VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS

Submitted by

SHREYA SATHYANARAYANA (1BM23CS318)

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 Jun 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 Shreya Sathyanarayana **(1BM23CS333),** 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 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.

Amruta BP
Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr. Kavitha Sooda
Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

	a) sequential b) indexed c) linked	

Course Outcome

CO1	Apply the different concepts and functionalities of Operating System.
CO2	Analyse various Operating system strategies and techniques.
CO3	Demonstrate the different functionalities of Operating System.
CO4	Conduct practical experiments to implement the functionalities of Operating system.

Program -1

Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time.
a) FCFS b) SJF c) Priority d) Round Robin

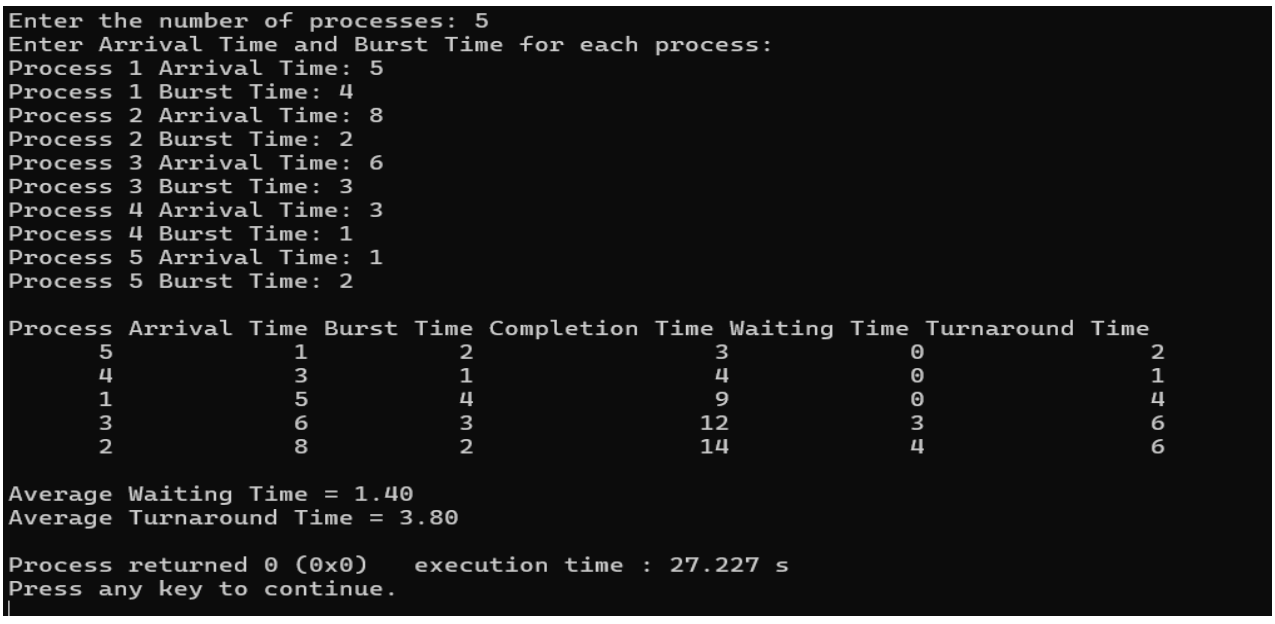
Code: a) FCFS

```
#include <stdio.h>

void sortByArrivalTime(int processes[], int at[], int bt[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (at[j] > at[j + 1]) {
                int temp = at[j];
                at[j] = at[j + 1];
                at[j + 1] = temp;

                temp = bt[j];
                bt[j] = bt[j + 1];
                bt[j + 1] = temp;

                temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
}

int main() {
    int n;
```

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

int processes[n], at[n], bt[n], wt[n], tat[n], ct[n];
int total_wt = 0, total_tat = 0;

printf("Enter Arrival Time and Burst Time for each process:\n");
for (int i = 0; i < n; i++) {
    processes[i] = i + 1;
    printf("Process %d Arrival Time: ", i + 1);
    scanf("%d", &at[i]);
    printf("Process %d Burst Time: ", i + 1);
    scanf("%d", &bt[i]);
}

sortByArrivalTime(processes, at, bt, n);

for (int i = 0; i < n; i++) {
    if (i == 0) {
        ct[i] = at[i] + bt[i];
    } else {
        ct[i] = (ct[i - 1] > at[i] ? ct[i - 1] : at[i]) + bt[i];
    }
    tat[i] = ct[i] - at[i];
    wt[i] = tat[i] - bt[i];
    total_wt += wt[i];
    total_tat += tat[i];
}
```

```
    printf("\nProcess Arrival Time Burst Time Completion Time Waiting Time Turnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%7d %12d %10d %16d %12d %15d\n", processes[i], at[i], bt[i], ct[i], wt[i], tat[i]);
    }

    printf("\nAverage Waiting Time = %.2f\n", (float)total_wt / n);
    printf("Average Turnaround Time = %.2f\n", (float)total_tat / n);

    return 0;
}
```

Result:



b) SJF (Preemptive)

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

void findWaitingTime(int n, int at[], int bt[], int wt[]) {
    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;
    int check = 0;

    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;
                check = 1;
            }
        }

        if (check == 0) {
            t++;
            continue;
        }
```

```
        rt[shortest]--;
        minm = rt[shortest];
        if (minm == 0) {
            minm = INT_MAX;
        }

        if (rt[shortest] == 0) {
            complete++;
            check = 0;

            finish_time = t + 1;
            wt[shortest] = finish_time - bt[shortest] - at[shortest];

            if (wt[shortest] < 0) {
                wt[shortest] = 0;
            }
        }

        t++;
    }
}

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

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

    findWaitingTime(n, at, bt, wt);
    findTurnAroundTime(n, bt, wt, tat);

    printf("Processes Arrival Time Burst Time Waiting Time Turnaround Time\n");
    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        printf("%8d %12d %10d %12d %15d\n", processes[i], at[i], bt[i], wt[i], tat[i]);
    }

    printf("\nAverage Waiting Time = %.2f", (float)total_wt / n);
    printf("\nAverage Turnaround Time = %.2f\n", (float)total_tat / n);
}

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

    int processes[n], at[n], bt[n];
    for (int i = 0; i < n; i++) {
        processes[i] = i + 1;
        printf("Enter Arrival Time and Burst Time for Process %d: ", i + 1);
        scanf("%d %d", &at[i], &bt[i]);
    }
```

```
    findAvgTime(n, processes, at, bt);

    return 0;
}
```

Result:

```
Enter the number of processes: 5
Enter Arrival Time and Burst Time for Process 1: 0 9
Enter Arrival Time and Burst Time for Process 2: 1 6
Enter Arrival Time and Burst Time for Process 3: 2 4
Enter Arrival Time and Burst Time for Process 4: 3 2
Enter Arrival Time and Burst Time for Process 5: 6 1
Processes Arrival Time Burst Time Waiting Time Turnaround Time
       1            0          9           13              22
       2            1          6            7              13
       3            2          4            3               7
       4            3          2            0               2
       5            6          1            0               1

Average Waiting Time = 4.60
Average Turnaround Time = 9.00

Process returned 0 (0x0)   execution time : 26.422 s
Press any key to continue.
```

C) SJF(Non-preemptive)

Code:

```
#include <stdio.h>

int findNextProcess(int n, int at[], int bt[], int completed[], int current_time) {
    int shortest = -1, min_bt = 1e9;

    for (int i = 0; i < n; i++) {
        if (!completed[i] && at[i] <= current_time && bt[i] < min_bt) {
            min_bt = bt[i];
            shortest = i;
        }
    }
    return shortest;
}

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

void calculateWaitingTime(int n, int at[], int bt[], int wt[], int completed[], int ct[]) {
    for (int i = 0; i < n; i++) {
        wt[i] = ct[i] - at[i] - bt[i];
        if (wt[i] < 0) wt[i] = 0;
    }
}
```

```
int main() {
    int n;

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

    int processes[n], at[n], bt[n], wt[n], tat[n], ct[n], rt[n], completed[n];
    float total_wt = 0, total_tat = 0;

    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        processes[i] = i + 1;
        completed[i] = 0;
        printf("Process %d Arrival Time: ", i + 1);
        scanf("%d", &at[i]);
        printf("Process %d Burst Time: ", i + 1);
        scanf("%d", &bt[i]);
    }

    int current_time = 0, completed_processes = 0;

    while (completed_processes < n) {
        int shortest = findNextProcess(n, at, bt, completed, current_time);

        if (shortest == -1) {
            current_time++;
        } else {
```

```
            rt[shortest] = current_time - at[shortest];
            if (rt[shortest] < 0)
                rt[shortest] = 0;

            current_time += bt[shortest];
            ct[shortest] = current_time;
            completed[shortest] = 1;
            completed_processes++;
        }
    }

    calculateWaitingTime(n, at, bt, wt, completed, ct);
    calculateTurnaroundTime(n, bt, wt, tat);

    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
    }

    printf("\nProcesses Arrival Time Burst Time Completion Time Waiting Time Turnaround Time Response Time\n");
    for (int i = 0; i < n; i++) {
        printf("%8d %12d %10d %15d %12d %15d %14d\n", processes[i], at[i], bt[i], ct[i], wt[i], tat[i], rt[i]);
    }

    printf("\nAverage Waiting Time = %.2f", total_wt / n);
    printf("\nAverage Turnaround Time = %.2f\n", total_tat / n);
```

```
    return 0;
}
```

Result:

```
Process 3 Arrival Time: 3
Process 3 Burst Time: 4
Process 4 Arrival Time: 5
Process 4 Burst Time: 6

Processes Arrival Time Burst Time Completion Time Waiting Time Turnaround Time Response Time
       1            0          7               7            0               7             0
       2            8          3              14            3               6             3
       3            3          4              11            4               8             4
       4            5          6              20            9              15             9

Average Waiting Time = 4.00
Average Turnaround Time = 9.00
```

c) Priority

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

typedef struct {
    int process_id;
    int arrival_time;
    int burst_time;
    int priority;
    int remaining_time;
    int waiting_time;
    int turnaround_time;
    int completion_time;
} Process;

void calculateTimes(Process processes[], int n) {
    for (int i = 0; i < n; i++) {
        processes[i].turnaround_time = processes[i].completion_time - processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;
    }
}

void priorityPreemptiveScheduling(Process processes[], int n) {
    int current_time = 0;
    int completed = 0;
    bool is_completed[n];
```

```
for (int i = 0; i < n; i++) {
    is_completed[i] = false;
    processes[i].remaining_time = processes[i].burst_time;
}

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

    for (int i = 0; i < n; i++) {
        if (!is_completed[i] && processes[i].arrival_time <= current_time &&
            processes[i].priority < highest_priority) {
            highest_priority = processes[i].priority;
            selected = i;
        }
    }

    if (selected == -1) {
        current_time++;
        continue;
    }
     processes[selected].remaining_time--;
    current_time++;
    if (processes[selected].remaining_time == 0) {
        processes[selected].completion_time = current_time;
        is_completed[selected] = true;
        completed++;
    }
```

```
    }

    calculateTimes(processes, n);
}

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

    Process processes[n];
    float total_tat = 0, total_wt = 0;

    for (int i = 0; i < n; i++) {
        processes[i].process_id = i + 1;
        printf("Enter Arrival Time, Burst Time, and Priority for Process %d: ", i + 1);
        scanf("%d %d %d", &processes[i].arrival_time, &processes[i].burst_time,
&processes[i].priority);
    }

    priorityPreemptiveScheduling(processes, n);

    printf("\nProcess ID\tArrival Time\tBurst Time\tPriority\tCompletion Time\tTurnaround
Time\tWaiting Time\n");
    for (int i = 0; i < n; i++) {
        total_tat += processes[i].turnaround_time;
        total_wt += processes[i].waiting_time;
        printf("%d\t\t%d\t\t%d\t\t\t\t%d\t\t\t%d\t\t\t%d\t\t\t%d\n",
               processes[i].process_id,
```

```
            processes[i].arrival_time,
            processes[i].burst_time,
            processes[i].priority,
            processes[i].completion_time,
            processes[i].turnaround_time,
            processes[i].waiting_time);
    }

    printf("\nAverage Turnaround Time: %.2f\n", total_tat / n);
    printf("Average Waiting Time: %.2f\n", total_wt / n);

    return 0;
}
```

Result:

```
Enter the number of processes: 5
Enter Arrival Time, Burst Time, and Priority for Process 1: 0 3 5
Enter Arrival Time, Burst Time, and Priority for Process 2: 2 2 3
Enter Arrival Time, Burst Time, and Priority for Process 3: 3 5 2
Enter Arrival Time, Burst Time, and Priority for Process 4: 4 4 4
Enter Arrival Time, Burst Time, and Priority for Process 5: 6 1 1

Process ID      Arrival Time    Burst Time Priority Completion Time Turnaround Time Waiting Time
1               0               3          515                      15               12
2               2               2          310                      8                6
3               3               5          29                       6                1
4               4               4          414                      10               6
5               6               1          17                       1                0

Average Turnaround Time: 8.00
Average Waiting Time: 5.00
```

d)Round Robin

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

void roundRobin(int n, int at[], int bt[], int quantum, int ct[], int tat[], int wt[], int rt[]) {
    int remaining_bt[n];
    bool first_execution[n];
    for (int i = 0; i < n; i++) {
        remaining_bt[i] = bt[i];
        first_execution[i] = false;
    }

    int t = 0;
    int completed = 0;

    while (completed < n) {
        bool executed = false;

        for (int i = 0; i < n; i++) {
            if (remaining_bt[i] > 0 && at[i] <= t) {
                executed = true;

                if (!first_execution[i]) {
                    rt[i] = t - at[i];
                    first_execution[i] = true;
                }

                if (remaining_bt[i] > quantum) {
```

```
                t += quantum;
                remaining_bt[i] -= quantum;
            } else {
                t += remaining_bt[i];
                ct[i] = t;
                tat[i] = ct[i] - at[i];
                wt[i] = tat[i] - bt[i];
                remaining_bt[i] = 0;
                completed++;
            }
        }
    }

    if (!executed) {
        int min_arrival = 1e9;
        for (int i = 0; i < n; i++) {
            if (remaining_bt[i] > 0 && at[i] > t) {
                if (at[i] < min_arrival) {
                    min_arrival = at[i];
                }
            }
        }
        if (min_arrival != 1e9) {
            t = min_arrival;
        }
    }
  }
}
```

```
int main() {
    int n, quantum;

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

    int processes[n], at[n], bt[n], ct[n], tat[n], wt[n], rt[n];
    float total_tat = 0, total_wt = 0, total_rt = 0;

    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        processes[i] = i + 1;
        printf("Process %d Arrival Time: ", i + 1);
        scanf("%d", &at[i]);
        printf("Process %d Burst Time: ", i + 1);
        scanf("%d", &bt[i]);
    }

    printf("Enter the Time Quantum: ");
    scanf("%d", &quantum);

    roundRobin(n, at, bt, quantum, ct, tat, wt, rt);

    printf("\nProcesses Arrival Time Burst Time Completion Time Turnaround Time Waiting
Time Response Time\n");
    for (int i = 0; i < n; i++) {
        total_tat += tat[i];
        total_wt += wt[i];
```

```
        total_rt += rt[i];
        printf("%8d %12d %10d %15d %15d %12d %13d\n", processes[i], at[i], bt[i], ct[i], tat[i], wt[i], rt[i]);
    }

    printf("\nAverage Turnaround Time = %.2f", total_tat / n);
    printf("\nAverage Waiting Time = %.2f", total_wt / n);
    printf("\nAverage Response Time = %.2f\n", total_rt / n);

    return 0;
}
```

Result:

```
Enter the number of processes: 6
Enter Arrival Time and Burst Time for each process:
Process 1 Arrival Time: 5
Process 1 Burst Time: 5
Process 2 Arrival Time: 4
Process 2 Burst Time: 6
Process 3 Arrival Time: 3
Process 3 Burst Time: 7
Process 4 Arrival Time: 1
Process 4 Burst Time: 9
Process 5 Arrival Time: 2
Process 5 Burst Time: 2
Process 6 Arrival Time: 6
Process 6 Burst Time: 3
Enter the Time Quantum: 4

Processes Arrival Time Burst Time Completion Time Turnaround Time Waiting Time Response Time
       1            5          5              27              22           17             5
       2            4          6              29              25           19            10
       3            3          7              32              29           22            15
       4            1          9              33              32           23             0
       5            2          2               7               5            3             3
       6            6          3              10               4            1             1

Average Turnaround Time = 19.50
Average Waiting Time = 14.17
Average Response Time = 5.67
```

Program -2

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

Code:

```
#include <stdio.h>


typedef struct {
    int id;
    int arrival_time;
    int burst_time;
    int completion_time;
    int waiting_time;
    int turnaround_time;
} Process;


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

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

int main() {
    int n, system_count = 0, user_count = 0;

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

    Process processes[n], system_processes[n], user_processes[n];

    printf("Enter process details (ID, Arrival Time, Burst Time, Type (0 for System, 1 for
User)):\n");
    for (int i = 0; i < n; i++) {
        int type;
        printf("Process %d ID: ", i + 1);
        scanf("%d", &processes[i].id);
        printf("Process %d Arrival Time: ", i + 1);
```

```
        scanf("%d", &processes[i].arrival_time);
        printf("Process %d Burst Time: ", i + 1);
        scanf("%d", &processes[i].burst_time);
        printf("Process %d Type (0 for System, 1 for User): ", i + 1);
        scanf("%d", &type);

        if (type == 0) {
            system_processes[system_count++] = processes[i];
        } else {
            user_processes[user_count++] = processes[i];
        }
    }

    sortByArrival(system_processes, system_count);
    sortByArrival(user_processes, user_count);

    printf("\nScheduling System Processes (Higher Priority - FCFS):\n");
    fcfs(system_processes, system_count);

    printf("\nScheduling User Processes (Lower Priority - FCFS):\n");
    fcfs(user_processes, user_count);

    printf("\nProcess Details After Scheduling:\n");
    printf("ID\tType\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting Time\n");

    for (int i = 0; i < system_count; i++) {
        printf("%d\tSystem\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",
```

```
            system_processes[i].id, system_processes[i].arrival_time,
system_processes[i].burst_time,
            system_processes[i].completion_time, system_processes[i].turnaround_time,
            system_processes[i].waiting_time);
    }
    for (int i = 0; i < user_count; i++) {
        printf("%d\tUser\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",
            user_processes[i].id, user_processes[i].arrival_time, user_processes[i].burst_time,
            user_processes[i].completion_time, user_processes[i].turnaround_time,
            user_processes[i].waiting_time);
    }

    return 0;
}
```

Result:

```
Process 1 Burst Time: 3
Process 1 Type (0 for System, 1 for User): 0
Process 2 ID: 2
Process 2 Arrival Time: 1
Process 2 Burst Time: 5
Process 2 Type (0 for System, 1 for User): 1
Process 3 ID: 3
Process 3 Arrival Time: 2
Process 3 Burst Time: 2
Process 3 Type (0 for System, 1 for User): 0
Process 4 ID: 4
Process 4 Arrival Time: 3
Process 4 Burst Time: 4
Process 4 Type (0 for System, 1 for User): 1

Process Details After Scheduling:
ID      Type    Arrival Time    Burst Time      Completion Time     Turnaround Time Waiting Time
1       System  0               3               3          30
3       System  2               2               5          31
2       User    1               5               6          50
4       User    3               4               10         73

Process returned 0 (0x0)   execution time : 91.391 s
Press any key to continue.
```

Program -3

Write a C program to simulate Real-Time CPU Scheduling algorithms:
a) Rate Monotonic
b) Earliest-deadline First
c) Proportional

Code:

a) Rate Monotonic

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

typedef struct {
    int id;
    int period;
    int execution;
    int deadline;
    int remaining;
} Process;

void rateMonotonicScheduling(Process processes[], int n, int total_time) {
    int time = 0;
    int completed[n];
    for (int i = 0; i < n; i++) {
        completed[i] = 0;
        processes[i].remaining = processes[i].execution;
        processes[i].deadline = processes[i].period;
    }

    int prev_process = -2;
    int start_time = 0;
```

```
printf("Time Interval\tProcess\n");

while (time < total_time) {
    int min_period = 1e9;
    int selected_process = -1;

    for (int i = 0; i < n; i++) {
        if (time >= completed[i] * processes[i].period && processes[i].remaining > 0) {
            if (processes[i].period < min_period) {
                min_period = processes[i].period;
                selected_process = i;
            }
        }
    }

    if (selected_process != prev_process) {
        if (prev_process != -2) {
            if (prev_process == -1)
                printf("[%d - %d]\tIdle\n", start_time, time);
            else
                printf("[%d - %d]\tP%d\n", start_time, time, processes[prev_process].id);
        }
        start_time = time;
        prev_process = selected_process;
    }

    if (selected_process == -1) {
```

```
            time++;
            continue;
        }

        processes[selected_process].remaining--;
        time++;

        if (processes[selected_process].remaining == 0) {
            completed[selected_process]++;
            processes[selected_process].remaining = processes[selected_process].execution;

            if (time > processes[selected_process].deadline) {
                printf(">>> P%d missed its deadline at time %d (deadline was %d)\n",
                        processes[selected_process].id, time, processes[selected_process].deadline);
            }

            processes[selected_process].deadline += processes[selected_process].period;
        }
    }

    if (prev_process != -1) {
        printf("[%d - %d]\tP%d\n", start_time, time, processes[prev_process].id);
    } else {
        printf("[%d - %d]\tIdle\n", start_time, time);
    }
}

int main() {
```

```
    int n, total_time;

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

    Process processes[n];

    printf("Enter the period and execution time for each process:\n");
    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("Process P%d Period: ", i + 1);
        scanf("%d", &processes[i].period);
        printf("Process P%d Execution Time: ", i + 1);
        scanf("%d", &processes[i].execution);
    }

    printf("Enter the total simulation time: ");
    scanf("%d", &total_time);

    rateMonotonicScheduling(processes, n, total_time);

    return 0;
}
```

Result:

```
Enter the number of processes: 2
Enter the period and execution time for each process:
Process P1 Period: 50
Process P1 Execution Time: 20
Process P2 Period: 100
Process P2 Execution Time: 35
Enter the total simulation time: 200
Time Interval   Process
[0 - 20]        P1
[20 - 50]       P2
[50 - 70]       P1
[70 - 75]       P2
[75 - 100]      Idle
[100 - 120]     P1
[120 - 150]     P2
[150 - 170]     P1
[170 - 175]     P2
[175 - 200]     Idle
```

b) Earliest-deadline First

Code:

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

typedef struct {
    int id;
    int arrival_time;
    int execution_time;
    int deadline;
    int remaining_time;
} Process;

void earliestDeadlineFirst(Process processes[], int n, int total_time) {
    int time = 0;
    int completed = 0;
    int prev_process = -2;
    int start_time = 0;

    printf("Time Interval\tProcess\n");

    while (time < total_time && completed < n) {
        int earliest_deadline = INT_MAX;
        int selected_process = -1;

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

```
        if (processes[i].arrival_time <= time && processes[i].remaining_time > 0 &&
processes[i].deadline < earliest_deadline) {
            earliest_deadline = processes[i].deadline;
            selected_process = i;
        }
    }

    if (selected_process != prev_process) {
        if (prev_process != -2) {
            if (prev_process == -1)
                printf("[%d - %d]\tIdle\n", start_time, time);
            else
                printf("[%d - %d]\tP%d\n", start_time, time, processes[prev_process].id);
        }
        start_time = time;
        prev_process = selected_process;
    }

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

    processes[selected_process].remaining_time--;
    time++;

    if (processes[selected_process].remaining_time == 0) {
        completed++;
        if (time > processes[selected_process].deadline) {
```

```
            printf(">>> P%d missed its deadline at time %d (deadline was %d)\n",
                   processes[selected_process].id, time, processes[selected_process].deadline);
        }
      }
    }

    if (prev_process != -1) {
      printf("[%d - %d]\tP%d\n", start_time, time, processes[prev_process].id);
    } else {
      printf("[%d - %d]\tIdle\n", start_time, time);
    }
}

int main() {
    int n, total_time;

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

    Process processes[n];

    printf("Enter the arrival time, execution time, and deadline for each process:\n");
    for (int i = 0; i < n; i++) {
      processes[i].id = i + 1;
      printf("Process P%d Arrival Time: ", i + 1);
      scanf("%d", &processes[i].arrival_time);
      printf("Process P%d Execution Time: ", i + 1);
      scanf("%d", &processes[i].execution_time);
```

```
        processes[i].remaining_time = processes[i].execution_time;
        printf("Process P%d Deadline: ", i + 1);
        scanf("%d", &processes[i].deadline);
    }

    printf("Enter the total simulation time: ");
    scanf("%d", &total_time);

    earliestDeadlineFirst(processes, n, total_time);

    return 0;
}
```

Result:

```
Enter the number of processes: 2
Enter the arrival time, execution time, and deadline for each process:
Process P1 Arrival Time: 0
Process P1 Execution Time: 25
Process P1 Deadline: 50
Process P2 Arrival Time: 0
Process P2 Execution Time: 35
Process P2 Deadline: 80
Enter the total simulation time: 170
Time Interval   Process
[0 - 25]        P1
[25 - 60]       P2
```

c) Proportional

Code:

```
#include <stdio.h>
typedef struct {
    int id;
    int priority;
    int burst_time;
    int allocated_time;
    int remaining_time;
} Process;

void proportionalScheduling(Process processes[], int n, int total_time) {
    int total_priority = 0;
    for (int i = 0; i < n; i++) {
        total_priority += processes[i].priority;
        processes[i].remaining_time = processes[i].burst_time;
    }

    printf("Time Interval\tProcess\n");

    int time = 0;
    while (time < total_time) {
        for (int i = 0; i < n; i++) {
            if (processes[i].remaining_time > 0) {
                int time_slice = (processes[i].priority * total_time) / total_priority;

                if (time_slice > processes[i].remaining_time) {
```

```
            time_slice = processes[i].remaining_time;
        }

        printf("[%d - %d]\tP%d\n", time, time + time_slice, processes[i].id);
        time += time_slice;
        processes[i].remaining_time -= time_slice;

        if (time >= total_time) {
            break;
        }
      }
    }
  }

  printf("Scheduling complete.\n");
}

int main() {
  int n, total_time;

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

  Process processes[n];

  printf("Enter the priority and burst time for each process:\n");
  for (int i = 0; i < n; i++) {
    processes[i].id = i + 1;
```

```
        printf("Process P%d Priority: ", i + 1);
        scanf("%d", &processes[i].priority);
        printf("Process P%d Burst Time: ", i + 1);
        scanf("%d", &processes[i].burst_time);
    }

    printf("Enter the total simulation time: ");
    scanf("%d", &total_time);

    proportionalScheduling(processes, n, total_time);

    return 0;
}
```

Result:

```
Enter the number of processes: 3
Enter the priority and burst time for each process:
Process P1 Priority: 3
Process P1 Burst Time: 6
Process P2 Priority: 2
Process P2 Burst Time: 4
Process P3 Priority: 1
Process P3 Burst Time: 2
Enter the total simulation time: 12
Time Interval   Process
[0 - 6] P1
[6 - 10]        P2
[10 - 12]       P3
Scheduling complete.
```

Program -4

Write a C program to simulate:

a) Producer-Consumer problem using semaphores.

b) Dining-Philosopher's problem

a) Producer-Consumer problem using semaphores.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#define BUFFER_SIZE 5
#define MAX_ITEMS 5
int buffer[BUFFER_SIZE];
int count = 0;
int produced = 0;
int consumed = 0;
pthread_mutex_t mutex;
pthread_cond_t not_full;
pthread_cond_t not_empty;

void* producer(void* arg) {
    int item = 0;
    while (1) {
        pthread_mutex_lock(&mutex);
        if (produced >= MAX_ITEMS) {
            pthread_mutex_unlock(&mutex);
            break;
        }
```

```
        while (count == BUFFER_SIZE) {
            pthread_cond_wait(&not_full, &mutex);
        }
        item++;
        buffer[count++] = item;
        produced++;
        printf("Producer produced: %d (Total produced: %d)\n", item, produced);
        pthread_cond_signal(&not_empty);
        pthread_mutex_unlock(&mutex);
        usleep(100000);
    }
    return NULL;
}

void* consumer(void* arg) {
    while (1) {
        pthread_mutex_lock(&mutex);
        if (consumed >= MAX_ITEMS) {
            pthread_mutex_unlock(&mutex);
            break;
        }
        while (count == 0) {
            pthread_cond_wait(&not_empty, &mutex);
        }
        int item = buffer[--count];
        consumed++;
        printf("Consumer consumed: %d (Total consumed: %d)\n", item, consumed);
        pthread_cond_signal(&not_full);
```

```
        pthread_mutex_unlock(&mutex);
        usleep(200000);
    }
    return NULL;
}

int main() {
    pthread_t prod_thread, cons_thread;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&not_full, NULL);
    pthread_cond_init(&not_empty, NULL);

    pthread_create(&prod_thread, NULL, producer, NULL);
    pthread_create(&cons_thread, NULL, consumer, NULL);

    pthread_join(prod_thread, NULL);
    pthread_join(cons_thread, NULL);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&not_full);
    pthread_cond_destroy(&not_empty);

    printf("\nAll %d items produced and consumed. Program exiting.\n", MAX_ITEMS);
    return 0;
}
```

Result:

```
Producer produced: 1 (Total produced: 1)
Consumer consumed: 1 (Total consumed: 1)
Producer produced: 2 (Total produced: 2)
Consumer consumed: 2 (Total consumed: 2)
Producer produced: 3 (Total produced: 3)
Producer produced: 4 (Total produced: 4)
Consumer consumed: 4 (Total consumed: 3)
Producer produced: 5 (Total produced: 5)
Consumer consumed: 5 (Total consumed: 4)
Consumer consumed: 3 (Total consumed: 5)

All 5 items produced and consumed. Program exiting.

Process returned 0 (0x0)   execution time : 10.792 s
Press any key to continue.
```

b) Dining-Philosopher's problem

Code:

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#define MAX 5
int chopstick[MAX] = {1, 1, 1, 1, 1};
int mutex = 1;
int philosopher_id = 0;

void Wait(int *s) {
    while (*s <= 0);
    (*s)--;
}

void Signal(int *s) {
    (*s)++;
}
```

```
void* philosopher(void* arg) {
    int id;
    Wait(&mutex);
    id = philosopher_id++;
    Signal(&mutex);
    int left = id;
    int right = (id + 1) % MAX;
    while (1) {
        printf("Philosopher %d is thinking.\n", id);
        sleep(1);
        Wait(&mutex);
        if (chopstick[left] && chopstick[right]) {
            chopstick[left] = chopstick[right] = 0;
            printf("Philosopher %d picked up chopsticks %d and %d and is eating.\n", id, left, right);
            Signal(&mutex);
            sleep(2);
            Wait(&mutex);
            chopstick[left] = chopstick[right] = 1;
            printf("Philosopher %d put down chopsticks %d and %d.\n", id, left, right);
            Signal(&mutex);
        } else {
            Signal(&mutex);
        }
    }
}

void main() {
```

```
    pthread_t p0, p1, p2, p3, p4;

    pthread_create(&p0, NULL, philosopher, NULL);
    pthread_create(&p1, NULL, philosopher, NULL);
    pthread_create(&p2, NULL, philosopher, NULL);
    pthread_create(&p3, NULL, philosopher, NULL);
    pthread_create(&p4, NULL, philosopher, NULL);

    pthread_join(p0, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    pthread_join(p3, NULL);
    pthread_join(p4, NULL);
}
```

Result:

```
Philosopher 0 is thinking.
Philosopher 1 is thinking.
Philosopher 2 is thinking.
Philosopher 3 is thinking.
Philosopher 4 is thinking.
Philosopher 4 picked up chopsticks 4 and 0 and is eating.
Philosopher 0 is thinking.
Philosopher 1 picked up chopsticks 1 and 2 and is eating.
Philosopher 3 is thinking.
Philosopher 2 is thinking.
Philosopher 0 is thinking.
Philosopher 3 is thinking.
Philosopher 2 is thinking.
Philosopher 4 put down chopsticks 4 and 0.
Philosopher 4 is thinking.
Philosopher 1 put down chopsticks 1 and 2.
Philosopher 3 is thinking.
Philosopher 1 is thinking.
Philosopher 2 picked up chopsticks 2 and 3 and is eating.
Philosopher 0 picked up chopsticks 0 and 1 and is eating.
Philosopher 4 is thinking.
```

Program -5

Write a C program to simulate

a) Bankers' algorithm for the purpose of deadlock avoidance.

b) Deadlock Detection

a) Bankers' algorithm for the purpose of deadlock avoidance

Code:

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_P 10
#define MAX_R 10
int n, m;
int Allocation[MAX_P][MAX_R];
int Maximum[MAX_P][MAX_R];
int Need[MAX_P][MAX_R];
int Available[MAX_R];
bool Finish[MAX_P];
int SafeSequence[MAX_P];

bool isSafeState();
bool canExecute(int process, int work[]);

int main() {
    input();
    calculateNeed();
    if (isSafeState()) {
        printf("System is in a safe state.\n");
        printSafeSequence();
    } else {
```

```
        printf("System is not in a safe state. Deadlock may occur.\n");
    }

    return 0;
}

void input() {
    printf("Enter number of processes: ");
    scanf("%d", &n);
    printf("Enter number of resources: ");
    scanf("%d", &m);
    printf("Enter Allocation Matrix (%d x %d):\n", n, m);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            scanf("%d", &Allocation[i][j]);

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

    printf("Enter Available Resources (%d):\n", m);
    for (int i = 0; i < m; i++)
        scanf("%d", &Available[i]);
    for (int i = 0; i < n; i++)
        Finish[i] = false;
}
```

```
void calculateNeed() {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            Need[i][j] = Maximum[i][j] - Allocation[i][j];
}

bool canExecute(int process, int work[]) {
    for (int j = 0; j < m; j++) {
        if (Need[process][j] > work[j])
            return false;
    }
    return true;
}

bool isSafeState() {
    int work[MAX_R];
    for (int i = 0; i < m; i++)
        work[i] = Available[i];
    int count = 0;
    while (count < n) {
        bool found = false;
        for (int i = 0; i < n; i++) {
            if (!Finish[i] && canExecute(i, work)) {
                for (int j = 0; j < m; j++)
                    work[j] += Allocation[i][j];
                Finish[i] = true;
                SafeSequence[count++] = i;
                found = true;
```

```
            }
        }
        if (!found)
            return false;
    }return true;
}

void printSafeSequence() {
    printf("Safe sequence is: ");
    for (int i = 0; i < n; i++) {
        printf("P%d", SafeSequence[i]);
        if (i < n - 1)
            printf(" -> ");
    }
    printf("\n");
}
```

Result:

```
Enter number of processes: 5
Enter number of resources: 3
Enter Allocation Matrix (5 x 3):
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter Maximum Matrix (5 x 3):
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter Available Resources (3):
3 3 2
System is in a safe state.
Safe sequence is: P1 -> P3 -> P4 -> P0 -> P2

Process returned 0 (0x0)   execution time : 41.430 s
Press any key to continue.
```

b) Deadlock Detection

Code:

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

#define MAX_P 10
#define MAX_R 10

int n, m;
int Allocation[MAX_P][MAX_R];
int Request[MAX_P][MAX_R];
int Available[MAX_R];
bool Finish[MAX_P];

int main() {
   input();
   detectDeadlock();
   return 0;
}

void input() {
   printf("Enter number of processes: ");
   scanf("%d", &n);
   printf("Enter number of resources: ");
   scanf("%d", &m);

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

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

   printf("Enter Available Resources (%d):\n", m);
   for (int i = 0; i < m; i++)
      scanf("%d", &Available[i]);

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

```
}

bool canExecute(int process) {
    for (int j = 0; j < m; j++) {
        if (Request[process][j] > Available[j])
            return false;
    }
    return true;
}

void detectDeadlock() {
    int count = 0;

    while (count < n) {
        bool found = false;

        for (int i = 0; i < n; i++) {
            if (!Finish[i] && canExecute(i)) {
                for (int j = 0; j < m; j++)
                    Available[j] += Allocation[i][j];

                Finish[i] = true;
                found = true;
                count++;
            }
        }

        if (!found)
            break;
    }

    printDeadlockProcesses();
}

void printDeadlockProcesses() {
    bool deadlock = false;
    for (int i = 0; i < n; i++) {
        if (!Finish[i]) {
            deadlock = true;
            printf("Process %d is in deadlock.\n", i);
        }
    }

    if (!deadlock)
```

```
    printf("No deadlock detected. All processes can complete.\n");
}
```

Result:

```
Enter number of processes: 5
Enter number of resources: 3
Enter Allocation Matrix (5 x 3):
0 1 0
2 0 0
3 0 3
2 1 1
0 0 2
Enter Request Matrix (5 x 3):
0 0 0
2 0 2
0 0 0
1 0 0
0 0 2
Enter Available Resources (3):
0 0 0
No deadlock detected. All processes can complete.

Process returned 0 (0x0)   execution time : 60.914 s
Press any key to continue.
```

Program -6

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

a) Worst-fit

b) Best-fit

c) First-fit

Code:

```
#include <stdio.h>
#define MAX_BLOCKS 10
#define MAX_PROCESSES 10
typedef struct {
    int size;
    int originalIndex;
} Block;

void printAllocation(const char *strategy, int allocation[], int processSize[], int processes) {
    printf("\n%s Allocation:\n", strategy);
    for (int i = 0; i < processes; i++) {
        if (allocation[i] != -1)
            printf("Process %d of size %d -> Block %d\n", i + 1, processSize[i], allocation[i] + 1);
        else
            printf("Process %d of size %d -> Not Allocated\n", i + 1, processSize[i]);
    }
}

void firstFit(int blockSize[], int blocks, int processSize[], int processes) {
    int allocation[MAX_PROCESSES];
    for (int i = 0; i < processes; i++) {
```

```
        allocation[i] = -1;
        for (int j = 0; j < blocks; j++) {
            if (blockSize[j] >= processSize[i]) {
                allocation[i] = j;
                blockSize[j] -= processSize[i];
                break;
            }
        }
    }
printAllocation("First Fit", allocation, processSize, processes);
}

void sortBlocks(Block arr[], int n, int ascending) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if ((ascending && arr[j].size > arr[j + 1].size) ||
                (!ascending && arr[j].size < arr[j + 1].size)) {
                Block temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

void bestFit(int blockSize[], int blocks, int processSize[], int processes) {
    int allocation[MAX_PROCESSES];
    Block sortedBlocks[MAX_BLOCKS];
```

```
    for (int i = 0; i < blocks; i++) {
        sortedBlocks[i].size = blockSize[i];
        sortedBlocks[i].originalIndex = i;
    }
  sortBlocks(sortedBlocks, blocks, 1);
 for (int i = 0; i < processes; i++) {
        allocation[i] = -1;
        for (int j = 0; j < blocks; j++) {
            if (sortedBlocks[j].size >= processSize[i]) {
                allocation[i] = sortedBlocks[j].originalIndex;
                sortedBlocks[j].size -= processSize[i];
                break;
            }
        }
    }
    printAllocation("Best Fit", allocation, processSize, processes);
}

void worstFit(int blockSize[], int blocks, int processSize[], int processes) {
    int allocation[MAX_PROCESSES];
    Block sortedBlocks[MAX_BLOCKS];
    for (int i = 0; i < blocks; i++) {
        sortedBlocks[i].size = blockSize[i];
        sortedBlocks[i].originalIndex = i;
    }

    sortBlocks(sortedBlocks, blocks, 0);
```

```
    for (int i = 0; i < processes; i++) {
        allocation[i] = -1;
        for (int j = 0; j < blocks; j++) {
            if (sortedBlocks[j].size >= processSize[i]) {
                allocation[i] = sortedBlocks[j].originalIndex;
                sortedBlocks[j].size -= processSize[i];
                break;
            }
        }
    }
    printAllocation("Worst Fit", allocation, processSize, processes);
}

int main() {
    int blockSize[MAX_BLOCKS], processSize[MAX_PROCESSES];
    int blocks, processes;
    printf("Enter number of memory blocks: ");
    scanf("%d", &blocks);
    printf("Enter size of each block:\n");
    for (int i = 0; i < blocks; i++) {
        printf("Block %d: ", i + 1);
        scanf("%d", &blockSize[i]);
    }
    printf("Enter number of processes: ");
    scanf("%d", &processes);
    printf("Enter size of each process:\n");
    for (int i = 0; i < processes; i++) {
```

```
        printf("Process %d: ", i + 1);
        scanf("%d", &processSize[i]);
    }
    int blockSize1[MAX_BLOCKS], blockSize2[MAX_BLOCKS], blockSize3[MAX_BLOCKS];
    for (int i = 0; i < blocks; i++) {
        blockSize1[i] = blockSize[i];
        blockSize2[i] = blockSize[i];
        blockSize3[i] = blockSize[i];
    }
    firstFit(blockSize1, blocks, processSize, processes);
    bestFit(blockSize2, blocks, processSize, processes);
    worstFit(blockSize3, blocks, processSize, processes);
    return 0;
}
```

Result:

```
Block 4: 300
Block 5: 600
Enter number of processes: 4
Enter size of each process:
Process 1: 212
Process 2: 417
Process 3: 112
Process 4: 426

First Fit Allocation:
Process 1 of size 212 -> Block 2
Process 2 of size 417 -> Block 5
Process 3 of size 112 -> Block 2
Process 4 of size 426 -> Not Allocated

Best Fit Allocation:
Process 1 of size 212 -> Block 4
Process 2 of size 417 -> Block 2
Process 3 of size 112 -> Block 3
Process 4 of size 426 -> Block 5

Worst Fit Allocation:
Process 1 of size 212 -> Block 5
Process 2 of size 417 -> Block 2
Process 3 of size 112 -> Block 5
Process 4 of size 426 -> Not Allocated

Process returned 0 (0x0)   execution time : 32.337 s
Press any key to continue.
```

Program -7

Write a C program to simulate page replacement algorithms.

a) FIFO

b) LRU

c) Optimal

Code:

```
#include <stdio.h>
#include <limits.h>
#define MAX_FRAMES 10
#define MAX_PAGES 50

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

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

void fifo(int pages[], int nPages, int nFrames) {
    int frames[MAX_FRAMES], ptr = 0, faults = 0;
    for (int i = 0; i < nFrames; i++) frames[i] = -1;
    printf("\nFIFO Page Replacement:\n");
    for (int i = 0; i < nPages; i++) {
```

```
        printf("Page %2d: ", pages[i]);
        if (findPage(frames, nFrames, pages[i]) == -1) {
            frames[ptr] = pages[i];
            ptr = (ptr + 1) % nFrames;
            faults++;
        }
        printFrames(frames, nFrames);
    }
    printf("Total Page Faults: %d\n", faults);
}

void lru(int pages[], int nPages, int nFrames) {
    int frames[MAX_FRAMES], counter[MAX_FRAMES], faults = 0, time = 0;
    for (int i = 0; i < nFrames; i++) {
        frames[i] = -1;
        counter[i] = 0;
    }
    printf("\nLRU Page Replacement:\n");
    for (int i = 0; i < nPages; i++) {
        printf("Page %2d: ", pages[i]);
        int pos = findPage(frames, nFrames, pages[i]);
        if (pos == -1) {
            int lru = 0;
            for (int j = 1; j < nFrames; j++)
                if (counter[j] < counter[lru]) lru = j;
            frames[lru] = pages[i];
            counter[lru] = ++time;
            faults++;
```

```
        } else {
            counter[pos] = ++time;
        }
        printFrames(frames, nFrames);
    }
    printf("Total Page Faults: %d\n", faults);
}

void optimal(int pages[], int nPages, int nFrames) {
    int frames[MAX_FRAMES], faults = 0;
    for (int i = 0; i < nFrames; i++) frames[i] = -1;
    printf("\nOptimal Page Replacement:\n");
    for (int i = 0; i < nPages; i++) {
        printf("Page %2d: ", pages[i]);
        if (findPage(frames, nFrames, pages[i]) == -1) {
            int replace = 0, farthest = i;
            for (int j = 0; j < nFrames; j++) {
                int k;
                for (k = i + 1; k < nPages; k++)
                    if (frames[j] == pages[k]) break;
                if (k > farthest) {
                    farthest = k;
                    replace = j;
                }
                if (k == nPages) {
                    replace = j;
                    break;
                }
```

```
            }
            frames[replace] = pages[i];
            faults++;
        }
        printFrames(frames, nFrames);
    }
    printf("Total Page Faults: %d\n", faults);
}

int main() {
    int pages[MAX_PAGES], nPages, nFrames;
    printf("Enter number of pages: ");
    scanf("%d", &nPages);
    printf("Enter page references: ");
    for (int i = 0; i < nPages; i++) scanf("%d", &pages[i]);
    printf("Enter number of frames: ");
    scanf("%d", &nFrames);
    fifo(pages, nPages, nFrames);
    lru(pages, nPages, nFrames);
    optimal(pages, nPages, nFrames);
    return 0;
}
```

Result:

```
Enter number of pages: 8
Enter page references: 1 2 3 2 4 1 5 2
Enter number of frames: 3

FIFO Page Replacement:
Page  1:  1  -  -
Page  2:  1  2  -
Page  3:  1  2  3
Page  2:  1  2  3
Page  4:  4  2  3
Page  1:  4  1  3
Page  5:  4  1  5
Page  2:  2  1  5
Total Page Faults: 7

LRU Page Replacement:
Page  1:  1  -  -
Page  2:  1  2  -
Page  3:  1  2  3
Page  2:  1  2  3
Page  4:  4  2  3
Page  1:  4  2  1
Page  5:  4  5  1
Page  2:  2  5  1
Total Page Faults: 7

Optimal Page Replacement:
Page  1:  1  -  -
Page  2:  1  2  -
Page  3:  1  2  3
Page  2:  1  2  3
Page  4:  1  2  4
Page  1:  1  2  4
Page  5:  5  2  4
Page  2:  5  2  4
Total Page Faults: 5

Process returned 0 (0x0)   execution time : 52.622 s
Press any key to continue.
```

Program -8

Write a c program to stimulate the following file allocation strategies

a) sequential

b) indexed

c) linked

Code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_BLOCKS 10

void sequentialAllocation(int blocks[], int total) {
    printf("Sequential: ");
    for (int i = 0; i < total; i++) printf("%d -> ", blocks[i]);
    printf("End\n");
}

void indexedAllocation(int index, int blocks[], int total) {
    printf("Indexed: Index Block %d -> [ ", index);
    for (int i = 0; i < total; i++) printf("%d ", blocks[i]);
    printf("]\n");
}

void linkedAllocation(int blocks[], int total) {
    printf("Linked: ");
    for (int i = 0; i < total; i++) printf("%d -> ", blocks[i]);
    printf("End\n");
}
```

```
int main() {
    int blocks[MAX_BLOCKS], total, choice, indexBlock;

    printf("Enter number of blocks (<=10): ");
    scanf("%d", &total);
    printf("Enter block numbers: ");
    for (int i = 0; i < total; i++) scanf("%d", &blocks[i]);
    printf("Choose allocation: 1. Sequential  2. Indexed  3. Linked\n");
    scanf("%d", &choice);

    if (choice == 2) {
        printf("Enter index block number: ");
        scanf("%d", &indexBlock);
    }

    if (choice == 1) sequentialAllocation(blocks, total);
    else if (choice == 2) indexedAllocation(indexBlock, blocks, total);
    else if (choice == 3) linkedAllocation(blocks, total);
    else printf("Invalid choice!\n");

    return 0;
}
```

Result:

```
Enter number of blocks (<=10): 6
Enter block numbers: 4
3
2
1
3
0
Choose allocation: 1. Sequential  2. Indexed  3. Linked
1
Sequential: 4 -> 3 -> 2 -> 1 -> 3 -> 0 -> End
```