VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT

on

Operating Systems

(23CS4PCOPS)

Submitted by:

Sudarshan Komar (1BM22CS291)

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 **Sudarshan Komar (1BM22CS291),** 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.

Sonika Sharma D
Assistant Professor
Department of CSE
BMSCE, Bengaluru

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

Table Of Contents

Lab Program No.	Program Details	Page No.
1	FCFS, SJF AND SRTF	2-7
2	PRIORITY AND ROUND ROBIN	8-17
3	.MULTILEVEL QUEUE SCHEDULING	18-20
4	RATE-MONOTONIC AND EARLIEST DEADLINE FIRST	21-28
5	PRODUCER-CONSUMER PROBLEM	29-30
6	DINERS-PHILOSOPHERS PROBLEM	31-32
7	BANKERS ALGORITHM (DEADLOCK AVOIDANCE)	33-35
8	DEADLOCK DETECTION	36-38
9	CONTIGUOUS MEMORY ALLOCATION (FIRST, BEST, WORSTFIT)	39-43
10	PAGE REPLACEMENT (FIFO, LRU, OPTIMAL)	44-48

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

(c) SRTF

CODE:

```
#include <stdio.h>
#include <limits.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("\nAverage Turnaround Time = %.2f\n", avg_Turn_around_time);
}
void SRTF()
{
    int total_waiting_time = 0, total_turnaround_time = 0;
    int completed = 0, current_time = 0, min_index = -1;
    int Remaining_time[20], is_completed[20] = {0};
    for (i = 0; i < n; i++)
    {
        Remaining_time[i] = Burst_time[i];
    }
    while (completed != n)
    {
        int min_burst_time = INT_MAX;
        for (i = 0; i < n; i++)
        {
            if (Arrival_time[i] <= current_time && is_completed[i] == 0)
            {
                if (Remaining_time[i] < min_burst_time)
                {
                    min_burst_time = Remaining_time[i];
                    min_index = i;
                }
                if (Remaining_time[i] == min_burst_time)
                {
                    if (Arrival_time[i] < Arrival_time[min_index])
                    {
                        min_burst_time = Remaining_time[i];
                        min_index = i;
                    }
                }
```

```
            }
        }
        if (min_index != -1)
        {
            Remaining_time[min_index]--;
            current_time++;
            if (Remaining_time[min_index] == 0)
            {
                is_completed[min_index] = 1;
                completed++;
                Turn_around_time[min_index] = current_time - Arrival_time[min_index];
                Waiting_time[min_index] = Turn_around_time[min_index] - Burst_time[min_index];
                total_waiting_time += Waiting_time[min_index];
                total_turnaround_time += Turn_around_time[min_index];
                min_index = -1;
            }
        }
        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("\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\n3. SRTF Scheduling\n");
```

```
        printf("\nEnter your choice: ");
        scanf("%d", &choice);
        switch (choice)
        {
        case 1:
            FCFS();
            break;
        case 2:
            SJF();
            break;
        case 3:
            SRTF();
            break;
        default:
            printf("Invalid Input!!!\n");
        }
    }
    return 0;
}
```

OUTPUTS:

```
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
3. SRTF 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
```

```
-----MAIN MENU-----
1. FCFS Scheduling
2. SJF Scheduling
3. SRTF Scheduling

Enter your choice: 2

Process         Arrival Time    Burst Time      Waiting Time    Turnaround Time
P[1]            0               10              1               11
P[2]            0               1               0               1
P[3]            3               2               9               11
P[4]            5               1               6               7
P[5]            10              5               4               9

Average Waiting Time = 4.00
Average Turnaround Time = 7.80
```

```
-----MAIN MENU-----
1. FCFS Scheduling
2. SJF Scheduling
3. SRTF Scheduling

Enter your choice: 3

Process         Arrival Time    Burst Time      Waiting Time    Turnaround Time
P[1]            0               10              4               14
P[2]            0               1               0               1
P[3]            3               2               0               2
P[4]            5               1               0               1
P[5]            10              5               4               9

Average Waiting Time = 1.60
Average Turnaround Time = 5.40
```

LAB-2

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

(a) Priority (Non-pre-emptive & Pre-emptive)

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

CODE:

<u>Priority (Non-pre-emptive)</u> <u>#lower value higher priority</u>

```
#include <stdio.h>
#include <stdlib.h>
struct process
{
    int process_id;
    int burst_time;
    int priority;
    int arrival_time;
    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);
        printf("Enter the arrival time: ");
        scanf("%d", &proc[i].arrival_time);
    }
    priority_scheduling(proc, n);

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

```
    int current_time = 0;
    wt[0] = 0;
    current_time = proc[0].arrival_time + proc[0].burst_time;
    for (i = 1; i < n; i++)
    {
        if (current_time < proc[i].arrival_time)
        {
            current_time = proc[i].arrival_time;
        }
        wt[i] = current_time - proc[i].arrival_time;
        current_time += proc[i].burst_time;
    }
}
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\tArrival Time\tBurst Time\tPriority\tWaiting Time\tTurnaround Time");
    for (i = 0; i < n; i++)
    {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d", proc[i].process_id, proc[i].arrival_time,
               proc[i].burst_time, proc[i].priority, wt[i], tat[i]);
    }
    printf("\n\nAverage Waiting Time = %f", (float)total_wt / n);
    printf("\nAverage Turnaround Time = %f\n", (float)total_tat / n);
}
void priority_scheduling(struct process proc[], int n)
{

    int i, j, pos;
    struct process temp;
    // Sort based on arrival time
    for (i = 0; i < n - 1; i++)
    {
        for (j = i + 1; j < n; j++)
        {
            if (proc[i].arrival_time > proc[j].arrival_time)
            {
                temp = proc[i];
                proc[i] = proc[j];
                proc[j] = temp;
```

```
            }
        }
    }
    // Sort based on priority (for processes with the same arrival time)
    for (i = 0; i < n - 1; i++)
    {
        pos = i;
        for (j = i + 1; j < n; j++)
        {
            if (proc[j].arrival_time <= proc[i].arrival_time && proc[j].priority < proc[pos].priority)
            {
                pos = j;
            }
        }
        if (pos != i)
        {
            temp = proc[i];
            proc[i] = proc[pos];
            proc[pos] = temp;
        }
    }
    find_average_time(proc, n);
}
```

OUTPUTS:

```
Enter the number of processes: 5

Enter the process ID: 1
Enter the burst time: 4
Enter the priority: 2
Enter the arrival time: 0

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

Enter the process ID: 3
Enter the burst time: 1
Enter the priority: 4
Enter the arrival time: 2

Enter the process ID: 4
Enter the burst time: 5
Enter the priority: 5
Enter the arrival time: 3

Enter the process ID: 5
Enter the burst time: 2
Enter the priority: 5
Enter the arrival time: 4

Process ID      Arrival Time    Burst Time      Priority        Waiting Time    Turnaround Time
1               0               4               2               0               4
2               1               3               3               3               6
3               2               1               4               5               6
4               3               5               5               5               10
5               4               2               5               9               11

Average Waiting Time = 4.400000
Average Turnaround Time = 7.400000
```

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\t%d\t\t%d\t\t%d\t\t%d\t\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("\nAverage Turnaround Time = %f\n", (float)total_tat / n);
}
void priority_scheduling(struct process proc[], int n)
{
    find_average_time(proc, n);
}
```

OUTPUTS:

```
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)

Code:

```
#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\t%d\t\t%d\t\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 a 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>
#define MAX_PROCESSES 50
void sort(int proc_id[], int at[], int bt[], int n)
{
    int temp;
    for (int i = 0; i < n; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            if (at[j] < at[i])
            {
                // Swap arrival times
                temp = at[i];
                at[i] = at[j];
                at[j] = temp;
                // Swap burst times
                temp = bt[i];
                bt[i] = bt[j];
                bt[j] = temp;
                // Swap process IDs
                temp = proc_id[i];
                proc_id[i] = proc_id[j];
                proc_id[j] = temp;
            }
        }
    }
}
void fcfs(int at[], int bt[], int ct[], int tat[], int wt[], int n, int *c)
{
    double ttat = 0.0, twt = 0.0;
    // Completion time
    for (int i = 0; i < n; i++)
    {
        if (*c >= at[i])
        {
            *c += bt[i];
        }
        else
        {
            *c = at[i] + bt[i];
```

```
        }
        ct[i] = *c;
    }

    // Turnaround time
    for (int i = 0; i < n; i++)
    {
        tat[i] = ct[i] - at[i];
    }
    // Waiting time
    for (int i = 0; i < n; i++)
    {
        wt[i] = tat[i] - bt[i];
    }
}
int main()
{
    int sn, un, c = 0;
    int n = 0;
    printf("Enter number of system processes: ");
    scanf("%d", &sn);
    n = sn;
    int sproc_id[MAX_PROCESSES], sat[MAX_PROCESSES], sbt[MAX_PROCESSES];
    int sct[MAX_PROCESSES], stat[MAX_PROCESSES], swt[MAX_PROCESSES];
    for (int i = 0; i < sn; i++)
    {
        sproc_id[i] = i + 1;
    }
    printf("Enter arrival times of the system processes:\n");
    for (int i = 0; i < sn; i++)
    {
        scanf("%d", &sat[i]);
    }
    printf("Enter burst times of the system processes:\n");
    for (int i = 0; i < sn; i++)
    {
        scanf("%d", &sbt[i]);
    }
    printf("Enter number of user processes: ");
    scanf("%d", &un);
    n = un;
    int uproc_id[MAX_PROCESSES], uat[MAX_PROCESSES], ubt[MAX_PROCESSES];
    int uct[MAX_PROCESSES], utat[MAX_PROCESSES], uwt[MAX_PROCESSES];
    for (int i = 0; i < un; i++)
    {
        uproc_id[i] = i + 1;
    }
    printf("Enter arrival times of the user processes:\n");
    for (int i = 0; i < un; i++)
    {
        scanf("%d", &uat[i]);
    }
```

```
    printf("Enter burst times of the user processes:\n");
    for (int i = 0; i < un; i++)
    {
        scanf("%d", &ubt[i]);
    }
    sort(sproc_id, sat, sbt, sn);
    sort(uproc_id, uat, ubt, un);
    fcfs(sat, sbt, sct, stat, swt, sn, &c);
    fcfs(uat, ubt, uct, utat, uwt, un, &c);
    printf("\nScheduling:\n");
    printf("System processes:\n");
    printf("PID\tAT\tBT\tCT\tTAT\tWT\n");
    for (int i = 0; i < sn; i++)
    {
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", sproc_id[i], sat[i], sbt[i], sct[i], stat[i], swt[i]);
    }
    printf("User processes:\n");
    printf("PID\tAT\tBT\tCT\tTAT\tWT\n");
    for (int i = 0; i < un; i++)
    {
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", uproc_id[i], uat[i], ubt[i], uct[i], utat[i], uwt[i]);
    }
    return 0;
}
```

OUTPUT:

```
Enter number of system processes: 2
Enter arrival times of the system processes:
0
0
Enter burst times of the system processes:
2
5
Enter number of user processes: 2
Enter arrival times of the user processes:
0
0
Enter burst times of the user processes:
1
3

Scheduling:
System processes:
PID     AT      BT      CT      TAT     WT
1       0       2       2       2       0
2       0       5       7       7       2
User processes:
PID     AT      BT      CT      TAT     WT
1       0       1       8       8       7
2       0       3       11      11      8
```

LAB - 4

Question 1:

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

(a) Rate Monotonic

(b) Earliest-deadline First

CODE:

(a) Rate monotonic

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
void sort(int proc[], int b[], int pt[], int n)
{
	int temp = 0;
	for (int i = 0; i < n; i++)
	{
		for (int j = i; j < n; j++)
		{
			if (pt[j] < pt[i])
			{
				temp = pt[i];
				pt[i] = pt[j];
				pt[j] = temp;
				temp = b[j];
				b[j] = b[i];
				b[i] = temp;
				temp = proc[i];
				proc[i] = proc[j];
				proc[j] = temp;
			}
		}
	}
}

int gcd(int a, int b)
{
	int r;
	while (b > 0)
	{
		r = a % b;
		a = b;
		b = r;
	}
```

```
	return a;
}

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

void main()
{
	int n;
	printf("Enter the number of processes:");
	scanf("%d", &n);
	int proc[n], b[n], pt[n], rem[n];
	printf("Enter the CPU burst times:\n");
	for (int i = 0; i < n; i++)
	{
		scanf("%d", &b[i]);
		rem[i] = b[i];
	}
	printf("Enter the time periods:\n");
	for (int i = 0; i < n; i++)
		scanf("%d", &pt[i]);
	for (int i = 0; i < n; i++)
		proc[i] = i + 1;

	sort(proc, b, pt, n);
	// LCM
	int l = lcmul(pt, n);
	printf("LCM=%d\n", l);

	printf("\nRate Monotone Scheduling:\n");
	printf("PID\t Burst\tPeriod\n");
	for (int i = 0; i < n; i++)
		printf("%d\t\t%d\t\t%d\n", proc[i], b[i], pt[i]);

	// feasibility
	double sum = 0.0;
	for (int i = 0; i < n; i++)
	{
		sum += (double)b[i] / pt[i];
	}
	double rhs = n * (pow(2.0, (1.0 / n)) - 1.0);
	printf("\n%lf <= %lf =>%s\n", sum, rhs, (sum <= rhs) ? "true" : "false");
```

```
        if (sum > rhs)
                exit(0);

        printf("Scheduling occurs for %d ms\n\n", l);

        // RMS
        int time = 0, prev = 0, x = 0;
        while (time < l)
        {
                int f = 0;
                for (int i = 0; i < n; i++)
                {
                        if (time % pt[i] == 0)
                                rem[i] = b[i];
                        if (rem[i] > 0)
                        {
                                if (prev != proc[i])
                                {
                                        printf("%dms onwards: Process %d running\n", time,
                                                  proc[i]);
                                        prev = proc[i];
                                }
                                rem[i]--;
                                f = 1;
                                break;
                                x = 0;
                        }
                }
                if (!f)
                {
                        if (x != 1)
                        {
                                printf("%dms onwards: CPU is idle\n", time);
                                x = 1;
                        }
                }
                time++;
        }
}
```

OUTPUT:

```
Enter the number of processes:3
Enter the CPU burst times:
3
2
2
Enter the time periods:
20
5
10
LCM=20

Rate Monotone Scheduling:
PID      Burst  Period
2               2               5
3               2               10
1               3               20

0.750000 <= 0.779763 =>true
Scheduling occurs for 20 ms
```

```
0ms onwards: Process 2 running
2ms onwards: Process 3 running
4ms onwards: Process 1 running
5ms onwards: Process 2 running
7ms onwards: Process 1 running
8ms onwards: CPU is idle
10ms onwards: Process 2 running
```

(b) Earliest-deadline First

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
void sort(int proc[], int d[], int b[], int pt[], int n)
{
int temp = 0;
for (int i = 0; i < n; i++)
{
for (int j = i; j < n; j++)
{
if (d[j] < d[i])
{
temp = d[j];
d[j] = d[i];
d[i] = temp;
temp = pt[i];
```

```
pt[i] = pt[j];
pt[j] = temp;
temp = b[j];
b[j] = b[i];
b[i] = temp;
temp = proc[i];
proc[i] = proc[j];
proc[j] = temp;
}
}
}
}

int gcd(int a, int b)
{
int r;
while (b > 0)
{
r = a % b;
a = b;
b = r;
}
return a;
}

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

void main()
{
int n;
```

```
printf("Enter the number of processes:");
scanf("%d", &n);
int proc[n], b[n], pt[n], d[n], rem[n];
printf("Enter the CPU burst times:\n");
for (int i = 0; i < n; i++)
{
scanf("%d", &b[i]);
rem[i] = b[i];
}
printf("Enter the deadlines:\n");
for (int i = 0; i < n; i++)
scanf("%d", &d[i]);
printf("Enter the time periods:\n");
for (int i = 0; i < n; i++)
scanf("%d", &pt[i]);
for (int i = 0; i < n; i++)
proc[i] = i + 1;

sort(proc, d, b, pt, n);
// LCM
int l = lcmul(pt, n);

printf("\nEarliest Deadline Scheduling:\n");
printf("PID\t Burst\tDeadline\tPeriod\n");
for (int i = 0; i < n; i++)
printf("%d\t\t%d\t\t%d\t\t%d\n", proc[i], b[i], d[i], pt[i]);

printf("Scheduling occurs for %d ms\n\n", l);

// EDF
int time = 0, prev = 0, x = 0;
int nextDeadlines[n];
for (int i = 0; i < n; i++)
{
nextDeadlines[i] = d[i];
rem[i] = b[i];
}
while (time < l)
```

```
{
for (int i = 0; i < n; i++)
{
if (time % pt[i] == 0 && time != 0)
{
nextDeadlines[i] = time + d[i];
rem[i] = b[i];
}
}
int minDeadline = l + 1;
int taskToExecute = -1;
for (int i = 0; i < n; i++)
{
if (rem[i] > 0 && nextDeadlines[i] < minDeadline)
{
minDeadline = nextDeadlines[i];
taskToExecute = i;
}
}
if (taskToExecute != -1)
{
printf("%dms : Task %d is running.\n", time, proc[taskToExecute]);
rem[taskToExecute]--;
}
else
{
printf("%dms: CPU is idle.\n", time);
}

time++;
}
}
```

OUTPUT:

```
Enter the number of processes:3
Enter the CPU burst times:
3
2
2
Enter the deadlines:
7
4
8
Enter the time periods:
20
5
10

Earliest Deadline Scheduling:
PID      Burst  Deadline       Period
2              2              4              5
1              3              7              20
3              2              8              10
Scheduling occurs for 20 ms
```

```
0ms : Task 2 is running.
1ms : Task 2 is running.
2ms : Task 1 is running.
3ms : Task 1 is running.
4ms : Task 1 is running.
5ms : Task 3 is running.
6ms : Task 3 is running.
7ms : Task 2 is running.
8ms : Task 2 is running.
9ms: CPU is idle.
10ms : Task 2 is running.
11ms : Task 2 is running.
12ms : Task 3 is running.
13ms : Task 3 is running.
14ms: CPU is idle.
15ms : Task 2 is running.
16ms : Task 2 is running.
17ms: CPU is idle.
18ms: CPU is idle.
19ms: CPU is idle.
```

LAB - 5

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: 2

Consumer consumes item 1
Enter your choice: 2
Buffer is empty!!
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: 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);
    }
    return 0;
}
```

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 - 6

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("\nDeadlock detected");
    else
        printf("\nNoDeadlock 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

Deadlockdetected
```

LAB-7

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>
#include <stdlib.h>
#define MAX 25

void firstFit(int nb, int nf, int b[], int f[]) {
    int ff[MAX] = {0};
    int allocated[MAX] = {0};
    for (int i = 0; i < nf; i++) {
        ff[i] = -1;
        for (int j = 0; j < nb; j++) {
            if (allocated[j] == 0 && b[j] >= f[i]) {
                ff[i] = j;
                allocated[j] = 1;
                break;
            }
        }
    }

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

```
void bestFit(int nb, int nf, int b[], int f[]) {
    int ff[MAX] = {0};
    int allocated[MAX] = {0};
    for (int i = 0; i < nf; i++) {
        int best = -1;
        ff[i] = -1;
        for (int j = 0; j < nb; j++) {
            if (allocated[j] == 0 && b[j] >= f[i]) {
                if (best == -1 || b[j] < b[best])
                    best = j;
            }
        }
        if (best != -1) {
            ff[i] = best;
            allocated[best] = 1;
        }
    }

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

void worstFit(int nb, int nf, int b[], int f[]) {
    int ff[MAX] = {0};
    int allocated[MAX] = {0};

    for (int i = 0; i < nf; i++) {
        int worst = -1;
        ff[i] = -1;
        for (int j = 0; j < nb; j++) {
            if (allocated[j] == 0 && b[j] >= f[i]) {
```

```
            if (worst == -1 || b[j] > b[worst])
                worst = j;
        }
    }
    if (worst != -1) {
        ff[i] = worst;
        allocated[worst] = 1;
    }
  }

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

int main() {
  int nb, nf, choice;

  printf("Memory Management Scheme");
  printf("\nEnter the number of blocks: ");
  scanf("%d", &nb);
  printf("Enter the number of files: ");
  scanf("%d", &nf);
  int b[nb], f[nf];
  printf("\nEnter the size of the blocks:\n");
  for (int i = 0; i < nb; i++) {
    printf("Block %d: ", i + 1);
    scanf("%d", &b[i]);
  }
  printf("Enter the size of the files:\n");
  for (int i = 0; i < nf; i++) {
    printf("File %d: ", i + 1);
```

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

    while (1) {
        printf("\n1. First Fit\n2. Best Fit\n3. Worst Fit\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("\n\tMemory Management Scheme - First Fit\n");
                firstFit(nb, nf, b, f);
                break;
            case 2:
                printf("\n\tMemory Management Scheme - Best Fit\n");
                bestFit(nb, nf, b, f);
                break;
            case 3:
                printf("\n\tMemory Management Scheme - Worst Fit\n");
                worstFit(nb, nf, b, f);
                break;
            case 4:
                printf("\nExiting...\n");
                exit(0);
                break;
            default:
                printf("\nInvalid choice.\n");
                break;
        }
    }
    return 0;
}
```

OUTPUT:

```
Memory Management Scheme
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
```

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

        Memory Management Scheme - First Fit

File_no:        File_size :     Block_no:       Block_size:
1               212             2               500
2               415             5               600
3               63              1               100
4               200             3               200
5               255             4               300
```

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

        Memory Management Scheme - Best Fit

File_no:        File_size :     Block_no:       Block_size:
1               212             4               300
2               415             2               500
3               63              1               100
4               200             3               200
5               255             5               600
```

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

        Memory Management Scheme - Worst Fit

File_no:        File_size :     Block_no:       Block_size:
1               212             5               600
2               415             2               500
3               63              4               300
4               200             3               200
5               255             -               -
```

LAB-8

Question 1:

Write a C program to simulate page replacement algorithms:

(a) FIFO

(b) LRU

(c) Optimal

CODE:

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

#define MAX_FRAMES 10
#define MAX_PAGES 100
int x=0;
void printFrames(int frames[], int framesCount, bool fault) {

    for (int i = 0; i < framesCount; i++) {
        if (frames[i] == -1)
            printf("  ");
        else{
            printf("%d ", frames[i]);
            }
    }
    if (fault) printf(" - page fault %d",++x);
    else printf(" ");
    printf("\n");
}

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

int getOptimalReplacementIndex(int pages[], int currentIndex, int frames[], int framesCount, int pagesCount) {
    int farthest = currentIndex;
    int index = -1;

    for (int i = 0; i < framesCount; i++) {
        int j;
        for (j = currentIndex; j < pagesCount; j++) {
            if (frames[i] == pages[j]) {
                if (j > farthest) {
```

```
                farthest = j;
                index = i;
            }
            break;
        }
    }
    if (j == pagesCount) {
        return i;
    }
}

return index == -1 ? 0 : index;
}

void fifo(int pages[], int pagesCount, int framesCount) {
    x=0;
    printf("FIFO Page Replacement Algorithm\n");

    int frames[MAX_FRAMES];
    int currentFrame = 0;
    int pageFaults = 0;

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

    for (int i = 0; i < pagesCount; i++) {
        bool fault = false;
        if (!isPageInFrames(pages[i], frames, framesCount)) {
            frames[currentFrame] = pages[i];
            currentFrame = (currentFrame + 1) % framesCount;
            fault = true;
            pageFaults++;
        }
        printFrames(frames, framesCount, fault);
    }

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

void optimal(int pages[], int pagesCount, int framesCount) {
    x=0;
    printf("Optimal Page Replacement Algorithm\n");

    int frames[MAX_FRAMES];
    int pageFaults = 0;

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

```
    for (int i = 0; i < pagesCount; i++) {
        bool fault = false;
        if (!isPageInFrames(pages[i], frames, framesCount)) {
            if (frames[i % framesCount] == -1) {
                frames[i % framesCount] = pages[i];
            } else {
                int index = getOptimalReplacementIndex(pages, i + 1, frames, framesCount, pagesCount);
                frames[index] = pages[i];
            }
            fault = true;
            pageFaults++;
        }
        printFrames(frames, framesCount, fault);
    }

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

void lru(int pages[], int pagesCount, int framesCount) {
    x=0;
    printf("LRU Page Replacement Algorithm\n");

    int frames[MAX_FRAMES];
    int pageFaults = 0;
    int recent[MAX_FRAMES];

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

    for (int i = 0; i < pagesCount; i++) {
        bool fault = false;
        if (!isPageInFrames(pages[i], frames, framesCount)) {
            int lruIndex = 0;
            for (int j = 1; j < framesCount; j++) {
                if (recent[j] < recent[lruIndex]) {
                    lruIndex = j;
                }
            }
            frames[lruIndex] = pages[i];
            fault = true;
            pageFaults++;
        }
        for (int j = 0; j < framesCount; j++) {
            if (frames[j] == pages[i]) {
                recent[j] = i;
            }
        }
        printFrames(frames, framesCount, fault);
    }
```

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

int main() {
    int pages[MAX_PAGES];
    int pagesCount;
    int framesCount;

    printf("Enter number of frames: ");
    scanf("%d", &framesCount);

    printf("Enter number of pages: ");
    scanf("%d", &pagesCount);

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

    fifo(pages, pagesCount, framesCount);
    optimal(pages, pagesCount, framesCount);
    lru(pages, pagesCount, framesCount);

    return 0;
}
```

OUTPUT:

```
Enter number of frames: 3
Enter number of pages: 20
Enter the page reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
```

```
FIFO Page Replacement Algorithm
7      - page fault 1
7 0    - page fault 2
7 0 1  - page fault 3
2 0 1  - page fault 4
2 0 1
2 3 1  - page fault 5
2 3 0  - page fault 6
4 3 0  - page fault 7
4 2 0  - page fault 8
4 2 3  - page fault 9
0 2 3  - page fault 10
0 2 3
0 2 3
0 1 3  - page fault 11
0 1 2  - page fault 12
0 1 2
0 1 2
7 1 2  - page fault 13
7 0 2  - page fault 14
7 0 1  - page fault 15
Total Page Faults: 15
```

```
Optimal Page Replacement Algorithm
7      - page fault 1
7 0    - page fault 2
7 0 1  - page fault 3
2 0 1  - page fault 4
2 0 1
2 0 3  - page fault 5
2 0 3
2 4 3  - page fault 6
2 4 3
2 4 3
2 0 3  - page fault 7
2 0 3
2 0 3
2 0 1  - page fault 8
2 0 1
2 0 1
2 0 1
7 0 1  - page fault 9
7 0 1
7 0 1
Total Page Faults: 9
```