VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS

(23CS4PCOPS)

Submitted by

MANYA VAID(1BM22CS150)

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

Apr-2024 to Aug-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 – 23CS4PCOPS" carried out by **MANYA VAID(1BM22CS150),** who is a 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 2024. The Lab report has been approved as it satisfies the academic requirements in respect of a **OPERATING SYSTEMS - (23CS4PCOPS)** work prescribed for the said degree.

Sowmya T	**Dr. Jyothi S Nayak**
Assistant Professor	Professor and Head
Department of CSE	Department of CSE
BMSCE, Bengaluru	BMSCE, Bengaluru

Index Sheet

Course Outcome

CO1	Apply the different concepts and functionalities of Operating System
CO2	Analyze 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

PROGRAM 1

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-preemptive)

INPUT

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

// Function to find the waiting time for all processes (Non-preemptive)
void findWaitingTimeFCFS(int processes[], int n, int bt[], int wt[], int at[], int ct[]) {
    for (int i = 0; i < n; i++) {
        wt[i] = ct[i] - at[i] - bt[i];
    }
}

// Function to find the waiting time for all processes (Preemptive)
void findWaitingTimeSJFPreemptive(int processes[], int n, int bt[], int wt[], int at[], int ct[]) {
    int rt[n];
    for (int i = 0; i < n; i++)
        rt[i] = bt[i];
    int complete = 0, t = 0, minm = INT_MAX;
    int shortest = 0, finish_time;
    while (complete != n) {
        for (int j = 0; j < n; j++) {
            if ((at[j] <= t) && (rt[j] < minm) && (rt[j] > 0)) {
                minm = rt[j];
                shortest = j;
            }
        }
        rt[shortest]--;
        minm = rt[shortest];
        if (minm == 0)
            minm = INT_MAX;
        if (rt[shortest] == 0) {
            complete++;
            finish_time = t + 1;
            wt[shortest] = finish_time - bt[shortest] - at[shortest];
            if (wt[shortest] < 0)
                wt[shortest] = 0;
            ct[shortest] = finish_time;
        }
        t++;
    }
}

// Function to find the waiting time for all processes (Non-preemptive)
void findWaitingTimeSJFNonPreemptive(int processes[], int n, int bt[], int wt[], int at[], int ct[]) {
    int rt[n];
    for (int i = 0; i < n; i++)
        rt[i] = bt[i];

    int complete = 0, t = 0, minm = INT_MAX;
    int shortest = 0, finish_time;
    while (complete != n) {
        for (int j = 0; j < n; j++) {
```

```c
            if ((at[j] <= t) && (rt[j] < minm) && (rt[j] > 0)) {
                minm = rt[j];
                shortest = j;
            }
        }
        t += rt[shortest];
        finish_time = t;
        wt[shortest] = finish_time - bt[shortest] - at[shortest];
        if (wt[shortest] < 0)
            wt[shortest] = 0;
        rt[shortest] = INT_MAX;
        complete++;
        ct[shortest] = finish_time;
        minm = INT_MAX;
    }
}

// Function to find the turnaround time for all processes
void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[], int ct[], int at[]) {
    for (int i = 0; i < n; i++)
        tat[i] = ct[i] - at[i];
}


// Function to calculate average time for FCFS
void findAverageTimeFCFS(int processes[], int n, int bt[], int at[], int ct[]) {
    int wt[n], tat[n];
    int total_wt = 0, total_tat = 0;

    findWaitingTimeFCFS(processes, n, bt, wt, at, ct);
    findTurnAroundTime(processes, n, bt, wt, tat, ct, at);

    printf("FCFS Scheduling\n");
    printf("Processes Arrival time Burst time Waiting time Turn around time Completion time\n");

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

    float avg_wt = (float)total_wt / n;
    float avg_tat = (float)total_tat / n;
    printf("Average waiting time = %f\n", avg_wt);
```

```c
    printf("Average waiting time = %f\n", avg_wt);
    printf("Average turn around time = %f\n", avg_tat);
}

// Function to calculate average time for SJF (Non-preemptive)
void findAverageTimeSJFNonPreemptive(int processes[], int n, int bt[], int at[], int ct[]) {
    int wt[n], tat[n];
    int total_wt = 0, total_tat = 0;

    findWaitingTimeSJFNonPreemptive(processes, n, bt, wt, at, ct);
    findTurnAroundTime(processes, n, bt, wt, tat, ct , at);

    printf("\nSJF (Non-preemptive) Scheduling\n");
    printf("Processes Arrival time Burst time Waiting time Turn around time Completion time\n");

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

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

// Function to calculate average time for SJF (Preemptive)
void findAverageTimeSJFPreemptive(int processes[], int n, int bt[], int at[], int ct[]) {
    int wt[n], tat[n];
    int total_wt = 0, total_tat = 0;

    findWaitingTimeSJFPreemptive(processes, n, bt, wt, at, ct);
    findTurnAroundTime(processes, n, bt, wt, tat, ct , at);

    printf("\nSJF (Preemptive) Scheduling\n");
    printf("Processes Arrival time Burst time Waiting time Turn around time Completion time\n");

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

```c
        printf("            %d", wt[i]);
        printf("            %d", tat[i]);
        printf("            %d\n", ct[i]);
    }

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

int main() {
    int processes[10], burst_time[10], arrival_time[10], completion_time[10];
    int n;

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

    printf("Enter arrival time and burst time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Arrival time of process[%d]: ", i + 1);
        scanf("%d", &arrival_time[i]);
        printf("Burst time of process[%d]: ", i + 1);
        scanf("%d", &burst_time[i]);
        processes[i] = i + 1;
    }

    completion_time[0] = arrival_time[0] + burst_time[0];
    for (int i = 1; i < n; i++) {
        if (arrival_time[i] > completion_time[i - 1]) {
            completion_time[i] = arrival_time[i] + burst_time[i];
        } else {
            completion_time[i] = completion_time[i - 1] + burst_time[i];
        }
    }

    findAverageTimeFCFS(processes, n, burst_time, arrival_time, completion_time);
    findAverageTimeSJFNonPreemptive(processes, n, burst_time, arrival_time, completion_time);
    findAverageTimeSJFPreemptive(processes, n, burst_time, arrival_time, completion_time);

    return 0;
}
```

OUTPUT

```
C:\Users\STUDENT\Desktop\1

Enter the number of processes: 4
Enter arrival time and burst time for each process:
Arrival time of process[1]: 0
Burst time of process[1]: 5
Arrival time of process[2]: 1
Burst time of process[2]: 3
Arrival time of process[3]: 2
Burst time of process[3]: 8
Arrival time of process[4]: 3
Burst time of process[4]: 6
FCFS Scheduling
Processes Arrival time Burst time Waiting time Turn around time Completion time
 1          0           5           0               5                5
 2          1           3           4               7                8
 3          2           8           6               14                16
 4          3           6           13               19                22
Average waiting time = 5.750000
Average turn around time = 11.250000

SJF (Non-preemptive) Scheduling
Processes Arrival time Burst time Waiting time Turn around time Completion time
 1          0           5           0               5                5
 2          1           3           4               7                8
 3          2           8           12               20                22
 4          3           6           5               11                14
Average waiting time = 5.250000
Average turn around time = 10.750000

SJF (Preemptive) Scheduling
Processes Arrival time Burst time Waiting time Turn around time Completion time
 1          0           5           3               8                8
 2          1           3           0               3                4
 3          2           8           12               20                22
 4          3           6           5               11                14
Average waiting time = 5.000000
Average turn around time = 10.500000

Process returned 0 (0x0)   execution time : 13.876 s
Press any key to continue.
```

PROGRAM 2

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)

INPUT

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

// Function to find the waiting time for all processes (Non-preemptive Priority)
void findWaitingTimePriorityNonPreemptive(int processes[], int n, int bt[], int wt[], int at[], int priority[], int ct[]) {
    int rt[n];
    for (int i = 0; i < n; i++)
        rt[i] = bt[i];

    int complete = 0, t = 0;
    while (complete != n) {
        int highest_priority = -1;
        int min_priority = INT_MAX;
        for (int j = 0; j < n; j++) {
            if (at[j] <= t && priority[j] < min_priority && rt[j] > 0) {
                min_priority = priority[j];
                highest_priority = j;
            }
        }
        if (highest_priority == -1) {
            t++;
            continue;
        }
        t += rt[highest_priority];
        ct[highest_priority] = t;
        rt[highest_priority] = 0;
        complete++;
        wt[highest_priority] = ct[highest_priority] - bt[highest_priority] - at[highest_priority];
```

```c
        if (wt[highest_priority] < 0)
            wt[highest_priority] = 0;
    }
}

// Function to find the waiting time for all processes (Preemptive Priority)
void findWaitingTimePriorityPreemptive(int processes[], int n, int bt[], int wt[], int at[], int priority[], int ct[]) {
    int rt[n];
    for (int i = 0; i < n; i++)
        rt[i] = bt[i];

    int complete = 0, t = 0;
    while (complete != n) {
        int highest_priority = -1;
        int min_priority = INT_MAX;
        for (int j = 0; j < n; j++) {
            if (at[j] <= t && priority[j] < min_priority && rt[j] > 0) {
                min_priority = priority[j];
                highest_priority = j;
            }
        }
        if (highest_priority == -1) {
            t++;
            continue;
        }
        rt[highest_priority]--;
        ct[highest_priority] = t + 1;
        if (rt[highest_priority] == 0) {
```

```c
            complete++;
            wt[highest_priority] = ct[highest_priority] - bt[highest_priority] - at[highest_priority];
            if (wt[highest_priority] < 0)
                wt[highest_priority] = 0;
        }
        t++;
    }
}

// Function to find the waiting time for all processes (Round Robin)
void findWaitingTimeRoundRobin(int processes[], int n, int bt[], int wt[], int quantum, int at[], int ct[]) {
    int rt[n];
    for (int i = 0; i < n; i++)
        rt[i] = bt[i];

    int t = 0;
    while (1) {
        int done = 1;
        for (int i = 0; i < n; i++) {
            if (rt[i] > 0) {
                done = 0;
                if (rt[i] > quantum) {
                    t += quantum;
                    rt[i] -= quantum;
                } else {
                    t += rt[i];
                    ct[i] = t;
                    wt[i] = ct[i] - bt[i] - at[i];
```

```c
                    if (wt[i] < 0)
                        wt[i] = 0;
                    rt[i] = 0;
                }
            }
        }
        if (done == 1)
            break;
    }
}

// Function to find the turnaround time for all processes
void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[], int ct[]) {
    for (int i = 0; i < n; i++)
        tat[i] = ct[i] - bt[i];
}

// Function to calculate average time for Priority (Non-preemptive)
void findAverageTimePriorityNonPreemptive(int processes[], int n, int bt[], int at[], int priority[], int ct[]) {
    int wt[n], tat[n];
    int total_wt = 0, total_tat = 0;

    findWaitingTimePriorityNonPreemptive(processes, n, bt, wt, at, priority, ct);
    findTurnAroundTime(processes, n, bt, wt, tat, ct);

    printf("\nPriority (Non-preemptive) Scheduling\n");
    printf("Processes Arrival time Burst time Waiting time Turn around time Completion time\n");
```

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

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

// Function to calculate average time for Priority (Preemptive)
void findAverageTimePriorityPreemptive(int processes[], int n, int bt[], int at[], int priority[], int ct[]) {
    int wt[n], tat[n];
    int total_wt = 0, total_tat = 0;

    findWaitingTimePriorityPreemptive(processes, n, bt, wt, at, priority, ct);
    findTurnAroundTime(processes, n, bt, wt, tat, ct);

    printf("\nPriority (Preemptive) Scheduling\n");
    printf("Processes Arrival time Burst time Waiting time Turn around time Completion time\n");
```

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

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

// Function to calculate average time for Round Robin
void findAverageTimeRoundRobin(int processes[], int n, int bt[], int at[], int quantum, int ct[]) {
    int wt[n], tat[n];
    int total_wt = 0, total_tat = 0;

    findWaitingTimeRoundRobin(processes, n, bt, wt, quantum, at, ct);
    findTurnAroundTime(processes, n, bt, wt, tat, ct);

    printf("\nRound Robin Scheduling (Quantum = %d)\n", quantum);
    printf("Processes Arrival time Burst time Waiting time Turn around time Completion time\n");
```

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

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

int main() {
    int processes[10], burst_time[10], arrival_time[10], priority[10], completion_time[10];
    int n, quantum;

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

    printf("Enter arrival time, burst time, and priority for Priority scheduling:\n");
    for (int i = 0; i < n; i++) {
        printf("Arrival time of process[%d]: ", i + 1);
        scanf("%d", &arrival_time[i]);
```

```c
        printf("Burst time of process[%d]: ", i + 1);
        scanf("%d", &burst_time[i]);
        printf("Priority of process[%d]: ", i + 1);
        scanf("%d", &priority[i]);
        processes[i] = i + 1;
    }

    printf("Enter the time quantum for Round Robin: ");
    scanf("%d", &quantum);
    completion_time[0] = arrival_time[0] + burst_time[0];
    for (int i = 1; i < n; i++) {
        if (arrival_time[i] > completion_time[i - 1]) {
            completion_time[i] = arrival_time[i] + burst_time[i];
        } else {
            completion_time[i] = completion_time[i - 1] + burst_time[i];
        }
    }

    findAverageTimePriorityNonPreemptive(processes, n, burst_time, arrival_time, priority, completion_time);
    findAverageTimePriorityPreemptive(processes, n, burst_time, arrival_time, priority, completion_time);
    findAverageTimeRoundRobin(processes, n, burst_time, arrival_time, quantum, completion_time);

    return 0;
}
```

OUTPUT

```
Enter the number of processes: 3
Enter arrival time, burst time, and priority for Priority scheduling:
Arrival time of process[1]: 0
Burst time of process[1]: 10
Priority of process[1]: 3
Arrival time of process[2]: 1
Burst time of process[2]: 1
Priority of process[2]: 1
Arrival time of process[3]: 2
Burst time of process[3]: 2
Priority of process[3]: 4
Enter the time quantum for Round Robin: 2

Priority (Non-preemptive) Scheduling
Processes Arrival time Burst time Waiting time Turn around time Completion time
 1          0          10          0                  0                  10
 2          1          1          9                  10                  11
 3          2          2          9                  11                  13
Average waiting time = 6.000000
Average turn around time = 7.000000

Priority (Preemptive) Scheduling
Processes Arrival time Burst time Waiting time Turn around time Completion time
 1          0          10          1                  1                  11
 2          1          1          0                  1                  2
 3          2          2          9                  11                  13
Average waiting time = 3.333333
Average turn around time = 4.333333

Round Robin Scheduling (Quantum = 2)
```

```
Average waiting time = 3.333333
Average turn around time = 4.333333

Round Robin Scheduling (Quantum = 2)
Processes Arrival time Burst time Waiting time Turn around time Completion time
 1          0          10          3                  3                  13
 2          1          1          1                  2                  3
 3          2          2          1                  3                  5
Average waiting time = 1.666667
Average turn around time = 2.666667


...Program finished with exit code 0
Press ENTER to exit console.
```

PROGRAM 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.

INPUT

```c
#include <stdio.h>
#define MAX_PROCESS 100
struct process {
  int pid;
  int arrival_time;
  int burst_time;
  int level;
};
void find_turnaround_time(struct process proc[], int n, int wt[], int tat[]) {
  int i;
  tat[0] = proc[0].burst_time;
  wt[0] = 0;
  for (i = 1; i < n; i++) {
    tat[i] = proc[i].burst_time + wt[i - 1];
    wt[i] = tat[i] - proc[i].burst_time;
  }
}

void find_avg_time(struct process proc[], int n) {
  int wt[n], tat[n], i;
  double total_wt = 0, total_tat = 0;
  find_turnaround_time(proc, n, wt, tat);
  printf("Process | Arrival Time | Burst Time | Level | Waiting Time | Turnaround Time\n");
  for (i = 0; i < n; i++) {
    total_wt += wt[i];
    total_tat += tat[i];
    printf(" %d \t | %d \t\t | %d \t\t | %d \t\t | %d \t\t | %d \n",
           proc[i].pid, proc[i].arrival_time, proc[i].burst_time, proc[i].level, wt[i], tat[i]);
  }
  printf("Average Waiting Time = %.2lf\n", total_wt / n);
  printf("Average Turnaround Time = %.2lf\n", total_tat / n);
}

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

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

  printf("Enter details of processes:\n");
  for (i = 0; i < n; i++) {
    printf("Process ID: ");
    scanf("%d", &proc[i].pid);
    printf("Arrival Time: ");
    scanf("%d", &proc[i].arrival_time);
    printf("Burst Time: ");
    scanf("%d", &proc[i].burst_time);

    printf("Process Level (1 - System, 2 - User): ");
```

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

        printf("Process Level (1 - System, 2 - User): ");
        scanf("%d", &proc[i].level);
    }

    for (i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (proc[j].arrival_time > proc[j + 1].arrival_time) {
                struct process temp = proc[j];
                proc[j] = proc[j + 1];
                proc[j + 1] = temp;
            }
        }
    }

    printf("\nMulti-level Queue Scheduling (FCFS)\n");
    find_avg_time(proc, n);

    return 0;
}
```

OUTPUT

```
C:\Users\STUDENT\Desktop\r

Enter the number of processes: 3
Enter details of processes:
Process ID: 1
Arrival Time: 0
Burst Time: 5
Process Level (1 - System, 2 - User): 1
Process ID: 2
Arrival Time: 2
Burst Time: 7
Process Level (1 - System, 2 - User): 2
Process ID: 3
Arrival Time: 1
Burst Time: 6
Process Level (1 - System, 2 - User): 1

Multi-level Queue Scheduling (FCFS)
Process | Arrival Time | Burst Time | Level | Waiting Time | Turnaround Time
 1      | 0            | 5          | 1     | 0            | 5
 3      | 1            | 6          | 1     | 0            | 6
 2      | 2            | 7          | 2     | 0            | 7
Average Waiting Time = 0.00
Average Turnaround Time = 6.00

Process returned 0 (0x0)   execution time : 24.017 s
Press any key to continue.
```

PROGRAM 4

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

a) Rate- Monotonic

b) Earliest-deadline First

c) Proportional scheduling

INPUT

```
#include<stdio.h>
typedef struct Process {
    int id;
    int arrival_time;
    int burst_time;
    int deadline;
    float priority;
} Process;

void rate_monotonic(Process processes[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].deadline > processes[j + 1].deadline) {
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }

    printf("Rate-Monotonic scheduling:\n");
    int current_time = 0;
    for (int i = 0; i < n; i++) {
        printf("Process %d executes from time %d to %d\n", processes[i].id, current_time, current_time + processes[i].burst_time);
        current_time += processes[i].burst_time;
    }
}

void earliest_deadline_first(Process processes[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].deadline > processes[j + 1].deadline) {
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }

    printf("\nEarliest-Deadline First scheduling:\n");
    int current_time = 0;
    for (int i = 0; i < n; i++) {
        printf("Process %d executes from time %d to %d\n", processes[i].id, current_time, current_time + processes[i].burst_time);
        current_time += processes[i].burst_time;
    }
}

void proportional_scheduling(Process processes[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
```

```c
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].priority < processes[j + 1].priority) {
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }

    printf("\nProportional scheduling:\n");
    int current_time = 0;
    for (int i = 0; i < n; i++) {
        printf("Process %d executes from time %d to %d\n", processes[i].id, current_time, current_time + processes[i].burst_time);
        current_time += processes[i].burst_time;
    }
}

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

    Process processes[n];
    for (int i = 0; i < n; i++) {
        printf("Enter details for Process %d:\n", i + 1);
        processes[i].id = i + 1;
        printf("Arrival time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Burst time: ");
        scanf("%d", &processes[i].burst_time);
        printf("Deadline: ");
        scanf("%d", &processes[i].deadline);
        printf("Priority: ");
        scanf("%f", &processes[i].priority);
    }

    rate_monotonic(processes, n);
    earliest_deadline_first(processes, n);
    proportional_scheduling(processes, n);

    return 0;
}
```

OUTPUT

```
Deadline: 2
Priority: 1
Enter details for Process 2:
Arrival time: 1
Burst time: 5
Deadline: 2
Priority: 1
Enter details for Process 3:
Arrival time: 4
Burst time: 2
Deadline: 5
Priority: 3
Rate-Monotonic scheduling:
Process 1 executes from time 0 to 3
Process 2 executes from time 3 to 8
Process 3 executes from time 8 to 10

Earliest-Deadline First scheduling:
Process 1 executes from time 0 to 3
Process 2 executes from time 3 to 8
Process 3 executes from time 8 to 10

Proportional scheduling:
Process 3 executes from time 0 to 2
Process 1 executes from time 2 to 5
Process 2 executes from time 5 to 10

Process returned 0 (0x0)   execution time : 31.694 s
Press any key to continue.
```

PROGRAM 5

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

INPUT

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

#define BUFFER_SIZE 10
#define NUM_ITEMS 10 // Number of items to be produced and consumed

// Shared buffer
int buffer[BUFFER_SIZE];
int count = 0;

// Semaphore variables
int empty = BUFFER_SIZE;
int full = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

// Counter for number of items produced/consumed
int produced_count = 0;
int consumed_count = 0;

// Semaphore wait (P) operation
void wait(int* sem) {
    pthread_mutex_lock(&mutex);
    while (*sem <= 0) {
        pthread_mutex_unlock(&mutex);
        sched_yield();
        pthread_mutex_lock(&mutex);
    }
    (*sem)--;
    pthread_mutex_unlock(&mutex);
}

// Semaphore signal (V) operation
void signal(int* sem) {
    pthread_mutex_lock(&mutex);
    (*sem)++;
    pthread_mutex_unlock(&mutex);
}

// Producer function
void* producer(void* arg) {
    int item;
    while (1) {
        if (produced_count >= NUM_ITEMS) {
            break;
        }
        item = produced_count;
        wait(&empty);
        pthread_mutex_lock(&mutex);
```

```c
        buffer[count] = item;
        count++;
        produced_count++;
        printf("Producer produced: %d\n", item);
        pthread_mutex_unlock(&mutex);
        signal(&full);
        sleep(rand() % 4);
    }
    return NULL;
}

// Consumer function
void* consumer(void* arg) {
    int item;
    while (1) {
        if (consumed_count >= NUM_ITEMS) {
            break;
        }
        wait(&full);
        pthread_mutex_lock(&mutex);
        count--;
        item = buffer[count];
        consumed_count++;
        printf("Consumer consumed: %d\n", item);
        pthread_mutex_unlock(&mutex);
        signal(&empty);
        sleep(rand() % 4);
    }
    return NULL;
}

int main() {
    pthread_t prod_thread, cons_thread;

    // Create the producer and consumer threads
    pthread_create(&prod_thread, NULL, producer, NULL);
    pthread_create(&cons_thread, NULL, consumer, NULL);

    // Wait for the threads to finish
    pthread_join(prod_thread, NULL);
    pthread_join(cons_thread, NULL);

    return 0;
}
```

OUTPUT

```
C:\Users\STUDENT\Desktop\L

Producer produced: 0
Consumer consumed: 0
Producer produced: 1
Consumer consumed: 1
Producer produced: 2
Consumer consumed: 2
Producer produced: 3
Producer produced: 4
Consumer consumed: 4
Consumer consumed: 3
Producer produced: 5
Producer produced: 6
Consumer consumed: 6
Consumer consumed: 5
Producer produced: 7
Consumer consumed: 7
Producer produced: 8
Consumer consumed: 8
Producer produced: 9
Consumer consumed: 9

Process returned 0 (0x0)   execution time : 13.107 s
Press any key to continue.
```

PROGRAM 6

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

INPUT

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

#define NUM_PHILOSOPHERS 5
#define ITERATIONS 5

pthread_mutex_t forks[NUM_PHILOSOPHERS];
pthread_t philosophers[NUM_PHILOSOPHERS];

int hungry_philosophers[NUM_PHILOSOPHERS];
int hungry_count;

typedef struct {
    pthread_mutex_t lock;
    pthread_cond_t cond;
    int count;
} semaphore_t;

void semaphore_init(semaphore_t* sem, int value) {
    pthread_mutex_init(&sem->lock, NULL);
    pthread_cond_init(&sem->cond, NULL);
    sem->count = value;
}

void semaphore_wait(semaphore_t* sem) {
    pthread_mutex_lock(&sem->lock);
    while (sem->count == 0) {
        pthread_cond_wait(&sem->cond, &sem->lock);
    }
    sem->count--;
    pthread_mutex_unlock(&sem->lock);
}

void semaphore_signal(semaphore_t* sem) {
    pthread_mutex_lock(&sem->lock);
    sem->count++;
    pthread_cond_signal(&sem->cond);
    pthread_mutex_unlock(&sem->lock);
}

void think(int philosopher_number) {
    printf("Philosopher %d is thinking.\n", philosopher_number);
    usleep(rand() % 1000 + 500);
```

```c
    usleep(rand() % 1000 + 500);
}

void eat(int philosopher_number) {
    printf("Philosopher %d is eating.\n", philosopher_number);
    usleep(rand() % 1000 + 500);
}

void pick_up_forks(int philosopher_number) {
    int left_fork = philosopher_number;
    int right_fork = (philosopher_number + 1) % NUM_PHILOSOPHERS;

    if (philosopher_number % 2 == 0) {
        pthread_mutex_lock(&forks[left_fork]);
        pthread_mutex_lock(&forks[right_fork]);
    } else {
        pthread_mutex_lock(&forks[right_fork]);
        pthread_mutex_lock(&forks[left_fork]);
    }

    printf("Philosopher %d picked up fork %d and fork %d.\n", philosopher_number, left_fork, right_fork);
}

void put_down_forks(int philosopher_number) {
    int left_fork = philosopher_number;
    int right_fork = (philosopher_number + 1) % NUM_PHILOSOPHERS;

    pthread_mutex_unlock(&forks[left_fork]);
    pthread_mutex_unlock(&forks[right_fork]);

    printf("Philosopher %d put down fork %d and fork %d.\n", philosopher_number, left_fork, right_fork);
}

void* philosopher(void* num) {
    int philosopher_number = *(int*)num;
    free(num);

    for (int i = 0; i < ITERATIONS; i++) {
        think(philosopher_number);
        pick_up_forks(philosopher_number);
        eat(philosopher_number);
        put_down_forks(philosopher_number);
    }
```

```c
    return NULL;
}

void allow_one_philosopher_to_eat() {
    for (int i = 0; i < hungry_count; i++) {
        int philosopher_number = hungry_philosophers[i];
        think(philosopher_number);
        pick_up_forks(philosopher_number);
        eat(philosopher_number);
        put_down_forks(philosopher_number);
    }
}

void allow_two_philosophers_to_eat() {
    int combination[3][2] = {
        {0, 1}, {0, 2}, {1, 2}
    };

    for (int i = 0; i < 3; i++) {
        printf("combination %d\n", i + 1);
        int p1 = hungry_philosophers[combination[i][0]];
        int p2 = hungry_philosophers[combination[i][1]];
        think(p1);
        think(p2);
        pick_up_forks(p1);
        pick_up_forks(p2);
        eat(p1);
        eat(p2);
        put_down_forks(p1);
        put_down_forks(p2);
    }
}

int main() {
    srand(time(NULL));
    int choice;

    printf("DINING PHILOSOPHER PROBLEM\n");
    printf("Enter the total no. of philosophers: %d\n", NUM_PHILOSOPHERS);
    printf("How many are hungry: ");
    scanf("%d", &hungry_count);

    printf("Enter the positions of the hungry philosophers:\n");
    for (int i = 0; i < hungry_count; i++) {
```

```c
    printf("Enter the positions of the hungry philosophers:\n");
    for (int i = 0; i < hungry_count; i++) {
        int pos;
        scanf("%d", &pos);
        hungry_philosophers[i] = pos;
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_mutex_init(&forks[i], NULL);
    }

    while (1) {
        printf("\n1.One can eat at a time  2.Two can eat at a time  3.Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        if (choice == 1) {
            printf("Allow one philosopher to eat at any time\n");
            allow_one_philosopher_to_eat();
        } else if (choice == 2) {
            printf("Allow two philosophers to eat at same time\n");
            allow_two_philosophers_to_eat();
        } else if (choice == 3) {
            break;
        } else {
            printf("Invalid choice. Please try again.\n");
        }
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_mutex_destroy(&forks[i]);
    }

    return 0;
}
```

OUTPUT

```
DINING PHILOSOPHER PROBLEM
Enter the total no. of philosophers: 5
How many are hungry: 3
Enter the positions of the hungry philosophers:
2
4
5

1.One can eat at a time  2.Two can eat at a time  3.Exit
Enter your choice: 1
Allow one philosopher to eat at any time
Philosopher 2 is thinking.
Philosopher 2 picked up fork 2 and fork 3.
Philosopher 2 is eating.
Philosopher 2 put down fork 2 and fork 3.
Philosopher 4 is thinking.
Philosopher 4 picked up fork 4 and fork 0.
Philosopher 4 is eating.
Philosopher 4 put down fork 4 and fork 0.
Philosopher 5 is thinking.
Philosopher 5 picked up fork 5 and fork 1.
Philosopher 5 is eating.
Philosopher 5 put down fork 5 and fork 1.

1.One can eat at a time  2.Two can eat at a time  3.Exit
Enter your choice: Killed


...Program finished with exit code 9
Press ENTER to exit console.
```

PROGRAM 7

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

INPUT

```c
#include <stdio.h>
#include <stdbool.h>
#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

// Function to check if the system is in a safe state
bool isSafeState(int processes, int resources, int available[], int max[][MAX_RESOURCES],
int allocation[][MAX_RESOURCES]){
    int work[MAX_RESOURCES];
    bool finish[MAX_PROCESSES] = {0};
    int safeSequence[MAX_PROCESSES];
    int need[MAX_PROCESSES][MAX_RESOURCES];
    // Calculate the need matrix
    for (int i = 0; i < processes; i++) {
        for (int j = 0; j < resources; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }

    for (int i = 0; i < resources; i++) {
        work[i] = available[i];
    }

    int count = 0;

    while (count < processes) {
        bool found = false;
        for (int p = 0; p < processes; p++) {
            if (!finish[p]) {
                bool canProceed = true;
                for (int r = 0; r < resources; r++) {
                    if (need[p][r] > work[r]) {
                        canProceed = false;
                        break;
                    }
                }
                if (canProceed) {
                    printf("P%d is visited ( ", p);
                    for (int r = 0; r < resources; r++) {
                        printf("%d ", work[r]);
                    }
                    printf(")\n");

                    for (int r = 0; r < resources; r++) {
```

```c
                        work[r] += allocation[p][r];
                    }
                    safeSequence[count++] = p;
                    finish[p] = true;
                    found = true;
                }
            }
        }
        if (!found) {
            printf("System is not in a safe state.\n");
            return false;
        }
    }

    printf("SYSTEM IS IN SAFE STATE\nThe Safe Sequence is -- (");
    for (int i = 0; i < processes; i++) {
        printf("P%d ", safeSequence[i]);
    }
    printf(")\n");

    printf("\nProcess\tAllocation\tMax\t\tNeed\n");
    for (int i = 0; i < processes; i++) {
        printf("P%d\t", i);
        for (int j = 0; j < resources; j++) {
            printf("%d ", allocation[i][j]);
        }
        printf("\t\t");
        for (int j = 0; j < resources; j++) {
            printf("%d ", max[i][j]);
        }
        printf("\t\t");
        for (int j = 0; j < resources; j++) {
            printf("%d ", need[i][j]);
        }
        printf("\n");
    }

    return true;
}

int main() {
    int processes, resources;
    int available[MAX_RESOURCES];
    int max[MAX_PROCESSES][MAX_RESOURCES];
```

```c
    int allocation[MAX_PROCESSES][MAX_RESOURCES];

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

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

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

    for (int i = 0; i < processes; i++) {
        printf("Enter details for P%d\n", i);
        printf("Enter allocation -- ");
        for (int j = 0; j < resources; j++) {
            scanf("%d", &allocation[i][j]);
        }
        printf("Enter Max -- ");
        for (int j = 0; j < resources; j++) {
            scanf("%d", &max[i][j]);
        }
    }

    isSafeState(processes, resources, available, max, allocation);

    return 0;
}
```

OUTPUT

```
Enter number of processes: 5
Enter number of resources: 3
Enter Available Resources --
3 3 2
Enter details for P0
Enter allocation -- 0 1 0
Enter Max -- 7 5 3
Enter details for P1
Enter allocation -- 2 0 0
Enter Max -- 3 2 2
Enter details for P2
Enter allocation -- 3 0 2
Enter Max -- 9 0 2
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
P1 is visited ( 3 3 2 )
P3 is visited ( 5 3 2 )
P4 is visited ( 7 4 3 )
P0 is visited ( 7 4 5 )
P2 is visited ( 7 5 5 )
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       2 0 0           3 2 2            1 2 2
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


...Program finished with exit code 0
Press ENTER to exit console.
```

PROGRAM 8

Write a program to simulate deadlock detection.

INPUT

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

int num_processes, num_resources;
int **allocation, **request;
int *available;

bool canAllocate(int *request, int *work, int R) {
    for (int i = 0; i < R; i++) {
        if (request[i] > work[i]) {
            return false;
        }
    }
    return true;
}

void allocateResources(int *work, int *allocation, int R) {
    for (int i = 0; i < R; i++) {
        work[i] += allocation[i];
    }
}

void displayFinishState(bool *finish, int P) {
    printf("Finish state: ");
    for (int i = 0; i < P; i++) {
        printf("%s ", finish[i] ? "true" : "false");
    }
    printf("\n");
}

void detectDeadlock() {
    int *work = (int *)malloc(num_resources * sizeof(int));
    bool *finish = (bool *)malloc(num_processes * sizeof(bool));
    int *sequence = (int *)malloc(num_processes * sizeof(int));
    int index = 0;
    for (int i = 0; i < num_resources; i++) {
        work[i] = available[i];
    }

    for (int i = 0; i < num_processes; i++) {
        bool allocated = false;
        for (int j = 0; j < num_resources; j++) {
            if (allocation[i][j] > 0) {
                allocated = true;
                break;
            }
        }
        finish[i] = !allocated;
    }
    while (true) {
        bool found = false;
        for (int i = 0; i < num_processes; i++) {
```

```c
    bool found = false;
    for (int i = 0; i < num_processes; i++) {
        if (!finish[i] && canAllocate(request[i], work, num_resources)) {
            allocateResources(work, allocation[i], num_resources);
            finish[i] = true;
            sequence[index++] = i;
            found = true;
            break;
        }
    }
    if (!found) {
        break;
    }
}
bool deadlock = false;
for (int i = 0; i < num_processes; i++) {
    if (!finish[i]) {
        printf("Deadlock detected: Process P%d is deadlocked.\n", i);
        deadlock = true;
    }
}
if (!deadlock) {
    printf("No deadlock detected.\nSafe execution sequence: ");
    for (int i = 0; i < num_processes; i++) {
        printf("P%d ", sequence[i]);
    }
    printf("\n");
}

free(work);
free(finish);
free(sequence);
}

void input() {
    printf("Enter number of processes: ");
    scanf("%d", &num_processes);
    printf("Enter number of resources: ");
    scanf("%d", &num_resources);
    allocation = (int **)malloc(num_processes * sizeof(int *));
    request = (int **)malloc(num_processes * sizeof(int *));
    for (int i = 0; i < num_processes; ++i) {
        allocation[i] = (int *)malloc(num_resources * sizeof(int));
        request[i] = (int *)malloc(num_resources * sizeof(int));
    }
    available = (int *)malloc(num_resources * sizeof(int));
    // Input allocation matrix
    printf("Enter allocation matrix:\n");
    for (int i = 0; i < num_processes; ++i) {
        for (int j = 0; j < num_resources; ++j) {
            scanf("%d", &allocation[i][j]);
        }
    }


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

int main() {
    input();
    detectDeadlock();
    for (int i = 0; i < num_processes; i++) {
        free(allocation[i]);
        free(request[i]);
    }
    free(allocation);
    free(request);
    free(available);

    return 0;
}
```

OUTPUT

```
Enter number of processes: 5
Enter number of resources: 3
Enter allocation matrix:
0 1 0
2 0 0
3 0 3
2 1 1
0 0 2
Enter available resources:
0 0 0
Enter request matrix:
0 0 0
2 0 2
0 0 0
1 0 0
0 0 2
No deadlock detected.
Safe execution sequence: P0 P2 P1 P3 P4

Process returned 0 (0x0)   execution time : 34.563 s
Press any key to continue.
```

PROGRAM 9

Write a C program to simulate the following contiguous memory allocation techniques
a) Worst-fit
b) Best-fit
c) First-fit

INPUT

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100

void printAllocation(const char *scheme, int allocation[], int processSize[], int blockSize[], int originalBlockSize[], int n) {
    printf("Memory Management Scheme - %s\n", scheme);
    printf("File_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment\n");
    for(int i = 0; i < n; i++) {
        if(allocation[i] != -1) {
            printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i+1, processSize[i], allocation[i]+1, originalBlockSize[allocation[i]], blockSize[allocation[i]]);
        } else {
            printf("%d\t\t%d\t\tNot Allocated\n", i+1, processSize[i]);
        }
    }
    printf("\n");
}

void allocateMemory(int blockSize[], int m, int processSize[], int n, const char *scheme) {
    int allocation[n], originalBlockSize[m];
    for(int i = 0; i < m; i++) {
        originalBlockSize[i] = blockSize[i];
    }

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

    for(int i = 0; i < n; i++) {
        int idx = -1;
        for(int j = 0; j < m; j++) {
            if(blockSize[j] >= processSize[i]) {
                if(scheme == "First Fit" || (scheme == "Best Fit" && (idx == -1 || blockSize[j] < blockSize[idx])) ||
                   (scheme == "Worst Fit" && (idx == -1 || blockSize[j] > blockSize[idx]))) {
                    idx = j;
                    if(scheme == "First Fit") break;
                }
            }
        }

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

    printAllocation(scheme, allocation, processSize, blockSize, originalBlockSize, n);
}

int main() {
    int blockSize[MAX], processSize[MAX];
    int m, n;

    printf("Enter the number of blocks: ");
```

```c
    scanf("%d", &m);
    printf("Enter the size of each block: \n");
    for(int i = 0; i < m; i++) {
        printf("Block %d: ", i+1);
        scanf("%d", &blockSize[i]);
    }

    printf("Enter the number of processes: ");
    scanf("%d", &n);
    printf("Enter the size of each process: \n");
    for(int i = 0; i < n; i++) {
        printf("Process %d: ", i+1);
        scanf("%d", &processSize[i]);
    }

    int blockSize1[MAX], blockSize2[MAX], blockSize3[MAX];
    for(int i = 0; i < m; i++) {
        blockSize1[i] = blockSize2[i] = blockSize3[i] = blockSize[i];
    }

    allocateMemory(blockSize1, m, processSize, n, "First Fit");
    allocateMemory(blockSize2, m, processSize, n, "Best Fit");
    allocateMemory(blockSize3, m, processSize, n, "Worst Fit");

    return 0;
}
```

OUTPUT

```
Enter the number of blocks: 5
Enter the size of each block:
Block 1: 400
Block 2: 700
Block 3: 200
Block 4: 300
Block 5: 600
Enter the number of processes: 4
Enter the size of each process:
Process 1: 212
Process 2: 517
Process 3: 312
Process 4: 526
Memory Management Scheme - First Fit
File_no:        File_size:      Block_no:       Block_size:     Fragment
1               212             1               400             188
2               517             2               700             183
3               312             5               600             288
4               526             Not Allocated

Memory Management Scheme - Best Fit
File_no:        File_size:      Block_no:       Block_size:     Fragment
1               212             4               300             88
2               517             5               600             83
3               312             1               400             88
4               526             2               700             174

Memory Management Scheme - Worst Fit
File_no:        File_size:      Block_no:       Block_size:     Fragment
1               212             2               700             176
2               517             5               600             83
3               312             2               700             176
4               526             Not Allocated


Process returned 0 (0x0)   execution time : 34.341 s
Press any key to continue.
```

LAB PROGRAM 10

Write a C program to simulate page replacement algorithms a) FIFO b) LRU c) Optimal

INPUT-

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_FRAMES 10
#define MAX_PAGES 100

void printFrames(int frames[], int size) {
    for (int i = 0; i < size; i++) {
        if (frames[i] == -1)
            printf("- ");
        else
            printf("%d ", frames[i]);
    }
    printf("\n");
}

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

void fifo(int pages[], int pageCount, int frameCount) {
    int frames[frameCount];
    int pageFaults = 0, index = 0;
    for (int i = 0; i < frameCount; i++) {
        frames[i] = -1; // Initialize frames
    }
    printf("The Page Replacement Process is \n");
    for (int i = 0; i < pageCount; i++) {
        if (!isPageInFrames(frames, frameCount, pages[i])) {
            frames[index] = pages[i];
            index = (index + 1) % frameCount;
            pageFaults++;
            printFrames(frames, frameCount);
            printf("PF No. %d\n", pageFaults);
        } else {
            printFrames(frames, frameCount);
        }
    }
    printf("The number of Page Faults using FIFO are %d\n", pageFaults);
}

void lru(int pages[], int pageCount, int frameCount) {
    int frames[frameCount];
    int time[frameCount];
    int pageFaults = 0;
    for (int i = 0; i < frameCount; i++) {
```

```c
        frames[i] = -1;
        time[i] = -1;
    }
    printf("The Page Replacement Process is \n");
    for (int i = 0; i < pageCount; i++) {
        int page = pages[i];
        int pageFound = 0;
        for (int k = 0; k < frameCount; k++) {
            if (frames[k] == page) {
                pageFound = 1;
                time[k] = i;
                break;
            }
        }
        if (!pageFound) {
            int lruIndex = 0;
            for (int j = 1; j < frameCount; j++) {
                if (time[j] < time[lruIndex]) {
                    lruIndex = j;
                }
            }
            frames[lruIndex] = page;
            time[lruIndex] = i;
            pageFaults++;
            printFrames(frames, frameCount);
            printf("PF No. %d\n", pageFaults);
        } else {
            printFrames(frames, frameCount);
        }
    }
    printf("The number of Page Faults using LRU are %d\n", pageFaults);
}

void optimal(int pages[], int pageCount, int frameCount) {
    int frames[frameCount];
    int pageFaults = 0;
    for (int i = 0; i < frameCount; i++) {
        frames[i] = -1;
    }
    printf("The Page Replacement Process is \n");
    for (int i = 0; i < pageCount; i++) {
        int page = pages[i];
        if (!isPageInFrames(frames, frameCount, page)) {
            int farthest = i;
            int replaceIndex = 0;
            for (int j = 0; j < frameCount; j++) {
                int k;
                for (k = i + 1; k < pageCount; k++) {
```

```c
                    if (frames[j] == pages[k]) {
                        break;
                    }
                }
                if (k == pageCount) {
                    replaceIndex = j;
                    break;
                } else if (k > farthest) {
                    farthest = k;
                    replaceIndex = j;
                }
            }
            frames[replaceIndex] = page;
            pageFaults++;
            printFrames(frames, frameCount);
            printf("PF No. %d\n", pageFaults);
        } else {
            printFrames(frames, frameCount);
        }
    }
    printf("The number of Page Faults using Optimal are %d\n", pageFaults);
}

int main() {
    int pages[MAX_PAGES];
    int pageCount, frameCount;
    printf("Enter number of pages: ");
    scanf("%d", &pageCount);
    printf("Enter the page reference string:\n");
    for (int i = 0; i < pageCount; i++) {
        scanf("%d", &pages[i]);
    }
    printf("Enter number of frames: ");
    scanf("%d", &frameCount);
    printf("\nFIFO Page Replacement Algorithm:\n");
    fifo(pages, pageCount, frameCount);
    printf("\nLRU Page Replacement Algorithm:\n");
    lru(pages, pageCount, frameCount);
    printf("\nOptimal Page Replacement Algorithm:\n");
    optimal(pages, pageCount, frameCount);
    return 0;
}
```

OUTPUT-

```
Enter number of pages: 20
Enter the page reference string:
0 9 0 1 8 1 8 7 8 7 1 2 8 2 7 8 2 3 8 3
Enter number of frames: 3

FIFO Page Replacement Algorithm:
The Page Replacement Process is
0
```

```
7 8 1
PF No. 5
7 8 1
7 8 1
7 8 1
7 2 1
PF No. 6
8 2 1
PF No. 7
8 2 1
8 2 7
PF No. 8
8 2 7
8 2 7
8 2 3
PF No. 9
8 2 3
8 2 3
The number of Page Faults using LRU are 9

Optimal Page Replacement Algorithm:
The Page Replacement Process is
0 - -
PF No. 1
0 9 -
PF No. 2
0 9 -
1 9 -
PF No. 3
1 8 -
PF No. 4
1 8 -
1 8 -
1 8 7
PF No. 5
1 8 7
1 8 7
1 8 7
2 8 7
PF No. 6
2 8 7
2 8 7
2 8 7
2 8 7
2 8 7
3 8 7
PF No. 7
3 8 7
3 8 7
The number of Page Faults using Optimal are 7
```