\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {

            case 1:
                printf("\nPriority Non-preemptive Scheduling:\n");
                priority_nonpreemptive(processes, n);

```

```

        break;
    case 2:
        printf("\nPriority Preemptive Scheduling:\n");
        priority_preemptive(processes, n);
        break;
    case 3:
        printf("\nRound Robin Scheduling:\n");
        printf("Enter the time quantum: ");
        scanf("%d", &quantum);
        round_robin(processes, n, quantum);
        break;
    case 4:
        exit(0);
    default:
        printf("Invalid choice!\n");
        break;
    }
}
}
}

```

OUTPUT

```

Enter the number of processes: 5
Process 1
Enter arrival time: 0
Enter burst time: 4
Enter priority: 4
Process 2
Enter arrival time: 1
Enter burst time: 3
Enter priority: 3
Process 3
Enter arrival time: 3
Enter burst time: 4
Enter priority: 1
Process 4
Enter arrival time: 6
Enter burst time: 2
Enter priority: 5
Process 5
Enter arrival time: 8
Enter burst time: 4
Enter priority: 2

Select a scheduling algorithm:
1. Priority (Non-preemptive)
2. Priority (Preemptive)
3. Round Robin
4. Exit
Enter your choice: 2

```

Enter your choice: 2

Priority Preemptive Scheduling:

Process	Turnaround Time	Waiting Time
1	15	11
2	7	4
3	4	0
4	11	9
5	4	0

Average Turnaround Time: 8.20

Average Waiting Time: 4.80

Select a scheduling algorithm:

1. Priority (Non-preemptive)
2. Priority (Preemptive)
3. Round Robin
4. Exit

Enter your choice: 1

Priority Non-preemptive Scheduling:

Process	Turnaround Time	Waiting Time
1	4	0
3	5	1
5	4	0
2	14	11
4	11	9

Average Turnaround Time: 7.60

Average Waiting Time: 4.20

```

Enter arrival time: 0
Enter burst time: 10
Enter priority: 1
Process 2
Enter arrival time: 1
Enter burst time: 9
Enter priority: 1
Process 3
Enter arrival time: 2
Enter burst time: 12
Enter priority: 1
Process 4
Enter arrival time: 3
Enter burst time: 6
Enter priority: 1

Select a scheduling algorithm:
1. Priority (Non-preemptive)
2. Priority (Preemptive)
3. Round Robin
4. Exit
Enter your choice: 3

Round Robin Scheduling:
Enter the time quantum: 3
Process Turnaround Time Waiting Time
1          34          24
2          29          20
3          35          23
4          21          15
Average Turnaround Time: 29.75
Average Waiting Time: 20.50

```

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

CODE:

```
#include <stdio.h>

#include<stdlib.h>

#include <stdbool.h>

#define MAX_QUEUE_SIZE 100

int totalTime=0;

int userProcess=0,systemProcess=0;

// Structure to represent a process

typedef struct {

int processID;

int arrivalTime;

int burstTime;

int remainingTime;

int priority; // 0 for system process, 1 for user process

} Process;

// Function to execute a process

void executeProcess(Process process) {

int i;

printf("Executing Process %d\n", process.processID);

// Simulating the execution time of the process

for (i = 1; i <= process.burstTime; i++) {

printf("Process %d: %d/%d\n", process.processID, i,

process.burstTime);
```

```

}

printf("Process %d executed\n", process.processID);

}

// Function to perform FCFS scheduling for a queue of processes
void scheduleFCFS(Process system[], Process user[]) {
    int i, j;
    for(i=0; i<systemProcess; i++)
    {
        for(j=i+1; j<systemProcess; j++)
        {
            if(system[i].arrivalTime>system[j].arrivalTime)
            {
                Process temp=system[i];
                system[i]=system[j];
                system[j]=temp;
            }
        }
    }
    for(i=0; i<userProcess; i++)
    {
        for(j=i+1; j<userProcess; j++)
        {
            if(user[i].arrivalTime>user[j].arrivalTime)
            {
                Process temp=user[i];
                user[i]=user[j];
                user[j]=temp;
            }
        }
    }
}

```

```

}
}
}
int completed=0;
int currentProcess=-1;
bool isUserProcess=false;
int size=userProcess+systemProcess;
while(1)
{
int count=0;
for(i=0;i<systemProcess;i++)
{
if(system[i].remainingTime<=0)
{
count++;
}
}
for(j=0;j<userProcess;j++)
{
if(user[j].remainingTime<=0)
{
count++;
}
}
if(count==size)
{
printf("\n end of processess");

```

```

exit(0);
}
for(i=0;i<systemProcess;i++)
{
if(totalTime>=system[i].arrivalTime &&
system[i].remainingTime>0)
{
currentProcess=i;
isUserProcess=false;
break;
}
}
if(currentProcess== -1)
{
for(j=0;j<userProcess;j++)
{
if(totalTime>=user[j].arrivalTime &&
user[j].remainingTime>0)
{
currentProcess=j;
isUserProcess=true;
break;
}
}
}
if(currentProcess== -1)
{

```



```

totalTime++;
printf("\n %d idle time...",totalTime);
if(totalTime==1000)
{
    exit(0);
}
continue;
}
if(isUserProcess==true)
{
    user[currentProcess].remainingTime--;
    printf("\n User process %d will excecute at %d
",user[currentProcess].processID,(totalTime));
    totalTime++;
    isUserProcess=false;
    currentProcess=-1;
    if(user[currentProcess].remainingTime==0)
    {
    }
    }else{
        completed++;
        int temp=totalTime;
        while(system[currentProcess].remainingTime--){
            totalTime++;
        }
        if(system[currentProcess].remainingTime==0)
        {

```

```

    completed++;
}

printf("\n System process %d will execute
from %d to %d ",system[currentProcess].processID,temp,(totalTime));
isUserProcess=false;
currentProcess=-1;
}
}
}

int main() {
    int numProcesses,i;
    Process processes[MAX_QUEUE_SIZE];
    // Reading the number of processes
    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);
    // Reading process details
    for (i = 0; i < numProcesses; i++) {
        printf("Process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &processes[i].arrivalTime);
        printf("Burst Time: ");
        scanf("%d", &processes[i].burstTime);
        printf("System(0)/User(1): ");
        scanf("%d", &processes[i].priority);
        processes[i].processID = i + 1;
        processes[i].remainingTime=processes[i].burstTime;
        if(processes[i].priority==1)

```

```

{
userProcess++;
}else{
systemProcess++;
}
}

Process systemQueue[MAX_QUEUE_SIZE];
int systemQueueSize = 0;
Process userQueue[MAX_QUEUE_SIZE];
int userQueueSize = 0;
for (i = 0; i < numProcesses; i++) {
if (processes[i].priority == 0) {
systemQueue[systemQueueSize++] = processes[i];
} else {
userQueue[userQueueSize++] = processes[i];
}
}

printf("Order of Excecution :\n");
scheduleFCFS(systemQueue,userQueue);
return 0;
}

```

OUTPUT

```
Enter the number of processes: 6
Process 1:
Arrival Time: 0
Burst Time: 3
System(0)/User(1): 0
Process 2:
Arrival Time: 2
Burst Time: 2
System(0)/User(1): 0
Process 3:
Arrival Time: 4
Burst Time: 4
System(0)/User(1): 1
Process 4:
Arrival Time: 4
Burst Time: 2
System(0)/User(1): 1
Process 5:
Arrival Time: 8
Burst Time: 2
System(0)/User(1): 0
Process 6:
Arrival Time: 10
Burst Time: 3
System(0)/User(1): 1
Order of Execution :

System process 1 will excecute from 0 to 3
System process 2 will excecute from 3 to 5
User process 3 will excecute at 5
User process 3 will excecute at 6
User process 3 will excecute at 7
System process 5 will excecute from 8 to 10
User process 3 will excecute at 10
User process 4 will excecute at 11
User process 4 will excecute at 12
User process 6 will excecute at 13
User process 6 will excecute at 14
User process 6 will excecute at 15
end of processess
```

Q)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>

int et[10], i, n, dl[10], p[10], ready[10], flag = 1;

int lcm(int a, int b) {
    int max = (a > b) ? a : b;
    while (1) {
        if (max % a == 0 && max % b == 0)
            return max;
        max++;
    }
}

int lcmArray(int arr[], int n) {
    int result = arr[0];
    for (int i = 1; i < n; i++) {
        result = lcm(result, arr[i]);
    }
    return result;
}

void mono() {
    int time = lcmArray(dl, n);
    int op = 0, pr = 0, pre = pr;
    while (op <= time) {
```

```

for (i = 0; i < n; i++) {
    if (op % dl[i] == 0) {
        ready[i] = 1;
    }
}

flag = 0;
for (i = 0; i < n; i++) {
    if (ready[i] == 1) {
        flag = 1;
        break;
    }
}

if (flag == 0) {
    pr = -1;
} else {
    pr = -1;
    for (i = 0; i < n; i++) {
        if (ready[i] == 1) {
            if (pr == -1 || dl[i] < dl[pr]) {
                pr = i;
            }
        }
    }
}

if (pr != pre) {
    if (pr == -1) {
        printf("%d Idle ",op);
    } else {

```

```

        printf("%d P%d ",op, pr + 1);
    }
    op++;
    if (pr != -1) {
        p[pr] = p[pr] - 1;
        if (p[pr] == 0) {
            p[pr] = et[pr];
            ready[pr] = 0;
        }
    }

    pre = pr;
}

printf("\n");
}

void edf() {
    int time = lcmArray(dl, n);
    int op = 0, pr = 0, pre = -1;
    int flag, i;

    while (op <= time) {
        for (i = 0; i < n; i++) {
            if (op % dl[i] == 0) {
                ready[i] = 1;
            }
        }
    }
}

```

```

flag = 0;
for (i = 0; i < n; i++) {
    if (ready[i] == 1) {
        flag = 1;
        break;
    }
}

if (flag == 0) {
    pr = -1;
} else {
    pr = -1;
    for (i = 0; i < n; i++) {
        if (ready[i] == 1) {
            if (pr == -1 || p[i] < p[pr]) {
                pr = i;
            }
        }
    }
}

if (pr != pre) {
    if (pr == -1) {
        printf("%d Idle ", op);
    } else {
        printf("%d P%d ", op, pr + 1);
    }
}

```



```

    }
}

op++;
if (pr != -1) {
    p[pr] = p[pr] - 1;
    if (p[pr] == 0) {
        p[pr] = et[pr];
        ready[pr] = 0;
    }
}

pre = pr;
}

printf("\n");
}
int main() {
    int ch, k = 1;
    while (k) {
        printf("Enter your choice: \n1. Monotonic \n2. EDF \n3. Exit\n");
        scanf("%d", &ch);
        if(ch==4)
            exit(0);
        printf("Enter the number of processes: ");
        scanf("%d", &n);
        printf("Enter execution times: \n");

```

```
    for (i = 0; i < n; i++)
        scanf("%d", &et[i]);

printf("Enter deadlines: \n");

    for (i = 0; i < n; i++)
        scanf("%d", &dl[i]);

    for (i = 0; i < n; i++)
        p[i] = et[i];

    for (i = 0; i < n; i++)
        ready[i] = 0;

    switch (ch) {
        case 1:
            mono();
            break;

        case 2:
            edf();
            break;

        case 3:
            k = 0;
            break;

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

```
}  
  
    return 0;  
  
}
```

OUTPUT

```
Enter your choice:  
1. Monotonic  
2. EDF  
3. Exit  
1  
Enter the number of processes: 3  
Enter execution times:  
3 2 2  
Enter deadlines:  
20 5 10  
0 P2 2 P3 4 P1 5 P2 7 P1 9 Idle 10 P2 12 P3 14 Idle 15 P2 17 Idle 20 P2
```

```
Enter your choice:  
1. Monotonic  
2. EDF  
3. Exit  
2  
Enter the number of processes: 3  
Enter execution times:  
3 2 2  
Enter deadlines:  
20 5 10  
0 P2 2 P3 4 P1 7 P2 9 Idle 10 P2 12 P3 14 Idle 15 P2 17 Idle 20 P2
```

Q)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 \n 2.Consumer \n");

    while(1)
    {
        printf("Enter your choice:");
        scanf("%d",&n);
        switch(n)
        {
            case 1:if((mutex==1)&&(empty!=0))
                producer();
            else
                printf("Buffer is full \n");
            break;
```

```

        case 2:if((mutex==1)&&(full!=0))
            consumer();
        else
            printf("Buffer is empty \n");
            break;
        case 3:exit(0);break;

    }
}
return 0;
} //main

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("\n Producer produces item %d \n",x);
    mutex=signal(mutex);

```

```

}

//producer

void consumer()
{
    mutex=wait(mutex);
    full=wait(full);
    empty=signal(empty);
    printf("\nConsumer consumes item %d \n",x);
    x--;
    mutex=signal(mutex);
}
//consumer

```

OUTPUT

```

1.Producer
2.Consumer
Enter your choice:1

    Producer produces item 1
Enter your choice:1

    Producer produces item 2
Enter your choice:1

    Producer produces item 3
Enter your choice:1
Buffer is full
Enter your choice:2

Consumer consumes item 3
Enter your choice:2

Consumer consumes item 2
Enter your choice:2

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

```

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

CODE:

```
#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

```
Enter number of philosophers:5
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 4 is Hungry
Philosopher 2 is Hungry
Philosopher 5 is Hungry
Philosopher 3 is Hungry
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 1 is Hungry
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 3 is Hungry
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
```

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

CODE:

```
#include<stdio.h>

struct file
{
    int all[10];
    int max[10];
    int need[10];
    int flag;
};

void main()
{
    struct file f[10];
    int fl;
    int i, j, k, p, b, n, r, g, cnt=0, id, newr;
    int avail[10],seq[10];

    printf("Enter number of processes : ");
    scanf("%d",&n);
    printf("Enter number of resources : ");
    scanf("%d",&r);
    for(i=0;i<n;i++)
    {
        printf("Enter details for P%d",i);
        printf("\nEnter allocation\t : \t");
```

```

for(j=0;j<r;j++)
scanf("%d",&f[i].all[j]);
printf("Enter Max\t\t: \t");
for(j=0;j<r;j++)
    scanf("%d",&f[i].max[j]);
f[i].flag=0;
}
printf("\nEnter Available Resources\t: \t");
for(i=0;i<r;i++)
scanf("%d",&avail[i]);

```

```

printf("\nEnter New Request Details :");
printf("\nEnter pid \t -- \t");
scanf("%d",&id);
printf("Enter Request for Resources \t : \t");
for(i=0;i<r;i++)
{
scanf("%d",&newr);
f[id].all[i] += newr;
avail[i]=avail[i] - newr;
}
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{

```

```

f[i].need[j]=f[i].max[j]-f[i].all[j];
if(f[i].need[j]<0)
f[i].need[j]=0;
}
}

cnt=0; fl=0;
while(cnt!=n)
{ g=0;
for(j=0;j<n;j++)
{
if(f[j].flag==0)
{ b=0;
for(p=0;p<r;p++)
{
if(avail[p]>=f[j].need[p]) b=b+1;
else b=b-1;
}
if(b==r)
{
printf("\nP%d is visited",j);
seq[fl++]=j;

f[j].flag=1;
for(k=0;k<r;k++)
    avail[k]=avail[k]+f[j].all[k];
cnt=cnt+1;

```

```

printf("");
for(k=0;k<r;k++)
    printf("%3d",avail[k]);
printf("");
g=1;
}
}
}
if(g==0)
{
printf("\n REQUEST NOT GRANTED -- DEADLOCK OCCURRED");
printf("\n SYSTEM IS IN UNSAFE STATE");
goto y;
}
}
printf("\nSYSTEM IS IN SAFE STATE");
printf("\nThe Safe Sequence is -- (");
for(i=0;i<fl;i++)
printf("P%d ",seq[i]);
printf(")");
y: printf("\nProcess\t\tAllocation\t\tMax\t\tNeed\n");
for(i=0;i<n;i++)

{
printf("P%d\t",i);
for(j=0;j<r;j++)
    printf("%5d",f[i].all[j]);

```



```

for(j=0;j<r;j++)
    printf("%5d",f[i].max[j]);
for(j=0;j<r;j++)
    printf("%5d",f[i].need[j]);
printf("\n");
}
}

```

OUTPUT

```

Enter details for P3
Enter allocation      --      2 1 1
Enter Max             --      2 2 2
Enter details for P4
Enter allocation      --      0 0 2
Enter Max             --      4 3 3

Enter Available Resources      :      3 3 2

Enter New Request Details :
Enter pid      --      1
Enter Request for Resources      :      1 0 2

P1 is visited( 5 3 2)
P3 is visited( 7 4 3)
P4 is visited( 7 4 5)
P0 is visited( 7 5 5)
P2 is visited( 10 5 7)
SYSTEM IS IN SAFE STATE
The Safe Sequence is -- (P1 P3 P4 P0 P2 )
Process      Allocation      Max      Need
P0           0   1   0   7   5   3   7   4   3
P1           3   0   2   3   2   2   0   2   0
P2           3   0   2   9   0   2   6   0   0
P3           2   1   1   2   2   2   0   1   1
P4           0   0   2   4   3   3   4   3   1

Process returned 5 (0x5)   execution time : 65.811 s
Press any key to continue.

```

Q)Write a C program to simulate deadlock detection

CODE:

```
#include <stdio.h>

#define MAX_PROCESSES 5
#define MAX_RESOURCES 3

int allocated[MAX_PROCESSES][MAX_RESOURCES];
int requested[MAX_PROCESSES][MAX_RESOURCES];
int available[MAX_RESOURCES];
int work[MAX_RESOURCES];
int finish[MAX_PROCESSES];

void initialize()
{
    // Initialize allocated and requested matrices
    for (int i = 0; i < MAX_PROCESSES; i++)
    {
        printf("Enter allocated resources for process P%d:\n", i);
        for (int j = 0; j < MAX_RESOURCES; j++)
            scanf("%d", &allocated[i][j]);

        printf("Enter requested resources for process P%d:\n", i);
        for (int j = 0; j < MAX_RESOURCES; j++)
            scanf("%d", &requested[i][j]);

        finish[i] = 0; // Process is not finished yet
    }
}
```

```

    }
}

int checkSafety()
{
    for (int i = 0; i < MAX_RESOURCES; i++)
        work[i] = available[i];

    int count = 0;
    while (count < MAX_PROCESSES)
    {
        int found = 0;
        for (int i = 0; i < MAX_PROCESSES; i++)
        {
            if (!finish[i])
            {
                int j;
                for (j = 0; j < MAX_RESOURCES; j++)
                {
                    if (requested[i][j] > work[j])
                        break;
                }
                if (j == MAX_RESOURCES)
                {
                    for (int k = 0; k < MAX_RESOURCES; k++)
                        work[k] += allocated[i][k];
                    finish[i] = 1;
                }
            }
        }
        count++;
    }
}

```

```

        found = 1;
        count++;
    }
}
}
if (!found)
    break;
}

return count == MAX_PROCESSES;
}

int main()
{
    initialize();

    // Assume available resources are initially zero
    for (int i = 0; i < MAX_RESOURCES; i++)
        available[i] = 0;

    if (checkSafety())
        printf("System is in safe state.\n");
    else
        printf("System is in unsafe state.\n");

    return 0;
}

```

OUTPUT

```
Enter allocated resources for process P0:
0 1 0
Enter requested resources for process P0:
0 0 0
Enter allocated resources for process P1:
2 0 0
Enter requested resources for process P1:
2 0 2
Enter allocated resources for process P2:
3 0 3
Enter requested resources for process P2:
0 0 0
Enter allocated resources for process P3:
2 1 1
Enter requested resources for process P3:
1 0 0
Enter allocated resources for process P4:
0 0 2
Enter requested resources for process P4:
0 0 2
System is in safe state.
```

```
Enter allocated resources for process P0:
0 1 0
Enter requested resources for process P0:
0 0 0
Enter allocated resources for process P1:
2 0 0
Enter requested resources for process P1:
2 0 2
Enter allocated resources for process P2:
3 0 3
Enter requested resources for process P2:
0 0 1
Enter allocated resources for process P3:
2 1 1
Enter requested resources for process P3:
1 0 0
Enter allocated resources for process P4:
0 0 2
Enter requested resources for process P4:
0 0 2
System is in unsafe state.
```

Q)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>

#include <conio.h>

#define max 25

int frag[max], b[max], f[max], nf, nb;
int bf[max], ff[max];

void firstfit() {
    int i, j, temp;
    static int bf[max];

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

```

    }

    frag[i] = temp;

    bf[ff[i]] = 1;
}

printf("\nFile_size:\tBlock_size:");

for (i = 1; i <= nf; i++) {
    printf("\n%d\t\t%d", f[i], b[ff[i]]);
}
}

void bestfit() {
    int i, j, temp, lowest = 10000;
    static int bf[max];

    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 Size:\tBlock Size:");
    for (i = 1; i <= nf && ff[i] != 0; i++) {
        printf("\n%d\t\t%d", f[i], b[ff[i]]);
    }
}

void worstfit() {
    int i, j, temp, highest = 0;
    static int bf[max];

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

int main() {
    int c;

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

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

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

```

```
while (1) {  
    printf("\n1. First Fit 2. Best Fit 3. Worst Fit 4. Exit");  
    printf("\nEnter choice:");  
    scanf("%d", &c);  
    switch (c) {  
        case 1:  
            firstfit();  
            break;  
        case 2:  
            bestfit();  
            break;  
        case 3:  
            worstfit();  
            break;  
        case 4:  
            return 0;  
        default:  
            printf("Invalid choice");  
    }  
}  
}
```

OUTPUT

```
Enter the number of blocks:8
Enter the number of files:3
Enter the size of the blocks:
Block 1:10000
Block 2:4000
Block 3:20000
Block 4:18000
Block 5:7000
Block 6:9000
Block 7:12000
Block 8:15000
Enter the size of the files:
File 1:12000
File 2:10000
File 3:9000

1. First Fit 2. Best Fit 3. Worst Fit 4. Exit
Enter choice:1

File_size:      Block_size:
12000           20000
10000           10000
9000            18000
1. First Fit 2. Best Fit 3. Worst Fit 4. Exit
Enter choice:2

File Size:      Block Size:
12000           12000
10000           10000
9000            9000
1. First Fit 2. Best Fit 3. Worst Fit 4. Exit
Enter choice:3

File_size:      Block_size:
12000           20000
10000           18000
9000            15000
```

Q)Write a C program to simulate page replacement algorithms

a) FIFO

b) LRU

c) Optimal

CODE:

```
#include<stdio.h>
int n,nf;
int in[100];
int p[50];
int hit=0;
int i,j,k;
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",&nf);
}

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

int isHit(int data)
{
    hit=0;
    for(j=0; j<nf; j++)
    {
```

```

        if(p[j]==data)
        {
            hit=1;
            break;
        }

    }

    return hit;
}

int getHitIndex(int data)
{
    int hitind;
    for(k=0; k<nf; k++)
    {
        if(p[k]==data)
        {
            hitind=k;
            break;
        }
    }
    return hitind;
}

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

}

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

```

```

}

void fifo()
{
    initialize();
    for(i=0; i<n; i++)
    {
        printf("\nFor %d :",in[i]);

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

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

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

```

```

void optimal() //replace the page that will be used in the most layer point of time
{
    initialize();
    int near[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; 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\n7.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
7.Exit
Enter your choice:1

Enter length of page reference sequence:14

Enter the page reference sequence:0 4 3 2 1 4 6 3 0 8 9 3 8 5

Enter no of frames:3

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

For 0 : 0
For 4 : 0 4
For 3 : 0 4 3
For 2 : 4 3 2
For 1 : 3 2 1
For 4 : 2 1 4
For 6 : 1 4 6
For 3 : 4 6 3
For 0 : 6 3 0
For 8 : 3 0 8
For 9 : 0 8 9
For 3 : 8 9 3
For 8 :No page fault
For 5 : 9 3 5
Total no of page faults:13

```

Page Replacement Algorithms

1.Enter data

2.FIFO

3.Optimal

4.LRU

7.Exit

Enter your choice:3

For 0 : 0

For 4 : 0 4

For 3 : 0 4 3

For 2 : 2 4 3

For 1 : 1 4 3

For 4 :No page fault

For 6 : 6 4 3

For 3 :No page fault

For 0 : 0 4 3

For 8 : 8 4 3

For 9 : 8 9 3

For 3 :No page fault

For 8 :No page fault

For 5 : 5 9 3

Total no of page faults:10

Enter your choice:4

For 0 : 0

For 4 : 0 4

For 3 : 0 4 3

For 2 : 2 4 3

For 1 : 2 1 3

For 4 : 2 1 4

For 6 : 6 1 4

For 3 : 6 3 4

For 0 : 6 3 0

For 8 : 8 3 0

For 9 : 8 9 0

For 3 : 8 9 3

For 8 :No page fault!

For 5 : 8 5 3

Total no of page faults:13

Q)Write a C program Write a C program to simulate disk scheduling algorithms

a) FCFS

b) SCAN

c) C-SCAN

a) SSTF

b) LOOK

c) c-LOOK

CODE:

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

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

```
int m, n, start; // Global variables for disk specifications
```

```
int a[15]; // Global array for the request queue
```

```
int absolute(int a, int b)
```

```
{
```

```
    int c = a - b;
```

```
    if (c < 0)
```

```
        return -c;
```

```
    else
```

```
        return c;
```

```
}
```

```
void fcfs()
```

```
{
```

```
    printf("\nFCFS:\n");
```

```
    int count = 0;
```

```
    int x = start;
```

```
    printf("Scheduling services the request in the order that follows:\n%d\t", start);
```

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

```
    {
```

```
        x -= a[i];
```

```
        if (x < 0)
```

```
            x = -x;
```

```

        count += x;
        x = a[i];
        printf("%d\t", x);
    }
    printf("\nTotal Head Movement: %d Cylinders\n", count);
}

```

```

void sstf()
{
    printf("\nSSTF:\n");
    int count = 0;
    int x = start;
    printf("Scheduling services the request in the order that follows:\n%d\t", start);
    for (int i = 0; i < n; i++)
    {
        int min = absolute(a[i], x);
        int pos = i;
        for (int j = i; j < n; j++)
        {
            if (min > absolute(x, a[j]))
            {
                pos = j;
                min = absolute(x, a[j]);
            }
        }
        count += absolute(x, a[pos]);
        x = a[pos];
        a[pos] = a[i];
        a[i] = x;
        printf("%d\t", x);
    }
    printf("\nTotal Head Movement: %d Cylinders\n", count);
}

```

```

//scan
void scan(int direction)
{
    printf("\nSCAN:\n");

```

```

int count = 0;
int pos = 0;

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

for (int i = 0; i < n; i++)
{
    if (a[i] < start)
        pos++;
}

int x = start;

if (direction == 1) // Right direction
{
    for (int i = pos; i < n; i++)
    {
        count += absolute(a[i], x);
        x = a[i];
        printf("%d\t", x);
    }
    if (x != m - 1)
    {
        count += absolute(x, m - 1);
        x = m - 1;
        printf("%d\t", x);
    }
}

```

```

        for (int i = pos - 1; i >= 0; i--)
        {
            count += absolute(a[i], x);
            x = a[i];
            printf("%d\t", x);
        }
    }
    else // Left direction
    {
        for (int i = pos - 1; i >= 0; i--)
        {
            count += absolute(a[i], x);
            x = a[i];
            printf("%d\t", x);
        }
        if (x != 0)
        {
            count += absolute(x, 0);
            x = 0;
            printf("%d\t", x);
        }
        for (int i = pos; i < n; i++)
        {
            count += absolute(a[i], x);
            x = a[i];
            printf("%d\t", x);
        }
    }

    printf("\nTotal Head Movement: %d Cylinders\n", count);
}

void look(int direction)
{
    printf("\nLOOK:\n");
    int count = 0;
    int pos = 0;

```

```

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

for (int i = 0; i < n; i++)
{
    if (a[i] < start)
        pos++;
}

int x = start;

if (direction == 1) // Right direction
{
    for (int i = pos; i < n; i++)
    {
        count += absolute(a[i], x);
        x = a[i];
        printf("%d\t", x);
    }
    for (int i = pos - 1; i >= 0; i--)
    {
        count += absolute(a[i], x);
        x = a[i];
        printf("%d\t", x);
    }
}
else // Left direction
{

```



```

    for (int i = pos - 1; i >= 0; i--)
    {
        count += absolute(a[i], x);
        x = a[i];
        printf("%d\t", x);
    }
    for (int i = pos; i < n; i++)
    {
        count += absolute(a[i], x);
        x = a[i];
        printf("%d\t", x);
    }
}

printf("\nTotal Head Movement: %d Cylinders\n", count);
}

```

```

void cscan(int direction)
{
    printf("\nC-SCAN:\n");
    int count = 0;
    int pos = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (a[j] > a[j + 1])
            {
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
        }
    }
    for (int i = 0; i < n; i++)
    {

```

```

    if (a[i] < start)
        pos++;
}

int x = start;

if (direction == 1) // Right direction
{
    for (int i = pos; i < n; i++)
    {
        count += absolute(x, a[i]);
        x = a[i];
        printf("%d\t", x);
    }
    count += absolute(m - 1, x);
    x = 0;
    printf("%d\t%d\t", m - 1, 0);
    for (int i = 0; i < pos; i++)
    {
        count += absolute(x, a[i]);
        x = a[i];
        printf("%d\t", x);
    }
}
else // Left direction
{
    for (int i = pos - 1; i >= 0; i--)
    {
        count += absolute(x, a[i]);
        x = a[i];
        printf("%d\t", x);
    }
    count += absolute(0, x);
    x = m - 1;
    printf("%d\t%d\t", 0, x);
    for (int i = n - 1; i >= pos; i--)
    {
        count += absolute(x, a[i]);

```

```

        x = a[i];
        printf("%d\t", x);
    }
}

printf("\nTotal Head Movement: %d Cylinders\n", count);
}

```

```

void clook(int direction)
{
    printf("\nC-LOOK:\n");
    int count = 0;
    int pos = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (a[j] > a[j + 1])
            {
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
        }
    }
    for (int i = 0; i < n; i++)
    {
        if (a[i] < start)
            pos++;
    }

    int x = start;

    if (direction == 1) // Right direction
    {
        for (int i = pos; i < n; i++)
        {

```

```

        count += absolute(x, a[i]);
        x = a[i];
        printf("%d\t", x);
    }
    for (int i = 0; i < pos; i++)
    {
        count += absolute(x, a[i]);
        x = a[i];
        printf("%d\t", x);
    }
}
else // Left direction
{
    for (int i = pos - 1; i >= 0; i--)
    {
        count += absolute(x, a[i]);
        x = a[i];
        printf("%d\t", x);
    }
    for (int i = n - 1; i >= pos; i--)
    {
        count += absolute(x, a[i]);
        x = a[i];
        printf("%d\t", x);
    }
}

printf("\nTotal Head Movement: %d Cylinders\n", count);
}

int main()
{
    int choice, direction;

    printf("Enter the number of cylinders: ");
    scanf("%d", &m);

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

```

```

scanf("%d", &n);

printf("Enter current position: ");
scanf("%d", &start);

printf("Enter the request queue: ");
for (int i = 0; i < n; i++)
{
    scanf("%d", &a[i]);
    if (a[i] >= m)
    {
        printf("\nInvalid input, re-enter: ");
        scanf("%d", &a[i]);
    }
}

printf("Enter the direction (1 for Right, 0 for Left): ");
scanf("%d", &direction);

do
{
    printf("\n\nDISK SCHEDULING ALGORITHMS\n1. FCFS\n2. SSTF\n3. SCAN\n4. C-SCAN\n5.
LOOK\n6. C-LOOK\n");
    printf("Enter choice: ");
    scanf("%d", &choice);

    switch (choice)
    {
    case 1:
        fcfs();
        break;
    case 2:
        sstf();
        break;
    case 3:
        scan(direction);
        break;
    case 4:

```

```

        cscan(direction);
        break;
case 5:
    look(direction);
    break;
case 6:
    clook(direction);
    break;
default:
    printf("Invalid choice\n");
}

printf("Do you want to continue? (1 to continue): ");
scanf("%d", &choice);
} while (choice == 1);

return 0;
}

```

OUTPUT-

```

Enter the number of cylinders: 200
Enter the number of requests: 8
Enter current position: 53
Enter the request queue: 98 183 37 122 14 124 65 67
Enter the direction (1 for Right, 0 for Left): 1

DISK SCHEDULING ALGORITHMS
1. FCFS
2. SSTF
3. SCAN
4. C-SCAN
5. LOOK
6. C-LOOK
Enter choice: 1

FCFS:
Scheduling services the request in the order that follows:
53    98    183    37    122    14    124    65    67
Total Head Movement: 640 Cylinders
Do you want to continue? (1 to continue): 1

DISK SCHEDULING ALGORITHMS
1. FCFS
2. SSTF
3. SCAN
4. C-SCAN
5. LOOK
6. C-LOOK
Enter choice: 2

SSTF:
Scheduling services the request in the order that follows:
53    65    67    37    14    98    122    124    183

```

```
Total Head Movement: 236 Cylinders
Do you want to continue? (1 to continue): 1
```

DISK SCHEDULING ALGORITHMS

1. FCFS
2. SSTF
3. SCAN
4. C-SCAN
5. LOOK
6. C-LOOK

Enter choice: 3

SCAN:

65 67 98 122 124 183 199 37 14

Total Head Movement: 331 Cylinders

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

DISK SCHEDULING ALGORITHMS

1. FCFS
2. SSTF
3. SCAN
4. C-SCAN
5. LOOK
6. C-LOOK

Enter choice: 4

C-SCAN:

65 67 98 122 124 183 199 0 14 37

Total Head Movement: 183 Cylinders

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

DISK SCHEDULING ALGORITHMS

1. FCFS
2. SSTF

Enter choice: 4

C-SCAN:

65 67 98 122 124 183 199 0 14 37

Total Head Movement: 183 Cylinders

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

DISK SCHEDULING ALGORITHMS

1. FCFS
2. SSTF
3. SCAN
4. C-SCAN
5. LOOK
6. C-LOOK

Enter choice: 5

LOOK:

65 67 98 122 124 183 37 14

Total Head Movement: 299 Cylinders

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

DISK SCHEDULING ALGORITHMS

1. FCFS
2. SSTF
3. SCAN
4. C-SCAN
5. LOOK
6. C-LOOK

Enter choice: 6

C-LOOK:

65 67 98 122 124 183 14 37

Total Head Movement: 322 Cylinders

