VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

OPERATING SYSTEMS

Submitted by

Shrinanda Shivprasad Dinde (1BM23CS324)

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

Feb-2025 to June-2025

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 Shrinanda Shivprasad Dinde(1BM23CS324), who is Bonafide student of B. M. S. College of Engineering. It is in partial fulfilment for the award of Bachelor of Engineering in Computer Science and Engineering of the Visvesvaraya Technological University, Belgaum during the year Feb 2025- June 2025. 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.

Spoorthi D M	Dr. Kavitha Sooda
Assistant Professor	Professor and Head
Department of CSE	Department of CSE
BMSCE, Bengaluru	BMSCE, Bengaluru

Index Sheet

10.	Write a C program to simulate page replacement algorithms a) FIFO LRU Optimal	41-45

Course Outcomes

C01	Apply the different concepts and functionalities of Operating System
C02	Analyse various Operating system strategies and techniques
C03	Demonstrate the different functionalities of Operating System.
C04	Conduct practical experiments to implement the functionalities of Operating system.

Program -1

Question:

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)

=>FCFS:

```
#include <stdio.h>

typedef struct {
    int id, at, bt, wt, tat, ct,rem,rt, started;
} Process;

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

void fcfs(Process p[], int n) {
    sortByArrival(p, n);
    int time = 0;

    for (int i = 0; i < n; i++) {
        if (time < p[i].at)
            time = p[i].at;

        p[i].ct = time + p[i].bt;
        p[i].tat = p[i].ct - p[i].at;
        p[i].wt = p[i].tat - p[i].bt;
        time = p[i].ct;
    }
}
```

```
void display(Process p[], int n) {
    printf("\nPID\tAT\tBT\tCT\tTAT\tWT\n");

    float totalWT = 0, totalTAT = 0;
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t%d\t%d\n", p[i].id, p[i].at, p[i].bt, p[i].ct, p[i].tat, p[i].wt);
        totalWT += p[i].wt;
        totalTAT += p[i].tat;
    }

    printf("\nAverage Waiting Time: %.2f\n", totalWT / n);
    printf("Average Turnaround Time: %.2f\n", totalTAT / n);
}

void main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    Process p[n];
    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        p[i].id = i + 1;
        printf("P[%d]: ", i + 1);
        scanf("%d %d", &p[i].at, &p[i].bt);
    }
    printf("First Come First Serve (FCFS)\n");
    fcfs(p, n);
    display(p, n);
}
```

Result:

```
FCFS
Enter number of processes: 4
Enter Arrival Time and Burst Time:
0 7
0 3
0 4
0 6

--- First Come First Serve (FCFS) ---

PID     AT      BT      CT      TAT     WT
P1      0       7       7       7       0
P2      0       3       10      10      7
P3      0       4       14      14      10
P4      0       6       20      20      14

Average Waiting Time: 7.75
Average Turnaround Time: 12.75

Process returned 31 (0x1F)   execution time : 14.220 s
Press any key to continue.
```

=>SJF(Non-preemptive):

```
#include <stdio.h>
#include <limits.h>
typedef struct {
    int id, arrival, burst, completion, turnaround, waiting;
} Process;

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

void sjf_non_preemptive(Process p[], int n) {
    int completed = 0, time = 0, minIdx;
    int isCompleted[n];
    for (int i = 0; i < n; i++) isCompleted[i] = 0;
    while (completed < n) {
        minIdx = -1;
        int minBurst = INT_MAX;
        for (int i = 0; i < n; i++) {
            if (!isCompleted[i] && p[i].arrival <= time && p[i].burst < minBurst) {
                minBurst = p[i].burst;
                minIdx = i;
            }
```

```
        }
        if (minIdx == -1) { time++; continue; }
        p[minIdx].completion = time + p[minIdx].burst;
        p[minIdx].turnaround = p[minIdx].completion - p[minIdx].arrival;
        p[minIdx].waiting = p[minIdx].turnaround - p[minIdx].burst;
        time = p[minIdx].completion;
        isCompleted[minIdx] = 1;
        completed++;
    }
}
void display(Process p[], int n) {
    printf("\nPID Arrival Burst Completion Turnaround Waiting\n");
    float totalWT = 0, totalTAT = 0;
    for (int i = 0; i < n; i++) {
        printf("%3d %7d %6d %10d %10d %8d\n", p[i].id, p[i].arrival, p[i].burst, p[i].completion,
p[i].turnaround, p[i].waiting);
        totalWT += p[i].waiting;
        totalTAT += p[i].turnaround;
    }
    printf("\nAverage Waiting Time: %.2f\n", totalWT / n);
    printf("Average Turnaround Time: %.2f\n", totalTAT / n);
}
int main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    Process p[n];
    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        p[i].id = i + 1;
        printf("P[%d]: ", i + 1);
        scanf("%d %d", &p[i].arrival, &p[i].burst);
```

```
    }
    printf("\nShortest Job First (Non-Preemptive) Scheduling\n");
    sjf_non_preemptive(p, n);
    display(p, n);
    return 0;
}
```

Result:

```
C:\Users\Admin\Desktop\sjfn
Enter number of processes: 4
Enter Arrival Time and Burst Time:
0 7
8 3
3 4
5 6

SJF
PID     AT      BT      CT      TAT     WT      RT
P1      0       7       7       7       0       0
P2      8       3       14      6       3       3
P3      3       4       11      8       4       4
P4      5       6       20      15      9       9

Average Waiting Time: 4.00
Average Turnaround Time: 9.00

Process returned 30 (0x1E)   execution time : 23.064 s
Press any key to continue.
```

=>SJF(preemptive):

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

typedef struct {
    int id, arrival, burst, remaining, waiting, turnaround, completion,response,started;
} Process;

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

```
            if (p[j].arrival > p[j + 1].arrival) {
                Process temp = p[j];
                p[j]=p[j+1];
                p[j+1]=temp;
            }
        }
    }
}

void sjfPreemptive(Process p[], int n) {
    int completed = 0, time = 0, minIndex, minBurst;

    while (completed < n) {
        minIndex = -1, minBurst = INT_MAX;
        for (int i = 0; i < n; i++) {
            if (p[i].arrival <= time && p[i].remaining > 0) {
                if (p[i].remaining < minBurst || (p[i].remaining == minBurst && p[i].arrival <
p[minIndex].arrival)) {
                    minBurst = p[i].remaining;
                    minIndex = i;
                }
            }
        }

        if (minIndex == -1) {
            time++;
            continue;
        }
        if (p[minIndex].started == 0) {
            p[minIndex].response = time - p[minIndex].arrival;
            p[minIndex].started = 1;
        }
```

```
        p[minIndex].remaining--;
        time++;
        if (p[minIndex].remaining == 0) {
            p[minIndex].completion = time;
            p[minIndex].turnaround = p[minIndex].completion - p[minIndex].arrival;
            p[minIndex].waiting = p[minIndex].turnaround - p[minIndex].burst;
            completed++;
        }
    }
}

void displayResults(Process p[], int n, const char *title) {
    printf("\n--- %s ---\n", title);
    printf("\nPID\tAT\tBT\tCT\tTAT\tWT\tRT\n");
    float totalWT = 0, totalTAT = 0, totalRT = 0;
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\n", p[i].id, p[i].arrival, p[i].burst, p[i].completion,
p[i].turnaround, p[i].waiting, p[i].response);
        totalWT += p[i].waiting;
        totalTAT += p[i].turnaround;
        totalRT += p[i].response;
    }
    printf("Average Waiting Time: %.2f\n", totalWT / n);
    printf("Average Turnaround Time: %.2f\n", totalTAT / n);
    printf("Average Response Time: %.2f\n", totalRT / n);
}

int main() {
    int n, choice;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    Process p[n], temp[n];
```

```
    printf("Enter Arrival Time and Burst Time:\n");
    for (int i = 0; i < n; i++) {
        p[i].id = i + 1; // Auto-generate PID
        scanf("%d %d", &p[i].arrival, &p[i].burst);
        p[i].remaining = p[i].burst;
        p[i].waiting = p[i].turnaround = p[i].completion = p[i].response = p[i].started = 0;
    }
    sjfPreemptive(p, n);
    displayResults(p, n, "Shortest Job First (Preemptive)");
    return 0;
}
```

Result:

```
Enter number of processes: 4
Enter Arrival Time and Burst Time:
0 8 1 4 2 9 3 5

--- Shortest Job First (Preemptive) ---

PID AT  BT  CT  TAT WT  RT
P1  0   8   17  17  9   0
P2  1   4   5   4   0   0
P3  2   9   26  24  15  15
P4  3   5   10  7   2   2
Average Waiting Time: 6.50
Average Turnaround Time: 13.00
Average Response Time: 4.25
```

Program - 2

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

=>CPU SCHEDULING:

```
#include <stdio.h>
#define MAX 10
typedef struct {
   int pid, at, bt, pt, remaining_bt, ct, tat, wt, rt, is_completed, st;
} Process;
void nonPreemptivePriority(Process p[], int n) {
   int time = 0, completed = 0;

   while (completed < n) {
      int lowest_priority = 9999, selected = -1;
      for (int i = 0; i < n; i++) {
         if (p[i].at <= time && !p[i].is_completed && p[i].pt < lowest_priority) {
            lowest_priority = p[i].pt;
            selected = i;
         }
      }
      if (selected == -1) {
         time++;
         continue;
      }
      if (p[selected].rt == -1) {
         p[selected].st = time;
         p[selected].rt = time - p[selected].at;
      }
      time += p[selected].bt;
      p[selected].ct = time;
      p[selected].tat = p[selected].ct - p[selected].at;
```

```
        p[selected].wt = p[selected].tat - p[selected].bt;
        p[selected].is_completed = 1;
        completed++;
    }
}
void preemptivePriority(Process p[], int n) {
    int time = 0, completed = 0;
    while (completed < n) {
        int lowest_priority = 9999, selected = -1;

        for (int i = 0; i < n; i++) {
            if (p[i].at <= time && p[i].remaining_bt > 0 && p[i].pt < lowest_priority) {
                lowest_priority = p[i].pt;
                selected = i;
            }
        }

        if (selected == -1) {
            time++;
            continue;
        }

        if (p[selected].rt == -1) {
            p[selected].st = time;
            p[selected].rt = time - p[selected].at;
        }

        p[selected].remaining_bt--;
        time++;

        if (p[selected].remaining_bt == 0) {
            p[selected].ct = time;
```

```
            p[selected].tat = p[selected].ct - p[selected].at;
            p[selected].wt = p[selected].tat - p[selected].bt;
            completed++;
        }
    }
}
void displayProcesses(Process p[], int n) {
    float avg_tat = 0, avg_wt = 0, avg_rt = 0;
    printf("\nPID\tAT\tBT\tPriority\tCT\tTAT\tWT\tRT\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t\t%d\t%d\t%d\t%d\n",
               p[i].pid, p[i].at, p[i].bt, p[i].pt, p[i].ct, p[i].tat, p[i].wt, p[i].rt);
        avg_tat += p[i].tat;
        avg_wt += p[i].wt;
        avg_rt += p[i].rt;
    }
    printf("\nAverage TAT: %.2f", avg_tat / n);
    printf("\nAverage WT: %.2f", avg_wt / n);
    printf("\nAverage RT: %.2f\n", avg_rt / n);
}
int main() {
    Process p[MAX];
    int n, choice;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("\nEnter Arrival Time, Burst Time, and Priority for Process %d:\n", p[i].pid);
        printf("Arrival Time: ");
        scanf("%d", &p[i].at);
        printf("Burst Time: ");
        scanf("%d", &p[i].bt);
```

```
        printf("Priority (lower number means higher priority): ");
        scanf("%d", &p[i].pt);
        p[i].remaining_bt = p[i].bt;
        p[i].is_completed = 0;
        p[i].rt = -1;
    }
    while (1) {
        printf("\nPriority Scheduling Menu:\n");
        printf("1. Non-Preemptive Priority Scheduling\n");
        printf("2. Preemptive Priority Scheduling\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                nonPreemptivePriority(p, n);
                printf("Non-Preemptive Scheduling Completed!\n");
                displayProcesses(p, n);
                break;
            case 2:
                preemptivePriority(p, n);
                printf("Preemptive Scheduling Completed!\n");
                displayProcesses(p, n);
                break;
            case 3:
                printf("Exiting...\n");
                return 0;
            default:
                printf("Invalid choice! Try again.\n");
        }
    }
```

```
    return 0;
}
```

Result:

```
Enter Arrival Time, Burst Time, and Priority for Process 1:
Arrival Time: 0
Burst Time: 5
Priority (lower number means higher priority): 4

Enter Arrival Time, Burst Time, and Priority for Process 2:
Arrival Time: 2
Burst Time: 4
Priority (lower number means higher priority): 2

Enter Arrival Time, Burst Time, and Priority for Process 3:
Arrival Time: 2
Burst Time: 2
Priority (lower number means higher priority): 6

Enter Arrival Time, Burst Time, and Priority for Process 4:
Arrival Time: 4
Burst Time: 4
Priority (lower number means higher priority): 3

Priority Scheduling Menu:
1. Non-Preemptive Priority Scheduling
2. Preemptive Priority Scheduling
3. Exit
Enter your choice: 1
Non-Preemptive Scheduling Completed!

PID     AT      BT      Priority        CT      TAT     WT      RT
1       0       5       4               5       5       0       0
2       2       4       2               9       7       3       3
3       2       2       6               15      13      11      11
4       4       4       3               13      9       5       5

Average TAT: 8.50
Average WT: 4.75
Average RT: 4.75

Priority Scheduling Menu:
1. Non-Preemptive Priority Scheduling
2. Preemptive Priority Scheduling
3. Exit
Enter your choice: 2
Preemptive Scheduling Completed!

PID     AT      BT      Priority        CT      TAT     WT      RT
1       0       5       4               13      13      8       0
2       2       4       2               6       4       0       3
3       2       2       6               15      13      11      11
4       4       4       3               10      6       2       5

Average TAT: 9.00
Average WT: 5.25
Average RT: 4.75
```

Program - 3

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.

=>MULTI LEVEL SCHEDULING:

```
#include <stdio.h>
#define MAX_PROCESSES 10
#define TIME_QUANTUM 2
typedef struct {
```

```
    int burst_time, arrival_time, queue_type, waiting_time, turnaround_time, response_time,
remaining_time;
} Process;

void round_robin(Process processes[], int n, int time_quantum, int *time) {
    int done, i;
    do {
        done = 1;
        for (i = 0; i < n; i++) {
            if (processes[i].remaining_time > 0) {
                done = 0;
                if (processes[i].remaining_time > time_quantum) {
                    *time += time_quantum;
                    processes[i].remaining_time -= time_quantum;
                } else {
                    *time += processes[i].remaining_time;
                    processes[i].waiting_time = *time - processes[i].arrival_time - processes[i].burst_time;
                    processes[i].turnaround_time = *time - processes[i].arrival_time;
                    processes[i].response_time = processes[i].waiting_time;
                    processes[i].remaining_time = 0;
                }
            }
        }
    } while (!done);
}

void fcfs(Process processes[], int n, int *time) {
    for (int i = 0; i < n; i++) {
        if (*time < processes[i].arrival_time) {
            *time = processes[i].arrival_time;
        }
        processes[i].waiting_time = *time - processes[i].arrival_time;
```

```
        processes[i].turnaround_time = processes[i].waiting_time + processes[i].burst_time;
        processes[i].response_time = processes[i].waiting_time;
        *time += processes[i].burst_time;
    }
}

int main() {
    Process processes[MAX_PROCESSES], system_queue[MAX_PROCESSES],
user_queue[MAX_PROCESSES];
    int n, sys_count = 0, user_count = 0, time = 0;
    float avg_waiting = 0, avg_turnaround = 0, avg_response = 0, throughput;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        printf("Enter Burst Time, Arrival Time and Queue of P%d: ", i + 1);
        scanf("%d %d %d", &processes[i].burst_time, &processes[i].arrival_time,
&processes[i].queue_type);
        processes[i].remaining_time = processes[i].burst_time;
        if (processes[i].queue_type == 1) {
            system_queue[sys_count++] = processes[i];
        } else {
            user_queue[user_count++] = processes[i];
        }
    }
    for (int i = 0; i < user_count - 1; i++) {
        for (int j = 0; j < user_count - i - 1; j++) {
            if (user_queue[j].arrival_time > user_queue[j + 1].arrival_time) {
                Process temp = user_queue[j];
                user_queue[j] = user_queue[j + 1];
                user_queue[j + 1] = temp;
            }
        }
```

```
    }
    printf("\nQueue 1 is System Process\nQueue 2 is User Process\n");
    round_robin(system_queue, sys_count, TIME_QUANTUM, &time);
    fcfs(user_queue, user_count, &time);
    printf("\nProcess  Waiting Time  Turn Around Time  Response Time\n");
    for (int i = 0; i < sys_count; i++) {
        avg_waiting += system_queue[i].waiting_time;
        avg_turnaround += system_queue[i].turnaround_time;
        avg_response += system_queue[i].response_time;
        printf("%d        %d           %d              %d\n", i + 1, system_queue[i].waiting_time,
system_queue[i].turnaround_time, system_queue[i].response_time);
    }
    for (int i = 0; i < user_count; i++) {
        avg_waiting += user_queue[i].waiting_time;
        avg_turnaround += user_queue[i].turnaround_time;
        avg_response += user_queue[i].response_time;
        printf("%d        %d           %d              %d\n", i + 1 + sys_count, user_queue[i].waiting_time,
user_queue[i].turnaround_time, user_queue[i].response_time);
    }
    avg_waiting /= n;
    avg_turnaround /= n;
    avg_response /= n;
    throughput = (float)n / time;
    printf("\nAverage Waiting Time: %.2f", avg_waiting);
    printf("\nAverage Turn Around Time: %.2f", avg_turnaround);
    printf("\nAverage Response Time: %.2f", avg_response);
    printf("\nThroughput: %.2f", throughput);
    printf("\nProcess returned %d (0x%d) execution time: %.3f s\n", time, time, (float)time);
    return 0;
}
```

Result:

```
Enter number of processes: 4
Enter Burst Time, Arrival Time and Queue of P1: 2 0 1
Enter Burst Time, Arrival Time and Queue of P2: 1 0 2
Enter Burst Time, Arrival Time and Queue of P3: 5 0 1
Enter Burst Time, Arrival Time and Queue of P4: 3 0 2

Queue 1 is System Process
Queue 2 is User Process

Process  Waiting Time  Turn Around Time  Response Time
1         0            2                0
2         2            7                2
3         7            8                7
4         8            11                8

Average Waiting Time: 4.25
Average Turn Around Time: 7.00
Average Response Time: 4.25
Throughput: 0.36
Process returned 11 (0x11) execution time: 11.000 s

Process returned 0 (0x0)   execution time : 21.307 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

=> Rate Monotonic

```
#include <stdio.h>
#define MAX_PROCESSES 10
typedef struct {
    int id;
    int burst_time;
    int period;
    int remaining_time;
    int next_deadline;
} Process;


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

```
            }
        }
    }
}

int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}
int lcm(int a, int b) {
    return (a * b) / gcd(a, b);
}

int calculate_lcm(Process processes[], int n) {
    int result = processes[0].period;
    for (int i = 1; i < n; i++) {
        result = lcm(result, processes[i].period);
    }
    return result;
}

double utilization_factor(Process processes[], int n) {
    double sum = 0;
    for (int i = 0; i < n; i++) {
        sum += (double)processes[i].burst_time / processes[i].period;
    }
    return sum;
}
double rms_threshold(int n) {
    return n * (pow(2.0, 1.0 / n) - 1);
}

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

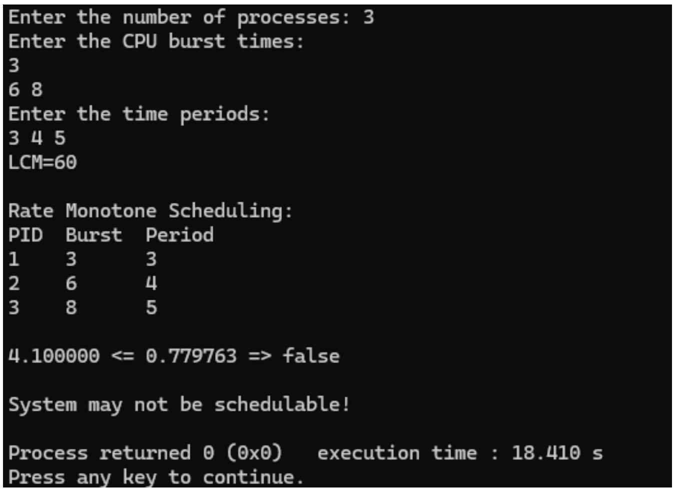
```
    int lcm_period = calculate_lcm(processes, n);
    printf("LCM=%d\n\n", lcm_period);
    printf("Rate Monotone Scheduling:\n");
    printf("PID  Burst  Period\n");
    for (int i = 0; i < n; i++) {
        printf("%d    %d      %d\n", processes[i].id, processes[i].burst_time, processes[i].period);
    }

    double utilization = utilization_factor(processes, n);
    double threshold = rms_threshold(n);
    printf("\n%.6f <= %.6f => %s\n", utilization, threshold, (utilization <= threshold) ? "true" :
"false");
    if (utilization > threshold) {
        printf("\nSystem may not be schedulable!\n");
        return;
    }
    int timeline = 0, executed = 0;
    while (timeline < lcm_period) {
        int selected = -1;
        for (int i = 0; i < n; i++) {
            if (timeline % processes[i].period == 0) {
                processes[i].remaining_time = processes[i].burst_time;
            }
            if (processes[i].remaining_time > 0) {
                selected = i;
                break;
            }
        }
        if (selected != -1) {
            printf("Time %d: Process %d is running\n", timeline, processes[selected].id);
            processes[selected].remaining_time--;
            executed++;
```

```
        } else {
            printf("Time %d: CPU is idle\n", timeline);
        }
        timeline++;
    }
}


int main() {
    int n;
    Process processes[MAX_PROCESSES];
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    printf("Enter the CPU burst times:\n");
    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        scanf("%d", &processes[i].burst_time);
        processes[i].remaining_time = processes[i].burst_time;
    }
    printf("Enter the time periods:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &processes[i].period);
    }
    sort_by_period(processes, n);
    rate_monotonic_scheduling(processes, n);
    return 0;
}
```

Result :



```
Enter the number of processes: 3
Enter the CPU burst times:
3
6 8
Enter the time periods:
3 4 5
LCM=60

Rate Monotone Scheduling:
PID  Burst  Period
1    3      3
2    6      4
3    8      5

4.100000 <= 0.779763 => false

System may not be schedulable!

Process returned 0 (0x0)   execution time : 18.410 s
Press any key to continue.
```

=> Earliest Deadline

```
#include <stdio.h>
int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

int lcm(int a, int b) {
    return (a * b) / gcd(a, b);
}

struct Process {
    int id, burst_time, deadline, period;
};

void earliest_deadline_first(struct Process p[], int n, int time_limit) {
```

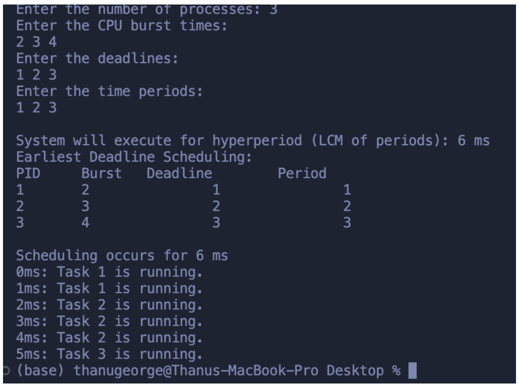
```c
    int time = 0;
    printf("Earliest Deadline Scheduling:\n");
    printf("PID\tBurst\tDeadline\tPeriod\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", p[i].id, p[i].burst_time, p[i].deadline, p[i].period);
    }

    printf("\nScheduling occurs for %d ms\n", time_limit);
    while (time < time_limit) {
        int earliest = -1;
        for (int i = 0; i < n; i++) {
            if (p[i].burst_time > 0) {
                if (earliest == -1 || p[i].deadline < p[earliest].deadline) {
                    earliest = i;
                }
            }
        }
        if (earliest == -1) break;
        printf("%dms: Task %d is running.\n", time, p[earliest].id);
        p[earliest].burst_time--;
        time++;
    }
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process processes[n];
    printf("Enter the CPU burst times:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &processes[i].burst_time);
```

```
        processes[i].id = i + 1;
    }
    printf("Enter the deadlines:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &processes[i].deadline);
    }
    printf("Enter the time periods:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &processes[i].period);
    }
    int hyperperiod = processes[0].period;
    for (int i = 1; i < n; i++) {
        hyperperiod = lcm(hyperperiod, processes[i].period);
    }
    printf("\nSystem will execute for hyperperiod (LCM of periods): %d ms\n", hyperperiod);
    earliest_deadline_first(processes, n, hyperperiod);
    return 0;
}
```

Result:



```
Enter the number of processes: 3
Enter the CPU burst times:
2 3 4
Enter the deadlines:
1 2 3
Enter the time periods:
1 2 3

System will execute for hyperperiod (LCM of periods): 6 ms
Earliest Deadline Scheduling:
PID     Burst   Deadline        Period
1       2               1               1
2       3               2               2
3       4               3               3

Scheduling occurs for 6 ms
0ms: Task 1 is running.
1ms: Task 1 is running.
2ms: Task 2 is running.
3ms: Task 2 is running.
4ms: Task 2 is running.
5ms: Task 3 is running.
(base) thanugeorge@Thanus-MacBook-Pro Desktop %
```

Program 5

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

=> Producer Consumer

```
#include <stdio.h>

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

void wait(int *s) {
    --(*s);
}

void signal(int *s) {
    ++(*s);
}

void producer() {
    wait(&empty);
    wait(&mutex);
    x++;
    printf("The item produced is %d\n", x);
    signal(&mutex);
    signal(&full);
}

void consumer() {
    wait(&full);
    wait(&mutex);
    printf("Consumed item %d\n", x);
    x--;
    signal(&mutex);
    signal(&empty);
}

int main() {
    int choice;
    do {
        printf("\n1. Produce\n2. Consume\n3. Exit\nEnter choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                if ((mutex == 1) && (empty != 0)) {
                    producer();
                } else {
```

```
                printf("The buffer is full\n");
            }
            break;
        case 2:
            if ((mutex == 1) && (full != 0)) {
                consumer();
            } else {
                printf("The buffer is empty\n");
            }
            break;
        case 3:
            printf("Exiting.\n");
            break;
        default:
            printf("Invalid choice.\n");
        }
    } while (choice != 3);
    return 0;
}
```

Result:

```
1. Produce
2. Consume
3. Exit
Enter choice: 2
The buffer is empty

1. Produce
2. Consume
3. Exit
Enter choice: 1
The item produced is 1

1. Produce
2. Consume
3. Exit
Enter choice: 2
Consumed item 1

1. Produce
2. Consume
3. Exit
Enter choice: 3
Exiting.
```

Program 6
Write a C program to simulate the concept of Dining Philosophers problem

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>
#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

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

sem_t mutex;
sem_t S[N];

void* philosopher(void* num);
void take_fork(int phnum);
void put_fork(int phnum);
void test(int phnum);

int main() {
    int i;
    pthread_t thread_id[N];

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

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

    for (i = 0; i < N; i++) {
        // create philosopher processes
        pthread_create(&thread_id[i], NULL, philosopher, &phil[i]);
        printf("Philosopher %d is thinking\n", i + 1);
    }

    for (i = 0; i < N; i++)
        pthread_join(thread_id[i], NULL);
```

```
    return 0;
}

void test(int phnum) {
    if (state[phnum] == HUNGRY
        && state[LEFT] != EATING
        && state[RIGHT] != EATING) {
        // state that eating
        state[phnum] = EATING;
        sleep(2);
        printf("Philosopher %d takes fork %d and %d\n", phnum + 1, LEFT + 1, phnum + 1);
        printf("Philosopher %d is Eating\n", phnum + 1);
        sem_post(&S[phnum]);
    }
}

// take up chopsticks
void take_fork(int phnum) {
    sem_wait(&mutex);
    // state that hungry
    state[phnum] = HUNGRY;
    printf("Philosopher %d is Hungry\n", phnum + 1);

    // eat if neighbours are not eating
    test(phnum);
    sem_post(&mutex);
    // if unable to eat wait to be signalled
    sem_wait(&S[phnum]);
    sleep(1);
}

// put down chopsticks
void put_fork(int phnum) {
    sem_wait(&mutex);

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

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

    test(LEFT);
    test(RIGHT);
```

```
    sem_post(&mutex);
}

void* philosopher(void* num) {
    int* i = (int*)num;
    while (1) {
        sleep(1);
        take_fork(*i);
        sleep(0);
        put_fork(*i);
    }
}
```

Result:

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

Program 7
Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

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

int main() {
    int n, m, i, j, k;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    printf("Enter number of resources: ");
    scanf("%d", &m);

    int alloc[n][m], max[n][m], avail[m];
    int need[n][m];

    printf("Enter allocation matrix (%d x %d):\n", n, m);
    for (i = 0; i < n; i++) {
        printf("Allocation for process %d: ", i);
        for (j = 0; j < m; j++)
            scanf("%d", &alloc[i][j]);
    }

    printf("Enter max matrix (%d x %d):\n", n, m);
    for (i = 0; i < n; i++) {
        printf("Max for process %d: ", i);
        for (j = 0; j < m; j++)
            scanf("%d", &max[i][j]);
    }

    printf("Enter available resources (%d values): ", m);
    for (i = 0; i < m; i++)
        scanf("%d", &avail[i]);

    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            need[i][j] = max[i][j] - alloc[i][j];

    bool finish[n];
    int safeSeq[n];
    int count = 0;
```

```
    for (i = 0; i < n; i++)
        finish[i] = false;

    while (count < n) {
        bool found = false;
        for (i = 0; i < n; i++) {
            if (!finish[i]) {
                for (j = 0; j < m; j++)
                    if (need[i][j] > avail[j])
                        break;

                if (j == m) {
                    for (k = 0; k < m; k++)
                        avail[k] += alloc[i][k];

                    safeSeq[count++] = i;
                    finish[i] = true;
                    found = true;
                }
            }
        }

        if (!found) {
            printf("System is not in safe state.\n");
            return 1;
        }
    }

    printf("System is in safe state.\n");
    printf("Safe sequence is: ");
    for (i = 0; i < n; i++) {
        printf("P%d", safeSeq[i]);
        if (i != n - 1)
            printf(" -> ");
    }
    printf("\n");

    return 0;
}
```

Result:

```
Enter number of processes: 5
Enter number of resources: 3
Enter allocation matrix (5 x 3):
Allocation for process 0: 0 1 0
Allocation for process 1: 2 0 0
Allocation for process 2: 3 0 2
Allocation for process 3: 2 1 1
Allocation for process 4: 0 0 2
Enter max matrix (5 x 3):
Max for process 0: 7 5 3
Max for process 1: 3 2 2
Max for process 2: 9 0 2
Max for process 3: 2 2 2
Max for process 4: 4 3 3
Enter available resources (3 values): 3 3 2
System is in safe state.
Safe sequence is: P1 -> P3 -> P4 -> P0 -> P2
```

Program 8

Write a C program to simulate deadlock detection

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

int main() {
    int n, m, i, j, k;

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

    int allocation[n][m], request[n][m], available[m];
    int work[m];
    bool finish[n];

    printf("Enter allocation matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            scanf("%d", &allocation[i][j]);

    printf("Enter request matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            scanf("%d", &request[i][j]);
```

```
printf("Enter available matrix:\n");
for (i = 0; i < m; i++) {
    scanf("%d", &available[i]);
    work[i] = available[i];
}

for (i = 0; i < n; i++) {
    bool zero_allocation = true;
    for (j = 0; j < m; j++) {
        if (allocation[i][j] != 0) {
            zero_allocation = false;
            break;
        }
    }
    finish[i] = zero_allocation;
}

bool found_process;
do {
    found_process = false;
    for (i = 0; i < n; i++) {
        if (!finish[i]) {
            bool can_allocate = true;
            for (j = 0; j < m; j++) {
                if (request[i][j] > work[j]) {
                    can_allocate = false;
                    break;
                }
            }
            if (can_allocate) {
                for (k = 0; k < m; k++)
                    work[k] += allocation[i][k];
                finish[i] = true;
                printf("Process %d can finish.\n", i);
                found_process = true;
            }
        }
    }
} while (found_process);

bool deadlock = false;
for (i = 0; i < n; i++) {
    if (!finish[i]) {
        deadlock = true;
```

```
            break;
        }
    }

    if (deadlock)
        printf("System is in a deadlock state.\n");
    else
        printf("System is not in a deadlock state.\n");

    return 0;
}
```

Result:

```
Enter number of processes and resources:
5 3
Enter allocation matrix:
0 1 0
2 0 0
3 0 3
2 1 1
0 0 2
Enter request matrix:
0 0 0
2 0 2
0 0 1
1 0 0
0 0 2
Enter available matrix:
0 0 0
Process 0 can finish.
System is in a deadlock state.
```

Program 9:
Write a C program to simulate the following contiguous memory allocation techniques
a) Worst-fit
b)Best-fit
c)First-fit

```
#include <stdio.h>

struct Block {
    int size;
    int allocated;

};

struct File {
    int size;
    int block_no;
};

void resetBlocks(struct Block blocks[], int n) {
    for (int i = 0; i < n; i++) {
        blocks[i].allocated = 0;
    }
}

void firstFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
    printf("\n\tMemory Management Scheme – First Fit\n");
    printf("File_no:\tFile_size\tBlock_no:\tBlock_size:\n");
    for (int i = 0; i < n_files; i++) {
        files[i].block_no = -1;
        for (int j = 0; j < n_blocks; j++) {
            if (!blocks[j].allocated && blocks[j].size >= files[i].size) {
                files[i].block_no = j + 1;
                blocks[j].allocated = 1;
                printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, files[i].size, j + 1, blocks[j].size);
                break;
            }
        }
        if (files[i].block_no == -1) {
            printf("%d\t\t%d\t\t_\t\t_\n", i + 1, files[i].size);
        }
    }
```

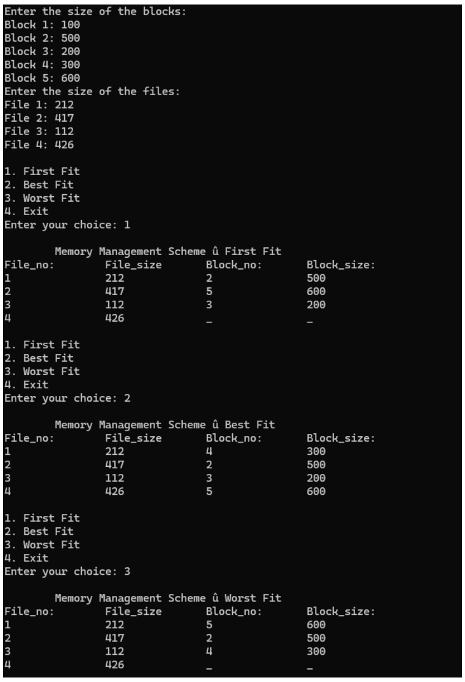
```
}

void bestFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
    printf("\n\tMemory Management Scheme – Best Fit\n");
    printf("File_no:\tFile_size\tBlock_no:\tBlock_size:\n");
    for (int i = 0; i < n_files; i++) {
        int bestIdx = -1;
        for (int j = 0; j < n_blocks; j++) {
            if (!blocks[j].allocated && blocks[j].size >= files[i].size) {
                if (bestIdx == -1 || blocks[j].size < blocks[bestIdx].size) {
                    bestIdx = j;
                }
            }
        }
        if (bestIdx != -1) {
            blocks[bestIdx].allocated = 1;
            files[i].block_no = bestIdx + 1;
            printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, files[i].size, bestIdx + 1, blocks[bestIdx].size);
        } else {
            printf("%d\t\t%d\t\t_\t\t_\n", i + 1, files[i].size);
        }
    }
}

void worstFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
    printf("\n\tMemory Management Scheme – Worst Fit\n");
    printf("File_no:\tFile_size\tBlock_no:\tBlock_size:\n");
    for (int i = 0; i < n_files; i++) {
        int worstIdx = -1;
        for (int j = 0; j < n_blocks; j++) {
            if (!blocks[j].allocated && blocks[j].size >= files[i].size) {
                if (worstIdx == -1 || blocks[j].size > blocks[worstIdx].size) {
                    worstIdx = j;
                }
            }
        }
        if (worstIdx != -1) {
            blocks[worstIdx].allocated = 1;
            files[i].block_no = worstIdx + 1;
            printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, files[i].size, worstIdx + 1, blocks[worstIdx].size);
        } else {
            printf("%d\t\t%d\t\t_\t\t_\n", i + 1, files[i].size);
        }
    }
```

```
int main() {
    int n_blocks, n_files, choice;
    printf("Memory Management Scheme\n");
    printf("Enter the number of blocks: ");
    scanf("%d", &n_blocks);
    printf("Enter the number of files: ");
    scanf("%d", &n_files);
    struct Block blocks[n_blocks];
    struct File files[n_files];
    printf("\nEnter the size of the blocks:\n");
    for (int i = 0; i < n_blocks; i++) {
        printf("Block %d: ", i + 1);
        scanf("%d", &blocks[i].size);
        blocks[i].allocated = 0;
    }
    printf("Enter the size of the files:\n");
    for (int i = 0; i < n_files; i++) {
        printf("File %d: ", i + 1);
        scanf("%d", &files[i].size);
    }
    do {
        printf("\n1. First Fit\n2. Best Fit\n3. Worst Fit\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        resetBlocks(blocks, n_blocks); // Reset block allocation before each strategy
        switch (choice) {
            case 1:
                firstFit(blocks, n_blocks, files, n_files);
                break;
            case 2:
                bestFit(blocks, n_blocks, files, n_files);
                break;
            case 3:
                worstFit(blocks, n_blocks, files, n_files);
                break;
            case 4:
                printf("\nExiting...\n");
                break;
            default:
                printf("Invalid choice.\n");
        }
    } while (choice != 4);
    return 0;
}
```

Result:



```
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: 417
File 3: 112
File 4: 426

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

        Memory Management Scheme û First Fit
File_no:        File_size       Block_no:       Block_size:
1               212             2               500
2               417             5               600
3               112             3               200
4               426             _               _

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

        Memory Management Scheme û Best Fit
File_no:        File_size       Block_no:       Block_size:
1               212             4               300
2               417             2               500
3               112             3               200
4               426             5               600

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

        Memory Management Scheme û Worst Fit
File_no:        File_size       Block_no:       Block_size:
1               212             5               600
2               417             2               500
3               112             4               300
4               426             _               _
```

Program 10
Write a C program to simulate page replacement algorithms
a) FIFO
b) LRU
c) Optimal

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

int main() {
    int n, frames, i, j, k, found, index = 0, page_faults = 0, hits = 0;
    char pages[100];

    printf("Enter the size of the pages:\n");
    scanf("%d", &n);
    printf("Enter the page strings:\n");
    scanf("%s", pages);
    printf("Enter the no of page frames:\n");
    scanf("%d", &frames);

    int mem[frames];
```

```
    for (i = 0; i < frames; i++) mem[i] = -1;

    for (i = 0; i < n; i++) {
        found = 0;
        for (j = 0; j < frames; j++) {
            if (mem[j] == pages[i] - '0') {
                hits++;
                found = 1;
                break;
            }
        }
        if (!found) {
            mem[index] = pages[i] - '0';
            index = (index + 1) % frames;
            page_faults++;
        }
    }

    printf("FIFO Page Faults: %d, Page Hits: %d\n", page_faults, hits);
    return 0;
}
```

RESULT:

```
Enter the size of the pages:
7
Enter the page strings:
1 3 0 3 5 6 3
Enter the no of page frames:
FIFO Page Faults: 2, Page Hits: 5
```

=>LRU

```
#include <stdio.h>

int main() {
    int n, frames, i, j, k, page_faults = 0, hits = 0;
    char pages[100];
    int mem[10], used[10];

    printf("Enter the size of the pages:\n");
    scanf("%d", &n);
```

```
    printf("Enter the page strings:\n");
    scanf("%s", pages);
    printf("Enter the no of page frames:\n");
    scanf("%d", &frames);

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

    for (i = 0; i < n; i++) {
        int page = pages[i] - '0';
        int found = 0;

        for (j = 0; j < frames; j++) {
            if (mem[j] == page) {
                hits++;
                used[j] = i;
                found = 1;
                break;
            }
        }

        if (!found) {
            int lru = 0;
            for (j = 1; j < frames; j++) {
                if (used[j] < used[lru]) lru = j;
            }
            mem[lru] = page;
            used[lru] = i;
            page_faults++;
        }
    }

    printf("LU Page Faults: %d, Page Hits: %d\n", page_faults, hits);
    return 0;
}
```

RESULT:

```
Enter the size of the pages:
7
Enter the page strings:
1 3 0 3 5 6 3
Enter the no of page frames:
LRU Page Faults: 3, Page Hits: 4

Process returned 0 (0x0)   execution time : 22.105 s
Press any key to continue.
```

=>OPTIMAL

```
#include <stdio.h>

int main() {
    int n, frames, i, j, k, page_faults = 0, hits = 0;
    printf("Enter the size of the pages:\n");
    scanf("%d", &n);
    char pages[n + 1];
    printf("Enter the page strings:\n");
    scanf("%s", pages);
    printf("Enter the no of page frames:\n");
    scanf("%d", &frames);
    int mem[frames], next_use[frames];
    for (i = 0; i < frames; i++) {
        mem[i] = -1;
    }


    for (i = 0; i < n; i++) {
        int page = pages[i] - '0';
        int found = 0;
        for (j = 0; j < frames; j++) {
            if (mem[j] == page) {
                hits++;
                found = 1;
                break;
            }
        }
        if (!found) {
            if (page_faults < frames) {
```

```
            mem[page_faults++] = page;
        } else {
            for (j = 0; j < frames; j++) {
                next_use[j] = -1;
                for (k = i + 1; k < n; k++) {
                    if (mem[j] == pages[k] - '0') {
                        next_use[j] = k;
                        break;
                    }
                }
            }

            int farthest = 0;
            for (j = 1; j < frames; j++) {
                if (next_use[j] > next_use[farthest]) {
                    farthest = j;
                }
            }
            mem[farthest] = page;
            page_faults++;
        }
    }
}

printf("Optimal Page Faults: %d, Page Hits: %d\n", page_faults, hits);
return 0;
}
```

RESULT:

```
Enter the size of the pages:
7
Enter the page strings:
1 3 0 3 5 6 3
Enter the no of page frames:
Optimal Page Faults: 7, Page Hits: 0
```