VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS

Submitted by

BHUVANA M (1BM22CS071)

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

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Apr-2024 to Aug-2024

B. M. S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "OPERATING SYSTEMS – 23CS4PCOPS" carried out by **BHUVANA M (1BM22CS071),** who is bonafide student of **B. M. S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2024. The Lab report has been approved as it satisfies the academic requirements in respect of a **OPERATING SYSTEMS - (23CS4PCOPS)** work prescribed for the said degree.

Name of the Lab-Incharge
Designation
Department of CSE
BMSCE, Bengaluru

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

Index Sheet

10	Write a C program to simulate page replacement algorithms a. FIFO b. LRU c. Optimal	59

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.

1. Write a C program to simulate the following non pre-emptive CPU scheduling algorithm to find turnaround time and waiting time.
   a. FCFS
   b. SJF(Pre-emptive & Non Pre-emptive)

Ans:

a. FCFS:

```
#include <stdio.h>

#define MAX_PROCESSES 10

// Structure to represent a process
struct Process {
    int process_id;
    int arrival_time;
    int burst_time;
    int completion_time;
    int waiting_time;
    int turnaround_time;
    int response_time;
};

// Function to calculate waiting time, turnaround time, completion time, and response time for
FCFS
void fcfs(struct Process processes[], int n) {
    int current_time = 0;

    // Sort processes by arrival time
    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) {
                struct Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
```

```
    }

    // Calculate completion time, waiting time, turnaround time, and response time for each
process
    for (int i = 0; i < n; i++) {
        if (processes[i].arrival_time > current_time) {
            current_time = processes[i].arrival_time;
        }
        processes[i].response_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;
    }

    // Calculate average waiting time, turnaround time, completion time, and response time
    float total_waiting_time = 0;
    float total_turnaround_time = 0;
    float total_completion_time = 0;
    float total_response_time = 0;

    for (int i = 0; i < n; i++) {
        total_waiting_time += processes[i].waiting_time;
        total_turnaround_time += processes[i].turnaround_time;
        total_completion_time += processes[i].completion_time;
        total_response_time += processes[i].response_time;
    }

    float avg_waiting_time = total_waiting_time / n;
    float avg_turnaround_time = total_turnaround_time / n;
    float avg_completion_time = total_completion_time / n;
    float avg_response_time = total_response_time / n;

    // Display results
    printf("\nProcess\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting
Time\tResponse Time\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t\t%d\t\t%d\t\t%d\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].completion_time,
               processes[i].turnaround_time, processes[i].waiting_time,
               processes[i].response_time);
    }
```

```
    printf("\nAverage Waiting Time: %.2f\n", avg_waiting_time);
    printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);
    printf("Average Completion Time: %.2f\n", avg_completion_time);
    printf("Average Response Time: %.2f\n", avg_response_time);
}

int main() {
    int n;
    struct Process processes[MAX_PROCESSES];

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

    if (n <= 0 || n > MAX_PROCESSES) {
        printf("Invalid number of processes.\n");
        return 1; // Return non-zero value to indicate error
    }

    // Input arrival time and burst time for each process
    printf("Enter arrival time and burst time for each process (separated by spaces):\n");
    for (int i = 0; i < n; i++) {
        processes[i].process_id = i + 1;
        printf("Process %d: ", i + 1);
        scanf("%d %d", &processes[i].arrival_time, &processes[i].burst_time);
    }

    // Call FCFS scheduling function
    fcfs(processes, n);
```

```
bhu@Bhuvanas-MacBook-Pro OS LAB % cd "/Users/bhu/Documents/OS LAB/" && gcc fcfs.c -o fcfs && "/Users/bhu/Documents/OS LAB/"fcfs
Enter the number of processes (max 10): 4
Enter arrival time and burst time for each process (separated by spaces):
Process 1: 0 7
Process 2: 8 3
Process 3: 3 4
Process 4: 5 6

Process Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time    Response Time
P1              0               7               7               7               0               0
P3              3               4               11              8               4               4
P4              5               6               17              12              6               6
P2              8               3               20              12              9               9

Average Waiting Time: 4.75
Average Turnaround Time: 9.75
Average Completion Time: 13.75
Average Response Time: 4.75
bhu@Bhuvanas-MacBook-Pro OS LAB % 
```

b. SJF (Pre-Emptive):

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

// Structure to represent a process
struct Process {
    int pid;
    int arrival_time;
    int burst_time;
    int remaining_time; // Remaining burst time for preemptive SJF
    int completion_time;
    int turnaround_time;
    int waiting_time;
    int start_time; // Time when the process starts execution for the first time
    bool started; // To check if the process has started
};

// Function to find the process with the shortest remaining burst time among the arrived
processes
int findShortestJob(struct Process processes[], int n, int current_time) {
    int shortest_job_index = -1;
    int shortest_job = INT_MAX;

    for (int i = 0; i < n; i++) {
        if (processes[i].arrival_time <= current_time && processes[i].remaining_time > 0 &&
processes[i].remaining_time < shortest_job) {
            shortest_job_index = i;
            shortest_job = processes[i].remaining_time;
        }
    }

    return shortest_job_index;
}

// Function to simulate the SJF preemptive scheduling algorithm
void SJF(struct Process processes[], int n) {
    int current_time = 0;
    int completed = 0;
    float total_waiting_time = 0;
    float total_turnaround_time = 0;
    float total_completion_time = 0;
```

```
    while (completed < n) {
        int shortest_job_index = findShortestJob(processes, n, current_time);

        if (shortest_job_index == -1) {
            current_time++; // Move to the next time unit
        } else {
            // Update the start time if the process starts for the first time
            if (!processes[shortest_job_index].started) {
                processes[shortest_job_index].start_time = current_time;
                processes[shortest_job_index].started = true;
            }

            // Execute the shortest job for one time unit
            processes[shortest_job_index].remaining_time--;
            current_time++;

            if (processes[shortest_job_index].remaining_time == 0) {
                // Update completion time and calculate turnaround time and waiting time
                processes[shortest_job_index].completion_time = current_time;
                processes[shortest_job_index].turnaround_time =
processes[shortest_job_index].completion_time - processes[shortest_job_index].arrival_time;
                processes[shortest_job_index].waiting_time =
processes[shortest_job_index].turnaround_time - processes[shortest_job_index].burst_time;

                // Accumulate total times
                total_waiting_time += processes[shortest_job_index].waiting_time;
                total_turnaround_time += processes[shortest_job_index].turnaround_time;
                total_completion_time += processes[shortest_job_index].completion_time;

                completed++;
            }
        }
    }

    // Display process details
    printf("\nProcess\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting
Time\tResponse Time\n");
    for (int i = 0; i < n; i++) {
        int response_time = processes[i].start_time - processes[i].arrival_time;
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid,
processes[i].arrival_time, processes[i].burst_time, processes[i].completion_time,
processes[i].turnaround_time, processes[i].waiting_time, response_time);
```

```c
    }

    // Calculate and display averages
    printf("\nAverage Completion Time: %.2f\n", total_completion_time / n);
    printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);
    printf("Average Waiting Time: %.2f\n", total_waiting_time / n);
}

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

    struct Process processes[n];
    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Burst Time: ");
        scanf("%d", &processes[i].burst_time);
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].pid = i + 1;
        processes[i].started = false;
    }

    SJF(processes, n);

    return 0;
}
```

Output:

```
Enter the total number of processes: 4
Enter Arrival Time and Burst Time for each process:
Process 1:
Arrival Time: 0
Burst Time: 7
Process 2:
Arrival Time: 8
Burst Time: 3
Process 3:
Arrival Time: 3
Burst Time: 2
Process 4:
Arrival Time: 5
Burst Time: 6

Process Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time    Response Time
1       0               7               9               9               2               0
2       8               3               12              4               1               1
3       3               2               5               2               0               0
4       5               6               18              13              7               7

Average Completion Time: 11.00
Average Turnaround Time: 7.00
Average Waiting Time: 2.50
```

b. SJF(Non Pre-Emptive):

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

struct Process {
    int pid;
    int arrival_time;
    int burst_time;
    int completion_time;
    int waiting_time;
    int turnaround_time;
};

void SJF(struct Process processes[], int n) {
    int current_time = 0;
    int completed = 0;
    int min_burst_index;

    while (completed < n) {
        min_burst_index = -1;
        int min_burst = INT_MAX; // Initialize min_burst to maximum possible value

        // Find the process with the minimum burst time among the arrived processes
        for (int i = 0; i < n; i++) {
            if (processes[i].arrival_time <= current_time && processes[i].completion_time == 0 &&
processes[i].burst_time < min_burst) {
                min_burst_index = i;
                min_burst = processes[i].burst_time;
            }
        }

        if (min_burst_index == -1) {
            current_time++; // Move to the next time unit
        } else {
            // Execute the process with the minimum burst time
            processes[min_burst_index].completion_time = current_time +
processes[min_burst_index].burst_time;
            processes[min_burst_index].turnaround_time =
processes[min_burst_index].completion_time - processes[min_burst_index].arrival_time;
            processes[min_burst_index].waiting_time =
processes[min_burst_index].turnaround_time - processes[min_burst_index].burst_time;
            current_time = processes[min_burst_index].completion_time;
```

```
            completed++;
        }
    }
}

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

    struct Process processes[n];
    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Burst Time: ");
        scanf("%d", &processes[i].burst_time);
        processes[i].pid = i + 1;
        processes[i].completion_time = 0; // Initialize completion time to 0
    }

    SJF(processes, n);

    // Print results
    printf("\nProcess\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting
Time\n");
    float total_completion_time = 0, total_turnaround_time = 0, total_waiting_time = 0;
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].arrival_time,
processes[i].burst_time, processes[i].completion_time, processes[i].turnaround_time,
processes[i].waiting_time);
        total_completion_time += processes[i].completion_time;
        total_turnaround_time += processes[i].turnaround_time;
        total_waiting_time += processes[i].waiting_time;
    }

    // Calculate and display averages
    printf("\nAverage Completion Time: %.2f\n", total_completion_time / n);
    printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);
    printf("Average Waiting Time: %.2f\n", total_waiting_time / n);

    return 0;
```

```
}
```

Output:

```
Enter the total number of processes: 4
Enter Arrival Time and Burst Time for each process:
Process 1:
Arrival Time: 0
Burst Time: 7
Process 2:
Arrival Time: 8
Burst Time: 3
Process 3:
Arrival Time: 3
Burst Time: 4
Process 4:
Arrival Time: 5
Burst Time: 6

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

Average Completion Time: 13.00
Average Turnaround Time: 9.00
Average Waiting Time: 4.00
```

2. Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time.
a. Priority (Pre-Emptive & Non Pre-Emptive)
b.Round Robin (Experiment with different quantum sizes for RR algorithm)

Ans:

a. Priority (Pre-Emptive) :

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

struct Process {
    int pid;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int priority;
    int completion_time;
    int waiting_time;
    int turnaround_time;
    int start_time;
    bool started;
};

int findHighestPriorityProcess(struct Process processes[], int n, int current_time) {
    int highest_priority_index = -1;
    int highest_priority = INT_MAX;

    for (int i = 0; i < n; i++) {
        if (processes[i].arrival_time <= current_time && processes[i].remaining_time > 0 &&
processes[i].priority < highest_priority) {
            highest_priority = processes[i].priority;
            highest_priority_index = i;
        }
    }

    return highest_priority_index;
}

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

```
    int completed = 0;

    while (completed < n) {
        int highest_priority_index = findHighestPriorityProcess(processes, n, current_time);

        if (highest_priority_index == -1) {
            current_time++;
        } else {
            if (!processes[highest_priority_index].started) {
                processes[highest_priority_index].start_time = current_time;
                processes[highest_priority_index].started = true;
            }

            processes[highest_priority_index].remaining_time--;
            current_time++;

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

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

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

```
        printf("Priority: ");
        scanf("%d", &processes[i].priority);
        processes[i].pid = i + 1;
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].started = false;
    }

    priorityPreemptiveScheduling(processes, n);

    printf("\nProcess\tArrival Time\tBurst Time\tPriority\tCompletion Time\tTurnaround Time\tWaiting Time\n");
    float total_completion_time = 0, total_turnaround_time = 0, total_waiting_time = 0;
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].arrival_time, processes[i].burst_time, processes[i].priority, processes[i].completion_time, processes[i].turnaround_time, processes[i].waiting_time);
        total_completion_time += processes[i].completion_time;
        total_turnaround_time += processes[i].turnaround_time;
        total_waiting_time += processes[i].waiting_time;
    }

    // Calculate and display averages
    printf("\nAverage Completion Time: %.2f\n", total_completion_time / n);
    printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);
    printf("Average Waiting Time: %.2f\n", total_waiting_time / n);

    return 0;
}
```

<u>Output:</u>

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

Process Arrival Time    Burst Time      Priority        Completion Time Turnaround Time Waiting Time
1       0               5               4               13              13              8
2       2               4               2               6               4               0
3       2               2               6               15              13              11
4       4               4               3               10              6               2

Average Completion Time: 11.00
Average Turnaround Time: 9.00
Average Waiting Time: 5.25
```

a. Priority (Non Pre-Emptive):

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

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

void priorityNonPreemptiveScheduling(struct Process processes[], int n) {
    int completed = 0;
    int current_time = 0;
    int min_priority_index;

    while (completed < n) {
        min_priority_index = -1;
        int min_priority = INT_MAX;

        // Find the process with the highest priority (smallest priority number) that has arrived and
is not yet completed
        for (int i = 0; i < n; i++) {
            if (processes[i].arrival_time <= current_time && processes[i].completion_time == 0 &&
processes[i].priority < min_priority) {
                min_priority = processes[i].priority;
                min_priority_index = i;
            }
        }

        if (min_priority_index == -1) {
            current_time++;
        } else {
            // Execute the selected process
            processes[min_priority_index].completion_time = current_time +
processes[min_priority_index].burst_time;
            processes[min_priority_index].turnaround_time =
processes[min_priority_index].completion_time - processes[min_priority_index].arrival_time;
```

```
            processes[min_priority_index].waiting_time =
processes[min_priority_index].turnaround_time - processes[min_priority_index].burst_time;
            current_time = processes[min_priority_index].completion_time;
            completed++;
        }
    }
}

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

    struct Process processes[n];
    printf("Enter Arrival Time, Burst Time, and Priority for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Burst Time: ");
        scanf("%d", &processes[i].burst_time);
        printf("Priority: ");
        scanf("%d", &processes[i].priority);
        processes[i].pid = i + 1;
        processes[i].completion_time = 0; // Initialize completion time to 0
    }

    priorityNonPreemptiveScheduling(processes, n);

    printf("\nProcess\tArrival Time\tBurst Time\tPriority\tCompletion Time\tTurnaround
Time\tWaiting Time\n");
    float total_completion_time = 0, total_turnaround_time = 0, total_waiting_time = 0;
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid,
processes[i].arrival_time, processes[i].burst_time, processes[i].priority,
processes[i].completion_time, processes[i].turnaround_time, processes[i].waiting_time);
        total_completion_time += processes[i].completion_time;
        total_turnaround_time += processes[i].turnaround_time;
        total_waiting_time += processes[i].waiting_time;
    }

    // Calculate and display averages
    printf("\nAverage Completion Time: %.2f\n", total_completion_time / n);
```

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

    return 0;
}
```

Output:

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

Process Arrival Time    Burst Time      Priority        Completion Time Turnaround Time Waiting Time
1       2               4               2               22              20              16
2       4               7               1               12              8               1
3       5               2               3               24              19              17
4       1               4               2               5               4               0
5       7               6               1               18              11              5

Average Completion Time: 16.20
Average Turnaround Time: 12.40
Average Waiting Time: 7.80
```

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

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

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

void round_robin(struct Process proc[], int n, int quantum) {
    int current_time = 0;
    int completed_processes = 0;

    while (completed_processes < n) {
        bool process_found = false;

        for (int i = 0; i < n; i++) {
            if (proc[i].remaining_time > 0 && proc[i].arrival_time <= current_time) {
                process_found = true;

                if (proc[i].remaining_time > quantum) {
                    current_time += quantum;
                    proc[i].remaining_time -= quantum;
                } else {
                    current_time += proc[i].remaining_time;
                    proc[i].completion_time = current_time;
                    proc[i].turnaround_time = proc[i].completion_time - proc[i].arrival_time;
                    proc[i].waiting_time = proc[i].turnaround_time - proc[i].burst_time;
                    proc[i].remaining_time = 0;
                    completed_processes++;
                }
            }
        }

        if (!process_found) {
            current_time++;
```

```
        }
    }

    // Print the results
    printf("\nPID\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting
Time\n");
    float total_completion_time = 0, total_turnaround_time = 0, total_waiting_time = 0;
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", proc[i].pid, proc[i].arrival_time,
proc[i].burst_time, proc[i].completion_time, proc[i].turnaround_time, proc[i].waiting_time);
        total_completion_time += proc[i].completion_time;
        total_turnaround_time += proc[i].turnaround_time;
        total_waiting_time += proc[i].waiting_time;
    }

    // Calculate and display averages
    printf("\nAverage Completion Time: %.2f\n", total_completion_time / n);
    printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);
    printf("Average Waiting Time: %.2f\n", total_waiting_time / n);
}

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

    if (n > MAX_PROCESSES) {
        printf("Number of processes exceeds maximum limit.\n");
        return 1;
    }

    struct Process proc[MAX_PROCESSES];
    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &proc[i].arrival_time);
        printf("Burst Time: ");
        scanf("%d", &proc[i].burst_time);
        proc[i].pid = i + 1;
        proc[i].remaining_time = proc[i].burst_time; // Initialize remaining time
        proc[i].turnaround_time = 0; // Initialize turnaround time
        proc[i].waiting_time = 0; // Initialize waiting time
```

```
        proc[i].completion_time = 0; // Initialize completion time
    }

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

    round_robin(proc, n, quantum);

    return 0;
}
```

Output:

```
Enter the total number of processes (max 10): 6
Enter Arrival Time and Burst Time for each process:
Process 1:
Arrival Time: 5
Burst Time: 5
Process 2:
Arrival Time: 4
Burst Time: 6
Process 3:
Arrival Time: 3
Burst Time: 7
Process 4:
Arrival Time: 1
Burst Time: 9
Process 5:
Arrival Time: 2
Burst Time: 2
Process 6:
Arrival Time: 6
Burst Time: 3
Enter Time Quantum: 4

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

Average Completion Time: 23.00
Average Turnaround Time: 19.50
Average Waiting Time: 14.17
```

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

Ans:

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

#define MAX_PROCESSES 20

struct Process {
    int pid;              // Process ID
    int arrival_time;     // Arrival time
    int burst_time;       // Burst time
    int completion_time;  // Completion time
    int turnaround_time;  // Turnaround time
    int waiting_time;     // Waiting time
    char type;            // 'S' for system process, 'U' for user process
};

void calculate_times(struct Process proc[], int n) {
    int current_time = 0;

    for (int i = 0; i < n; i++) {
        // Calculate completion time, turnaround time, and waiting time
        current_time += proc[i].burst_time;
        proc[i].completion_time = current_time;
        proc[i].turnaround_time = proc[i].completion_time - proc[i].arrival_time;
        proc[i].waiting_time = proc[i].turnaround_time - proc[i].burst_time;
    }
}

void print_processes(struct Process proc[], int n) {
    printf("PID\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting
Time\tType\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%c\n",
               proc[i].pid, proc[i].arrival_time, proc[i].burst_time,
               proc[i].completion_time, proc[i].turnaround_time,
               proc[i].waiting_time, proc[i].type);
```

```
    }
}

int compare_arrival(const void *a, const void *b) {
    return ((struct Process *)a)->arrival_time - ((struct Process *)b)->arrival_time;
}

void compute_averages(struct Process proc[], int n) {
    float total_completion_time = 0, total_turnaround_time = 0, total_waiting_time = 0;

    for (int i = 0; i < n; i++) {
        total_completion_time += proc[i].completion_time;
        total_turnaround_time += proc[i].turnaround_time;
        total_waiting_time += proc[i].waiting_time;
    }

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

int main() {
    struct Process processes[MAX_PROCESSES];
    int n, system_count = 0, user_count = 0;

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

    if (n > MAX_PROCESSES) {
        printf("Number of processes exceeds maximum limit.\n");
        return 1;
    }

    printf("Enter Process details (Type: 'S' for system, 'U' for user):\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d:\n", i + 1);
        printf("Type (S/U): ");
        scanf(" %c", &processes[i].type); // Added space to consume any whitespace
        printf("Arrival Time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Burst Time: ");
        scanf("%d", &processes[i].burst_time);
        processes[i].pid = i + 1;
```

```
        if (processes[i].type == 'S') {
            system_count++;
        } else if (processes[i].type == 'U') {
            user_count++;
        }
    }

    // Separate system and user processes
    struct Process system_processes[MAX_PROCESSES];
    struct Process user_processes[MAX_PROCESSES];

    int sys_index = 0, user_index = 0;
    for (int i = 0; i < n; i++) {
        if (processes[i].type == 'S') {
            system_processes[sys_index++] = processes[i];
        } else if (processes[i].type == 'U') {
            user_processes[user_index++] = processes[i];
        }
    }

    // Sort system and user processes by arrival time
    qsort(system_processes, system_count, sizeof(struct Process), compare_arrival);
    qsort(user_processes, user_count, sizeof(struct Process), compare_arrival);

    // Calculate times for system processes
    calculate_times(system_processes, system_count);
    // Calculate times for user processes
    calculate_times(user_processes, user_count);

    // Print results
    printf("\nSystem Processes:\n");
    print_processes(system_processes, system_count);
    compute_averages(system_processes, system_count);

    printf("\nUser Processes:\n");
    print_processes(user_processes, user_count);
    compute_averages(user_processes, user_count);

    return 0;
}
```

<u>Output:</u>

```
Enter the total number of processes (max 20): 5
Enter Process details (Type: 'S' for system, 'U' for user):
Process 1:
Type (S/U): S
Arrival Time: 0
Burst Time: 5
Process 2:
Type (S/U): U
Arrival Time: 1
Burst Time: 3
Process 3:
Type (S/U): S
Arrival Time: 2
Burst Time: 8
Process 4:
Type (S/U): U
Arrival Time: 3
Burst Time: 4
Process 5:
Type (S/U): S
Arrival Time: 4
Burst Time: 2

System Processes:
PID     Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time    Type
1       0               5               5               5               0               S
3       2               8               13              11              3               S
5       4               2               15              11              9               S

Average Completion Time: 11.00
Average Turnaround Time: 9.00
Average Waiting Time: 4.00

User Processes:
PID     Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time    Type
2       1               3               3               2               -1              U
4       3               4               7               4               0               U

Average Completion Time: 5.00
Average Turnaround Time: 3.00
Average Waiting Time: -0.50
```

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

a. Rate - Monotonic
b. Earliest-deadline First
c. Proportional scheduling

Ans:

a. Rate - Monotonic

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

typedef struct {
    int task_id;
    int period;
    int execution_time;
    int remaining_time;
} Task;

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_hyperperiod(Task tasks[], int num_tasks) {
    int hyperperiod = tasks[0].period;
    for (int i = 1; i < num_tasks; i++) {
        hyperperiod = lcm(hyperperiod, tasks[i].period);
    }
    return hyperperiod;
}

void rate_monotonic_scheduler(Task tasks[], int num_tasks, int hyperperiod) {
    for (int time = 0; time < hyperperiod; time++) {
        int task_to_run = -1;

        for (int i = 0; i < num_tasks; i++) {
            if (time % tasks[i].period == 0) {
```

```c
            tasks[i].remaining_time = tasks[i].execution_time;
        }
        if (tasks[i].remaining_time > 0) {
            if (task_to_run == -1 || tasks[i].period < tasks[task_to_run].period) {
                task_to_run = i;
            }
        }
    }

    if (task_to_run != -1) {
        tasks[task_to_run].remaining_time--;
        printf("Time %d: Running task %d\n", time, tasks[task_to_run].task_id);
    } else {
        printf("Time %d: Idle\n", time);
    }

    usleep(1000000); // Simulate real-time delay (1 second)
  }
}

int main() {
    int num_tasks;

    printf("Enter the number of tasks: ");
    scanf("%d", &num_tasks);

    Task *tasks = (Task *)malloc(num_tasks * sizeof(Task));

    for (int i = 0; i < num_tasks; i++) {
        tasks[i].task_id = i + 1;
        printf("Enter period for Task %d: ", tasks[i].task_id);
        scanf("%d", &tasks[i].period);
        printf("Enter execution time for Task %d: ", tasks[i].task_id);
        scanf("%d", &tasks[i].execution_time);
        tasks[i].remaining_time = 0;
    }

    int hyperperiod = calculate_hyperperiod(tasks, num_tasks);
    printf("Hyperperiod: %d\n", hyperperiod);
    rate_monotonic_scheduler(tasks, num_tasks, hyperperiod);

    free(tasks);
    return 0;
```

```
}
```

Output:

```
Enter the number of tasks: 2
Enter period for Task 1: 50
Enter execution time for Task 1: 20
Enter period for Task 2: 100
Enter execution time for Task 2: 35
Hyperperiod: 100
Time 0: Running task 1
Time 1: Running task 1
Time 2: Running task 1
Time 3: Running task 1
Time 4: Running task 1
Time 5: Running task 1
Time 6: Running task 1
Time 7: Running task 1
Time 8: Running task 1
Time 9: Running task 1
Time 10: Running task 1
Time 11: Running task 1
Time 12: Running task 1
Time 13: Running task 1
Time 14: Running task 1
Time 15: Running task 1
Time 16: Running task 1
Time 17: Running task 1
Time 18: Running task 1
Time 19: Running task 1
Time 20: Running task 2
Time 21: Running task 2
Time 22: Running task 2
Time 23: Running task 2
Time 24: Running task 2
Time 25: Running task 2
Time 26: Running task 2
Time 27: Running task 2
Time 28: Running task 2
Time 29: Running task 2
Time 30: Running task 2
Time 31: Running task 2
Time 32: Running task 2
Time 33: Running task 2
Time 34: Running task 2
Time 35: Running task 2
Time 36: Running task 2
Time 37: Running task 2
Time 38: Running task 2
Time 39: Running task 2
Time 40: Running task 2
Time 41: Running task 2
Time 42: Running task 2
Time 43: Running task 2
Time 44: Running task 2
Time 45: Running task 2
Time 46: Running task 2
Time 47: Running task 2
Time 48: Running task 2
Time 49: Running task 2
```

```
Time 50: Running task 1
Time 51: Running task 1
Time 52: Running task 1
Time 53: Running task 1
Time 54: Running task 1
Time 55: Running task 1
Time 56: Running task 1
Time 57: Running task 1
Time 58: Running task 1
Time 59: Running task 1
Time 60: Running task 1
Time 61: Running task 1
Time 62: Running task 1
Time 63: Running task 1
Time 64: Running task 1
Time 65: Running task 1
Time 66: Running task 1
Time 67: Running task 1
Time 68: Running task 1
Time 69: Running task 1
Time 70: Running task 2
Time 71: Running task 2
Time 72: Running task 2
Time 73: Running task 2
Time 74: Running task 2
Time 75: Idle
Time 76: Idle
Time 77: Idle
Time 78: Idle
Time 79: Idle
Time 80: Idle
Time 81: Idle
Time 82: Idle
Time 83: Idle
Time 84: Idle
Time 85: Idle
Time 86: Idle
Time 87: Idle
Time 88: Idle
Time 89: Idle
Time 90: Idle
Time 91: Idle
Time 92: Idle
Time 93: Idle
Time 94: Idle
Time 95: Idle
Time 96: Idle
Time 97: Idle
Time 98: Idle
Time 99: Idle
```

b. Earliest-deadline First

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

typedef struct {
    int task_id;
    int period;
    int execution_time;
    int remaining_time;
    int deadline;
} Task;

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_hyperperiod(Task tasks[], int num_tasks) {
    int hyperperiod = tasks[0].period;
    for (int i = 1; i < num_tasks; i++) {
        hyperperiod = lcm(hyperperiod, tasks[i].period);
    }
    return hyperperiod;
}

void earliest_deadline_first_scheduler(Task tasks[], int num_tasks, int hyperperiod) {
    for (int time = 0; time < hyperperiod; time++) {
        int task_to_run = -1;

        for (int i = 0; i < num_tasks; i++) {
            if (time % tasks[i].period == 0) {
                tasks[i].remaining_time = tasks[i].execution_time;
                tasks[i].deadline = time + tasks[i].period;
            }
            if (tasks[i].remaining_time > 0) {
                if (task_to_run == -1 || tasks[i].deadline < tasks[task_to_run].deadline) {
                    task_to_run = i;
                }
```

```
            }
        }

        if (task_to_run != -1) {
            tasks[task_to_run].remaining_time--;
            printf("Time %d: Running task %d with deadline %d\n", time,
tasks[task_to_run].task_id, tasks[task_to_run].deadline);
        } else {
            printf("Time %d: Idle\n", time);
        }

        usleep(1000000); // Simulate real-time delay (1 second)
    }
}

int main() {
    int num_tasks;

    printf("Enter the number of tasks: ");
    scanf("%d", &num_tasks);

    Task *tasks = (Task *)malloc(num_tasks * sizeof(Task));

    for (int i = 0; i < num_tasks; i++) {
        tasks[i].task_id = i + 1;
        printf("Enter period for Task %d: ", tasks[i].task_id);
        scanf("%d", &tasks[i].period);
        printf("Enter execution time for Task %d: ", tasks[i].task_id);
        scanf("%d", &tasks[i].execution_time);
        tasks[i].remaining_time = 0;
        tasks[i].deadline = 0;
    }

    int hyperperiod = calculate_hyperperiod(tasks, num_tasks);
    printf("Hyperperiod: %d\n", hyperperiod);
    earliest_deadline_first_scheduler(tasks, num_tasks, hyperperiod);

    free(tasks);
    return 0;
}
```

<u>Output:</u>

```
Enter the number of tasks: 2
Enter period for Task 1: 50
Enter execution time for Task 1: 25
Enter period for Task 2: 80
Enter execution time for Task 2: 35
Hyperperiod: 400
Time 0: Running task 1 with deadline 50
Time 1: Running task 1 with deadline 50
Time 2: Running task 1 with deadline 50
Time 3: Running task 1 with deadline 50
Time 4: Running task 1 with deadline 50
Time 5: Running task 1 with deadline 50
Time 6: Running task 1 with deadline 50
Time 7: Running task 1 with deadline 50
Time 8: Running task 1 with deadline 50
Time 9: Running task 1 with deadline 50
Time 10: Running task 1 with deadline 50
Time 11: Running task 1 with deadline 50
Time 12: Running task 1 with deadline 50
Time 13: Running task 1 with deadline 50
Time 14: Running task 1 with deadline 50
Time 15: Running task 1 with deadline 50
Time 16: Running task 1 with deadline 50
Time 17: Running task 1 with deadline 50
Time 18: Running task 1 with deadline 50
Time 19: Running task 1 with deadline 50
Time 20: Running task 1 with deadline 50
Time 21: Running task 1 with deadline 50
Time 22: Running task 1 with deadline 50
Time 23: Running task 1 with deadline 50
Time 24: Running task 1 with deadline 50
Time 25: Running task 2 with deadline 80
Time 26: Running task 2 with deadline 80
Time 27: Running task 2 with deadline 80
Time 28: Running task 2 with deadline 80
Time 29: Running task 2 with deadline 80
Time 30: Running task 2 with deadline 80
Time 31: Running task 2 with deadline 80
Time 32: Running task 2 with deadline 80
Time 33: Running task 2 with deadline 80
Time 34: Running task 2 with deadline 80
Time 35: Running task 2 with deadline 80
Time 36: Running task 2 with deadline 80
Time 37: Running task 2 with deadline 80
Time 38: Running task 2 with deadline 80
Time 39: Running task 2 with deadline 80
Time 40: Running task 2 with deadline 80
Time 41: Running task 2 with deadline 80
Time 42: Running task 2 with deadline 80
Time 43: Running task 2 with deadline 80
Time 44: Running task 2 with deadline 80
Time 45: Running task 2 with deadline 80
Time 46: Running task 2 with deadline 80
Time 47: Running task 2 with deadline 80
Time 48: Running task 2 with deadline 80
Time 49: Running task 2 with deadline 80
Time 50: Running task 2 with deadline 80
```

```
Time 51: Running task 2 with deadline 80
Time 52: Running task 2 with deadline 80
Time 53: Running task 2 with deadline 80
Time 54: Running task 2 with deadline 80
Time 55: Running task 2 with deadline 80
Time 56: Running task 2 with deadline 80
Time 57: Running task 2 with deadline 80
Time 58: Running task 2 with deadline 80
Time 59: Running task 2 with deadline 80
Time 60: Running task 1 with deadline 100
Time 61: Running task 1 with deadline 100
Time 62: Running task 1 with deadline 100
Time 63: Running task 1 with deadline 100
Time 64: Running task 1 with deadline 100
Time 65: Running task 1 with deadline 100
Time 66: Running task 1 with deadline 100
Time 67: Running task 1 with deadline 100
Time 68: Running task 1 with deadline 100
Time 69: Running task 1 with deadline 100
Time 70: Running task 1 with deadline 100
Time 71: Running task 1 with deadline 100
Time 72: Running task 1 with deadline 100
Time 73: Running task 1 with deadline 100
Time 74: Running task 1 with deadline 100
Time 75: Running task 1 with deadline 100
Time 76: Running task 1 with deadline 100
Time 77: Running task 1 with deadline 100
Time 78: Running task 1 with deadline 100
Time 79: Running task 1 with deadline 100
Time 80: Running task 1 with deadline 100
Time 81: Running task 1 with deadline 100
Time 82: Running task 1 with deadline 100
Time 83: Running task 1 with deadline 100
Time 84: Running task 1 with deadline 100
Time 85: Running task 2 with deadline 160
Time 86: Running task 2 with deadline 160
Time 87: Running task 2 with deadline 160
Time 88: Running task 2 with deadline 160
Time 89: Running task 2 with deadline 160
Time 90: Running task 2 with deadline 160
Time 91: Running task 2 with deadline 160
Time 92: Running task 2 with deadline 160
Time 93: Running task 2 with deadline 160
Time 94: Running task 2 with deadline 160
Time 95: Running task 2 with deadline 160
Time 96: Running task 2 with deadline 160
Time 97: Running task 2 with deadline 160
Time 98: Running task 2 with deadline 160
Time 99: Running task 2 with deadline 160
Time 100: Running task 1 with deadline 150
```

c. Proportional Scheduling

```
#include <stdio.h>

#define MAX_PROCESSES 10

// Structure to represent a process
struct Process {
    int pid;             // Process ID
    int arrival_time;    // Arrival time
    int burst_time;      // Burst time (total CPU time required)
    int weight;          // Weight for proportional scheduling
    int allocated_time;  // Time allocated to this process in the current scheduling cycle
};

// Function to calculate proportional scheduling
void proportionalScheduling(struct Process proc[], int n) {
    int total_weight = 0;

    // Calculate total weight of all processes
    for (int i = 0; i < n; i++) {
        total_weight += proc[i].weight;
    }

    // Calculate and print the allocated time for each process
    printf("PID\tArrival Time\tBurst Time\tWeight\tAllocated Time\n");
    for (int i = 0; i < n; i++) {
        // Calculate the allocated time based on proportional weight
        proc[i].allocated_time = (proc[i].weight * 100) / total_weight;
        printf("%d\t%d\t\t%d\t\t%d\t%d\n", proc[i].pid, proc[i].arrival_time, proc[i].burst_time,
proc[i].weight, proc[i].allocated_time);
    }
}

int main() {
    int n;

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

    if (n <= 0 || n > MAX_PROCESSES) {
        printf("Invalid number of processes.\n");
```

```
        return 1;
    }

    struct Process proc[MAX_PROCESSES];

    // Input process details
    printf("Enter Arrival Time, Burst Time, and Weight for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &proc[i].arrival_time);
        printf("Burst Time: ");
        scanf("%d", &proc[i].burst_time);
        printf("Weight: ");
        scanf("%d", &proc[i].weight);
        proc[i].pid = i + 1; // Assign PID
    }

    // Perform proportional scheduling
    proportionalScheduling(proc, n);

    return 0;
}
```

Output:

```
Enter the total number of processes (max 10): 3
Enter Arrival Time, Burst Time, and Weight for each process:
Process 1:
Arrival Time: 0
Burst Time: 10
Weight: 5
Process 2:
Arrival Time: 2
Burst Time: 20
Weight: 3
Process 3:
Arrival Time: 4
Burst Time: 15
Weight: 2
PID     Arrival Time    Burst Time      Weight  Allocated Time
1       0               10              5       50
2       2               20              3       30
3       4               15              2       20
```

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

Ans:

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

#define BUFFER_SIZE 5 // Size of the buffer
#define MAX_PRODUCE 10 // Maximum number of items to produce/consume

int buffer[BUFFER_SIZE]; // Buffer for storing produced items
int count = 0; // Number of items in the buffer

int mutex = 1; // Binary semaphore for mutual exclusion
int empty = BUFFER_SIZE; // Counting semaphore for empty slots
int full = 0; // Counting semaphore for full slots

// wait function
void wait(int *semaphore) {
    while (*semaphore <= 0) {
        // busy wait
    }
    (*semaphore)--;
}

// signal function
void signal(int *semaphore) {
    (*semaphore)++;
}

void* producer(void* arg) {
    int item;

    for (int i = 0; i < MAX_PRODUCE; i++) {
        item = i + 1; // Produce an item

        // Wait on empty and mutex
        wait(&empty);
        wait(&mutex);

        // Critical section: add item to the buffer
```

```
        buffer[count] = item;
        printf("Producer produced item %d\n", item);
        count++;

        // Signal mutex and full
        signal(&mutex);
        signal(&full);

        sleep(1); // Simulate time taken to produce
    }
    return NULL;
}

void* consumer(void* arg) {
    int item;

    for (int i = 0; i < MAX_PRODUCE; i++) {
        // Wait on full and mutex
        wait(&full);
        wait(&mutex);

        // Critical section: remove item from the buffer
        item = buffer[count - 1];
        count--;
        printf("Consumer consumed item %d\n", item);

        // Signal mutex and empty
        signal(&mutex);
        signal(&empty);

        sleep(1); // Simulate time taken to consume
    }
    return NULL;
}

int main() {
    pthread_t prod, cons;

    // Create producer and consumer threads
    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    // Wait for the threads to finish
```

```
    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    return 0;
}
```

Output:

```
Producer produced item 1
Consumer consumed item 1
Producer produced item 2
Consumer consumed item 2
Producer produced item 3
Consumer consumed item 3
Producer produced item 4
Consumer consumed item 4
Producer produced item 5
Consumer consumed item 5
Producer produced item 6
Consumer consumed item 6
Producer produced item 7
Consumer consumed item 7
Producer produced item 8
Consumer consumed item 8
Producer produced item 9
Consumer consumed item 9
Producer produced item 10
Consumer consumed item 10
```

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

Ans:

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

#define MAX_PHILOSOPHERS 10

int chopstick[MAX_PHILOSOPHERS]; // Semaphore array for chopsticks
int num_philosophers;

// wait function for semaphores
void semaphore_wait(int *semaphore) {
    while (*semaphore <= 0) {
        // busy wait
    }
    (*semaphore)--;
}

// signal function for semaphores
void semaphore_signal(int *semaphore) {
    (*semaphore)++;
}

void *philosopher(void *arg) {
    int philosopher_id = *((int *)arg);
    int left_fork = philosopher_id;
    int right_fork = (philosopher_id + 1) % num_philosophers;

    while (1) {
        printf("Philosopher %d is thinking.\n", philosopher_id);
        sleep(1); // thinking for 1 second

        // Pick up forks
        semaphore_wait(&chopstick[left_fork]);
        semaphore_wait(&chopstick[right_fork]);

        // Eat
        printf("Philosopher %d is eating.\n", philosopher_id);
        sleep(1); // eating for 1 second
```

```c
        // Put down forks
        semaphore_signal(&chopstick[left_fork]);
        semaphore_signal(&chopstick[right_fork]);

        // Repeat
    }
    return NULL;
}

int main() {
    pthread_t philosophers[MAX_PHILOSOPHERS];
    int philosopher_ids[MAX_PHILOSOPHERS];

    printf("Enter the number of philosophers (max %d): ", MAX_PHILOSOPHERS);
    scanf("%d", &num_philosophers);

    if (num_philosophers < 2 || num_philosophers > MAX_PHILOSOPHERS) {
        printf("Invalid number of philosophers. Exiting.\n");
        return 1;
    }

    for (int i = 0; i < num_philosophers; ++i) {
        chopstick[i] = 1; // Initialize chopsticks to 1 (available)
    }

    for (int i = 0; i < num_philosophers; ++i) {
        philosopher_ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, &philosopher_ids[i]);
    }

    for (int i = 0; i < num_philosophers; ++i) {
        pthread_join(philosophers[i], NULL);
    }

    return 0;
}
```

Output:

```
Enter the number of philosophers (max 10): 3
Philosopher 0 is thinking.
Philosopher 1 is thinking.
Philosopher 2 is thinking.
Philosopher 0 is eating.
Philosopher 1 is eating.
Philosopher 0 is thinking.
Philosopher 1 is thinking.
Philosopher 2 is eating.
Philosopher 2 is thinking.
Philosopher 1 is eating.
Philosopher 0 is eating.
Philosopher 1 is thinking.
Philosopher 0 is thinking.
Philosopher 2 is eating.
Philosopher 0 is eating.
Philosopher 2 is thinking.
Philosopher 1 is eating.
Philosopher 0 is thinking.
Philosopher 1 is thinking.
Philosopher 2 is eating.
Philosopher 2 is thinking.
Philosopher 1 is eating.
Philosopher 0 is eating.
```

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

<u>Ans:</u>

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

// Function to find if the system is in a safe state
bool isSafeState(int P, int R, int processes[], int avail[], int max[][R], int allot[][R]) {
    int need[P][R];
    for (int i = 0; i < P; i++)
        for (int j = 0; j < R; j++)
            need[i][j] = max[i][j] - allot[i][j];

    bool finish[P];
    for (int i = 0; i < P; i++)
        finish[i] = 0;

    int safeSeq[P];
    int work[R];
    for (int i = 0; i < R; i++)
        work[i] = avail[i];

    int count = 0;
    while (count < P) {
        bool found = false;
        for (int p = 0; p < P; p++) {
            if (finish[p] == 0) {
                int j;
                for (j = 0; j < R; j++)
                    if (need[p][j] > work[j])
                        break;
                if (j == R) {
                    for (int k = 0; k < R; k++)
                        work[k] += allot[p][k];
                    safeSeq[count++] = p;
                    finish[p] = 1;
                    found = true;
                }
            }
```

```
        }
        if (found == false) {
            printf("System is not in a safe state\n");
            return false;
        }
    }
    printf("System is in a safe state.\nSafe sequence is: ");
    for (int i = 0; i < P; i++)
        printf("%d ", safeSeq[i]);
    printf("\n");
    return true;
}

int main() {
    int P, R;

    // User input for number of processes and resources
    printf("Enter the number of processes: ");
    scanf("%d", &P);
    printf("Enter the number of resources: ");
    scanf("%d", &R);

    int processes[P];
    int total[R];
    int avail[R];
    int max[P][R];
    int allot[P][R];

    // User input for total instances of each resource
    printf("Enter the total instances of each resource: ");
    for (int i = 0; i < R; i++)
        scanf("%d", &total[i]);

    // User input for allocation resource matrix
    printf("Enter the allocation resource matrix:\n");
    for (int i = 0; i < P; i++) {
        printf("Process %d: ", i);
        for (int j = 0; j < R; j++)
            scanf("%d", &allot[i][j]);
    }

    // Calculate available resources by subtracting allocated from total instances
    for (int j = 0; j < R; j++) {
```

```
        int sum = 0;
        for (int i = 0; i < P; i++)
            sum += allot[i][j];
        avail[j] = total[j] - sum;
    }

    // User input for maximum resource matrix
    printf("Enter the maximum resource matrix:\n");
    for (int i = 0; i < P; i++) {
        printf("Process %d: ", i);
        for (int j = 0; j < R; j++)
            scanf("%d", &max[i][j]);
    }

    for (int i = 0; i < P; i++)
        processes[i] = i;

    isSafeState(P, R, processes, avail, max, allot);

    return 0;
}
```

<u>Output:</u>

```
Enter the number of processes: 5
Enter the number of resources: 3
Enter the total instances of each resource: 10 5 7
Enter the allocation resource matrix:
Process 0: 0 1 0
Process 1: 2 0 0
Process 2: 3 0 2
Process 3: 2 1 1
Process 4: 0 0 2
Enter the maximum resource matrix:
Process 0: 7 5 3
Process 1: 3 2 2
Process 2: 9 0 2
Process 3: 2 2 2
Process 4: 4 3 3
System is in a safe state.
Safe sequence is: 1 3 4 0 2
```

8. Write a C program to simulate deadlock detection

Ans:

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

void deadlockDetection(int P, int R, int processes[], int avail[], int alloc[][R], int request[][R]) {
    int work[R];
    bool finish[P];

    // Initialize work with available resources
    for (int i = 0; i < R; i++)
        work[i] = avail[i];

    // Initialize finish for all processes as false
    for (int i = 0; i < P; i++)
        finish[i] = false;

    // Find an unfinished process with its requests less than or equal to work
    bool found;
    do {
        found = false;
        for (int i = 0; i < P; i++) {
            if (!finish[i]) {
                int j;
                for (j = 0; j < R; j++) {
                    if (request[i][j] > work[j])
                        break;
                }
                if (j == R) { // If all requests of process i can be satisfied
                    for (int k = 0; k < R; k++)
                        work[k] += alloc[i][k];
                    finish[i] = true;
                    found = true;
                }
            }
        }
    } while (found);

    // Check if any process is not finished
    bool deadlock = false;
```

```
    for (int i = 0; i < P; i++) {
        if (!finish[i]) {
            deadlock = true;
            printf("Process %d is in deadlock.\n", i);
        }
    }
    if (!deadlock)
        printf("No deadlock detected.\n");
}

int main() {
    int P, R;

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

    printf("Enter the number of resource types: ");
    scanf("%d", &R);

    int processes[P];
    int avail[R];
    int alloc[P][R];
    int request[P][R];

    for (int i = 0; i < P; i++)
        processes[i] = i;

    printf("Enter the available resources (A B C ...): ");
    for (int i = 0; i < R; i++)
        scanf("%d", &avail[i]);

    printf("Enter the allocation matrix:\n");
    for (int i = 0; i < P; i++) {
        printf("Process %d: ", i);
        for (int j = 0; j < R; j++)
            scanf("%d", &alloc[i][j]);
    }

    printf("Enter the request matrix:\n");
    for (int i = 0; i < P; i++) {
        printf("Process %d: ", i);
        for (int j = 0; j < R; j++)
            scanf("%d", &request[i][j]);
```

```
    }

    deadlockDetection(P, R, processes, avail, alloc, request);

    return 0;
}
```

Output:

```
Enter the number of processes: 4
Enter the number of resource types: 3
Enter the available resources (A B C ...): 0 0 0
Enter the allocation matrix:
Process 0: 1 0 2
Process 1: 2 1 1
Process 2: 1 0 3
Process 3: 1 2 2
Enter the request matrix:
Process 0: 0 0 1
Process 1: 1 0 2
Process 2: 0 0 0
Process 3: 3 3 0
Process 3 is in deadlock.
```

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

a. Worst-fit
b. Best-fit
c. First-fit
d. Next-fit

Ans:

a. Worst-fit

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

void worstFit(int blocks[], int n, int process[], int m);
void printBlocks(int blocks[], int n);

int main() {
    int n, m;
    printf("Enter the number of memory blocks: ");
    scanf("%d", &n);
    int blocks[n], original_blocks[n];
    printf("Enter the size of each memory block: \n");
    for (int i = 0; i < n; i++) {
        printf("Block %d: ", i + 1);
        scanf("%d", &blocks[i]);
        original_blocks[i] = blocks[i]; // Save the original block sizes
    }

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

    printf("\nWorst Fit Allocation:\n");
    worstFit(blocks, n, process, m);
```

```c
    return 0;
}

void worstFit(int blocks[], int n, int process[], int m) {
    int allocation[m];
    for (int i = 0; i < m; i++) {
        allocation[i] = -1;
    }

    for (int i = 0; i < m; i++) {
        int worstIdx = -1;
        for (int j = 0; j < n; j++) {
            if (blocks[j] >= process[i]) {
                if (worstIdx == -1 || blocks[j] > blocks[worstIdx])
                    worstIdx = j;
            }
        }

        if (worstIdx != -1) {
            allocation[i] = worstIdx;
            blocks[worstIdx] -= process[i];
        }
    }

    printf("Process No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < m; i++) {
        printf("%d\t\t%d\t\t", i + 1, process[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
    printBlocks(blocks, n);
}

void printBlocks(int blocks[], int n) {
    printf("Memory Blocks: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", blocks[i]);
    }
    printf("\n");
}
```

Output:

```
Enter the number of memory blocks: 6
Enter the size of each memory block:
Block 1: 300
Block 2: 600
Block 3: 350
Block 4: 200
Block 5: 750
Block 6: 125
Enter the number of processes: 5
Enter the size of each process:
Process 1: 115
Process 2: 500
Process 3: 358
Process 4: 200
Process 5: 375

Worst Fit Allocation:
Process No.     Process Size    Block No.
1               115             5
2               500             5
3               358             2
4               200             3
5               375             Not Allocated
Memory Blocks: 300 242 150 200 135 125
```

b. Best-Fit

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

void bestFit(int blocks[], int n, int process[], int m);
void printBlocks(int blocks[], int n);

int main() {
    int n, m;
    printf("Enter the number of memory blocks: ");
    scanf("%d", &n);
    int blocks[n], original_blocks[n];
    printf("Enter the size of each memory block: \n");
    for (int i = 0; i < n; i++) {
        printf("Block %d: ", i + 1);
        scanf("%d", &blocks[i]);
        original_blocks[i] = blocks[i]; // Save the original block sizes
    }

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

    printf("\nBest Fit Allocation:\n");
    bestFit(blocks, n, process, m);

    return 0;
}

void bestFit(int blocks[], int n, int process[], int m) {
    int allocation[m];
    for (int i = 0; i < m; i++) {
        allocation[i] = -1;
    }

    for (int i = 0; i < m; i++) {
        int bestIdx = -1;
```

```
        for (int j = 0; j < n; j++) {
            if (blocks[j] >= process[i]) {
                if (bestIdx == -1 || blocks[j] < blocks[bestIdx])
                    bestIdx = j;
            }
        }

        if (bestIdx != -1) {
            allocation[i] = bestIdx;
            blocks[bestIdx] -= process[i];
        }
    }

    printf("Process No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < m; i++) {
        printf("%d\t\t%d\t\t", i + 1, process[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
    printBlocks(blocks, n);
}

void printBlocks(int blocks[], int n) {
    printf("Memory Blocks: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", blocks[i]);
    }
    printf("\n");
}
```

<u>Output:</u>

```
Enter the number of memory blocks: 6
Enter the size of each memory block:
Block 1: 300
Block 2: 600
Block 3: 350
Block 4: 200
Block 5: 750
Block 6: 125
Enter the number of processes: 5
Enter the size of each process:
Process 1: 115
Process 2: 500
Process 3: 358
Process 4: 200
Process 5: 375

Best Fit Allocation:
Process No.     Process Size    Block No.
1               115             6
2               500             2
3               358             5
4               200             4
5               375             5
Memory Blocks: 300 100 350 0 17 10
```

c. First-Fit

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

void firstFit(int blocks[], int n, int process[], int m);
void printBlocks(int blocks[], int n);

int main() {
    int n, m;
    printf("Enter the number of memory blocks: ");
    scanf("%d", &n);
    int blocks[n], original_blocks[n];
    printf("Enter the size of each memory block: \n");
    for (int i = 0; i < n; i++) {
        printf("Block %d: ", i + 1);
        scanf("%d", &blocks[i]);
        original_blocks[i] = blocks[i]; // Save the original block sizes
    }

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

    printf("\nFirst Fit Allocation:\n");
    firstFit(blocks, n, process, m);

    return 0;
}

void firstFit(int blocks[], int n, int process[], int m) {
    int allocation[m];
    for (int i = 0; i < m; i++) {
        allocation[i] = -1;
    }

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

```c
            if (blocks[j] >= process[i]) {
                allocation[i] = j;
                blocks[j] -= process[i];
                break;
            }
        }
    }

    printf("Process No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < m; i++) {
        printf("%d\t\t%d\t\t", i + 1, process[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
    printBlocks(blocks, n);
}

void printBlocks(int blocks[], int n) {
    printf("Memory Blocks: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", blocks[i]);
    }
    printf("\n");
}
```

Output:

```
Enter the number of memory blocks: 6
Enter the size of each memory block:
Block 1: 300
Block 2: 600
Block 3: 350
Block 4: 200
Block 5: 750
Block 6: 125
Enter the number of processes: 5
Enter the size of each process:
Process 1: 115
Process 2: 500
Process 3: 358
Process 4: 200
Process 5: 375

First Fit Allocation:
Process No.     Process Size    Block No.
1               115             1
2               500             2
3               358             5
4               200             3
5               375             5
Memory Blocks: 185 100 150 200 17 125
```

d. Next-Fit

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

void nextFit(int blocks[], int n, int process[], int m);
void printBlocks(int blocks[], int n);

int main() {
    int n, m;
    printf("Enter the number of memory blocks: ");
    scanf("%d", &n);
    int blocks[n], original_blocks[n];
    printf("Enter the size of each memory block: \n");
    for (int i = 0; i < n; i++) {
        printf("Block %d: ", i + 1);
        scanf("%d", &blocks[i]);
        original_blocks[i] = blocks[i]; // Save the original block sizes
    }

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

    printf("\nNext Fit Allocation:\n");
    nextFit(blocks, n, process, m);

    return 0;
}

void nextFit(int blocks[], int n, int process[], int m) {
    int allocation[m];
    for (int i = 0; i < m; i++) {
        allocation[i] = -1;
    }

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

```
        int count = 0;
        while (count < n) {
            if (blocks[j] >= process[i]) {
                allocation[i] = j;
                blocks[j] -= process[i];
                break;
            }
            j = (j + 1) % n;
            count++;
        }
    }

    printf("Process No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < m; i++) {
        printf("%d\t\t%d\t\t", i + 1, process[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
    printBlocks(blocks, n);
}

void printBlocks(int blocks[], int n) {
    printf("Memory Blocks: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", blocks[i]);
    }
    printf("\n");
}
```

Output:

```
Enter the number of memory blocks: 6
Enter the size of each memory block:
Block 1: 300
Block 2: 600
Block 3: 350
Block 4: 200
Block 5: 750
Block 6: 125
Enter the number of processes: 5
Enter the size of each process:
Process 1: 115
Process 2: 500
Process 3: 358
Process 4: 200
Process 5: 375

Next Fit Allocation:
Process No.     Process Size    Block No.
1               115             1
2               500             2
3               358             5
4               200             5
5               375             Not Allocated
Memory Blocks: 185 100 350 200 192 125
```

Program containing all contagious memory allocation techniques (First-Fit , Best-Fit , Worst-Fit , Next-Fit).

<u>Ans:</u>

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

// Function prototypes
void firstFit(int blocks[], int n, int process[], int m);
void bestFit(int blocks[], int n, int process[], int m);
void worstFit(int blocks[], int n, int process[], int m);
void nextFit(int blocks[], int n, int process[], int m);

void printBlocks(int blocks[], int n) {
    printf("Memory Blocks: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", blocks[i]);
    }
    printf("\n");
}

int main() {
    int n, m;
    printf("Enter the number of memory blocks: ");
    scanf("%d", &n);
    int blocks[n], original_blocks[n];
    printf("Enter the size of each memory block: \n");
    for (int i = 0; i < n; i++) {
        printf("Block %d: ", i + 1);
        scanf("%d", &blocks[i]);
        original_blocks[i] = blocks[i]; // Save the original block sizes
    }

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

```
    }

    printf("\nFirst Fit Allocation:\n");
    firstFit(blocks, n, process, m);
    for (int i = 0; i < n; i++) blocks[i] = original_blocks[i]; // Reset block sizes

    printf("\nBest Fit Allocation:\n");
    bestFit(blocks, n, process, m);
    for (int i = 0; i < n; i++) blocks[i] = original_blocks[i]; // Reset block sizes

    printf("\nWorst Fit Allocation:\n");
    worstFit(blocks, n, process, m);
    for (int i = 0; i < n; i++) blocks[i] = original_blocks[i]; // Reset block sizes

    printf("\nNext Fit Allocation:\n");
    nextFit(blocks, n, process, m);
    for (int i = 0; i < n; i++) blocks[i] = original_blocks[i]; // Reset block sizes

    return 0;
}

void firstFit(int blocks[], int n, int process[], int m) {
    int allocation[m];
    for (int i = 0; i < m; i++) {
        allocation[i] = -1;
    }

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (blocks[j] >= process[i]) {
                allocation[i] = j;
                blocks[j] -= process[i];
                break;
            }
        }
    }

    printf("Process No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < m; i++) {
        printf("%d\t\t%d\t\t", i + 1, process[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
```

```
            printf("Not Allocated\n");
    }
    printBlocks(blocks, n);
}

void bestFit(int blocks[], int n, int process[], int m) {
    int allocation[m];
    for (int i = 0; i < m; i++) {
        allocation[i] = -1;
    }

    for (int i = 0; i < m; i++) {
        int bestIdx = -1;
        for (int j = 0; j < n; j++) {
            if (blocks[j] >= process[i]) {
                if (bestIdx == -1 || blocks[j] < blocks[bestIdx])
                    bestIdx = j;
            }
        }

        if (bestIdx != -1) {
            allocation[i] = bestIdx;
            blocks[bestIdx] -= process[i];
        }
    }

    printf("Process No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < m; i++) {
        printf("%d\t\t%d\t\t", i + 1, process[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
    printBlocks(blocks, n);
}

void worstFit(int blocks[], int n, int process[], int m) {
    int allocation[m];
    for (int i = 0; i < m; i++) {
        allocation[i] = -1;
    }
```

```
    for (int i = 0; i < m; i++) {
        int worstIdx = -1;
        for (int j = 0; j < n; j++) {
            if (blocks[j] >= process[i]) {
                if (worstIdx == -1 || blocks[j] > blocks[worstIdx])
                    worstIdx = j;
            }
        }

        if (worstIdx != -1) {
            allocation[i] = worstIdx;
            blocks[worstIdx] -= process[i];
        }
    }

    printf("Process No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < m; i++) {
        printf("%d\t\t%d\t\t", i + 1, process[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
    printBlocks(blocks, n);
}

void nextFit(int blocks[], int n, int process[], int m) {
    int allocation[m];
    for (int i = 0; i < m; i++) {
        allocation[i] = -1;
    }

    int j = 0;
    for (int i = 0; i < m; i++) {
        int count = 0;
        while (count < n) {
            if (blocks[j] >= process[i]) {
                allocation[i] = j;
                blocks[j] -= process[i];
                break;
            }
            j = (j + 1) % n;
            count++;
```

```c
        }
    }

    printf("Process No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < m; i++) {
        printf("%d\t\t%d\t\t", i + 1, process[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
    printBlocks(blocks, n);
}
```

Output:

```
Enter the number of memory blocks: 6
Enter the size of each memory block:
Block 1: 300
Block 2: 600
Block 3: 350
Block 4: 200
Block 5: 750
Block 6: 125
Enter the number of processes: 5
Enter the size of each process:
Process 1: 115
Process 2: 500
Process 3: 358
Process 4: 200
Process 5: 375

First Fit Allocation:
Process No.     Process Size    Block No.
1               115             1
2               500             2
3               358             5
4               200             3
5               375             5
Memory Blocks: 185 100 150 200 17 125

Best Fit Allocation:
Process No.     Process Size    Block No.
1               115             6
2               500             2
3               358             5
4               200             4
5               375             5
Memory Blocks: 300 100 350 0 17 10

Worst Fit Allocation:
Process No.     Process Size    Block No.
1               115             5
2               500             5
3               358             2
4               200             3
5               375             Not Allocated
Memory Blocks: 300 242 150 200 135 125

Next Fit Allocation:
Process No.     Process Size    Block No.
1               115             1
2               500             2
3               358             5
4               200             5
5               375             Not Allocated
Memory Blocks: 185 100 350 200 192 125
```

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

<u>Ans:</u>

a. FIFO

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

void fifo(int pages[], int n, int frames);

int main() {
    int n, frames;

    printf("Enter the number of pages: ");
    scanf("%d", &n);
    int pages[n];
    printf("Enter the page sequence: \n");
    for (int i = 0; i < n; i++) {
        printf("Page %d: ", i + 1);
        scanf("%d", &pages[i]);
    }

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

    printf("\nFIFO Page Replacement:\n");
    fifo(pages, n, frames);

    return 0;
}

void fifo(int pages[], int n, int frames) {
    int frame[frames];
    for (int i = 0; i < frames; i++)
        frame[i] = -1;

    int pageFaults = 0, pageHits = 0, index = 0;
    for (int i = 0; i < n; i++) {
```

```
        bool found = false;
        for (int j = 0; j < frames; j++) {
            if (frame[j] == pages[i]) {
                found = true;
                pageHits++;
                break;
            }
        }
        if (!found) {
            frame[index] = pages[i];
            index = (index + 1) % frames;
            pageFaults++;
        }

        printf("Frame: ");
        for (int j = 0; j < frames; j++) {
            if (frame[j] != -1)
                printf("%d ", frame[j]);
            else
                printf("- ");
        }
        printf("\n");
    }

    double missRatio = (double)pageFaults / n;
    double hitRatio = (double)pageHits / n;

    printf("Total Page Faults = %d\n", pageFaults);
    printf("Total Page Hits = %d\n", pageHits);
    printf("Miss Ratio = %.2f\n", missRatio);
    printf("Hit Ratio = %.2f\n", hitRatio);
}
```

Output:

```
Enter the number of pages: 15
Enter the page sequence:
Page 1: 1
Page 2: 2
Page 3: 3
Page 4: 4
Page 5: 2
Page 6: 1
Page 7: 5
Page 8: 6
Page 9: 2
Page 10: 1
Page 11: 2
Page 12: 3
Page 13: 7
Page 14: 6
Page 15: 3
Enter the number of frames: 4

FIFO Page Replacement:
Frame: 1 - - -
Frame: 1 2 - -
Frame: 1 2 3 -
Frame: 1 2 3 4
Frame: 1 2 3 4
Frame: 1 2 3 4
Frame: 5 2 3 4
Frame: 5 6 3 4
Frame: 5 6 2 4
Frame: 5 6 2 1
Frame: 5 6 2 1
Frame: 3 6 2 1
Frame: 3 7 2 1
Frame: 3 7 6 1
Frame: 3 7 6 1
Total Page Faults = 11
Total Page Hits = 4
Miss Ratio = 0.73
Hit Ratio = 0.27
```

b. LRU

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

void lru(int pages[], int n, int frames);

int main() {
    int n, frames;

    printf("Enter the number of pages: ");
    scanf("%d", &n);
    int pages[n];
    printf("Enter the page sequence: \n");
    for (int i = 0; i < n; i++) {
        printf("Page %d: ", i + 1);
        scanf("%d", &pages[i]);
    }

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

    printf("\nLRU Page Replacement:\n");
    lru(pages, n, frames);

    return 0;
}

void lru(int pages[], int n, int frames) {
    int frame[frames];
    for (int i = 0; i < frames; i++)
        frame[i] = -1;

    int pageFaults = 0, pageHits = 0, time[frames];
    for (int i = 0; i < frames; i++)
        time[i] = 0;

    for (int i = 0; i < n; i++) {
        bool found = false;
        for (int j = 0; j < frames; j++) {
            if (frame[j] == pages[i]) {
                found = true;
                time[j] = i + 1;
```

```
            pageHits++;
            break;
        }
    }

    if (!found) {
        int least = 0;
        for (int j = 1; j < frames; j++) {
            if (time[j] < time[least])
                least = j;
        }
        frame[least] = pages[i];
        time[least] = i + 1;
        pageFaults++;
    }

    printf("Frame: ");
    for (int j = 0; j < frames; j++) {
        if (frame[j] != -1)
            printf("%d ", frame[j]);
        else
            printf("- ");
    }
    printf("\n");
  }

  double missRatio = (double)pageFaults / n;
  double hitRatio = (double)pageHits / n;

  printf("Total Page Faults = %d\n", pageFaults);
  printf("Total Page Hits = %d\n", pageHits);
  printf("Miss Ratio = %.2f\n", missRatio);
  printf("Hit Ratio = %.2f\n", hitRatio);
}
```

Output:

```
Enter the number of pages: 15
Enter the page sequence:
Page 1: 1
Page 2: 2
Page 3: 3
Page 4: 4
Page 5: 2
Page 6: 1
Page 7: 5
Page 8: 6
Page 9: 2
Page 10: 1
Page 11: 2
Page 12: 3
Page 13: 7
Page 14: 6
Page 15: 3
Enter the number of frames: 4

LRU Page Replacement:
Frame: 1 - - -
Frame: 1 2 - -
Frame: 1 2 3 -
Frame: 1 2 3 4
Frame: 1 2 3 4
Frame: 1 2 3 4
Frame: 1 2 5 4
Frame: 1 2 5 6
Frame: 1 2 5 6
Frame: 1 2 5 6
Frame: 1 2 5 6
Frame: 1 2 3 6
Frame: 1 2 3 7
Frame: 6 2 3 7
Frame: 6 2 3 7
Total Page Faults = 9
Total Page Hits = 6
Miss Ratio = 0.60
Hit Ratio = 0.40
```

c. Optimal :

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

void optimal(int pages[], int n, int frames);

int main() {
    int n, frames;

    printf("Enter the number of pages: ");
    scanf("%d", &n);
    int pages[n];
    printf("Enter the page sequence: \n");
    for (int i = 0; i < n; i++) {
        printf("Page %d: ", i + 1);
        scanf("%d", &pages[i]);
    }

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

    printf("\nOptimal Page Replacement:\n");
    optimal(pages, n, frames);

    return 0;
}

void optimal(int pages[], int n, int frames) {
    int frame[frames];
    for (int i = 0; i < frames; i++)
        frame[i] = -1;

    int pageFaults = 0, pageHits = 0;
    for (int i = 0; i < n; i++) {
        bool found = false;
        for (int j = 0; j < frames; j++) {
            if (frame[j] == pages[i]) {
                found = true;
                pageHits++;
                break;
            }
        }
```

```
    if (!found) {
        int replaceIndex = -1, farthest = i;
        for (int j = 0; j < frames; j++) {
            if (frame[j] == -1) {
                replaceIndex = j;
                break;
            }

            int k;
            for (k = i + 1; k < n; k++) {
                if (frame[j] == pages[k])
                    break;
            }
            if (k == n) {
                replaceIndex = j;
                break;
            } else if (k > farthest) {
                farthest = k;
                replaceIndex = j;
            }
        }
        frame[replaceIndex] = pages[i];
        pageFaults++;
    }

    printf("Frame: ");
    for (int j = 0; j < frames; j++) {
        if (frame[j] != -1)
            printf("%d ", frame[j]);
        else
            printf("- ");
    }
    printf("\n");
}

double missRatio = (double)pageFaults / n;
double hitRatio = (double)pageHits / n;

printf("Total Page Faults = %d\n", pageFaults);
printf("Total Page Hits = %d\n", pageHits);
printf("Miss Ratio = %.2f\n", missRatio);
printf("Hit Ratio = %.2f\n", hitRatio);
```

}

Output:

```
Enter the number of pages: 15
Enter the page sequence:
Page 1: 1
Page 2: 2
Page 3: 3
Page 4: 4
Page 5: 2
Page 6: 1
Page 7: 5
Page 8: 6
Page 9: 2
Page 10: 1
Page 11: 2
Page 12: 3
Page 13: 7
Page 14: 6
Page 15: 3
Enter the number of frames: 4

Optimal Page Replacement:
Frame: 1 - - -
Frame: 1 2 - -
Frame: 1 2 3 -
Frame: 1 2 3 4
Frame: 1 2 3 4
Frame: 1 2 3 4
Frame: 1 2 3 5
Frame: 1 2 3 6
Frame: 1 2 3 6
Frame: 1 2 3 6
Frame: 1 2 3 6
Frame: 1 2 3 6
Frame: 7 2 3 6
Frame: 7 2 3 6
Frame: 7 2 3 6
Total Page Faults = 7
Total Page Hits = 8
Miss Ratio = 0.47
Hit Ratio = 0.53
```

Program containing all the page replacement algorithms (FIFO , Optimal and LRU)

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

void fifo(int pages[], int n, int frames);
void optimal(int pages[], int n, int frames);
void lru(int pages[], int n, int frames);

int main() {
    int n, frames;

    printf("Enter the number of pages: ");
    scanf("%d", &n);
    int pages[n];
    printf("Enter the page sequence: \n");
    for (int i = 0; i < n; i++) {
        printf("Page %d: ", i + 1);
        scanf("%d", &pages[i]);
    }

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

    printf("\nFIFO Page Replacement:\n");
    fifo(pages, n, frames);

    printf("\nOptimal Page Replacement:\n");
    optimal(pages, n, frames);

    printf("\nLRU Page Replacement:\n");
    lru(pages, n, frames);

    return 0;
}

void fifo(int pages[], int n, int frames) {
    int frame[frames];
    for (int i = 0; i < frames; i++)
        frame[i] = -1;
```

```
    int pageFaults = 0, pageHits = 0, index = 0;
    for (int i = 0; i < n; i++) {
        bool found = false;
        for (int j = 0; j < frames; j++) {
            if (frame[j] == pages[i]) {
                found = true;
                pageHits++;
                break;
            }
        }
        if (!found) {
            frame[index] = pages[i];
            index = (index + 1) % frames;
            pageFaults++;
        }

        printf("Frame: ");
        for (int j = 0; j < frames; j++) {
            if (frame[j] != -1)
                printf("%d ", frame[j]);
            else
                printf("- ");
        }
        printf("\n");
    }

    double missRatio = (double)pageFaults / n;
    double hitRatio = (double)pageHits / n;

    printf("Total Page Faults = %d\n", pageFaults);
    printf("Total Page Hits = %d\n", pageHits);
    printf("Miss Ratio = %.2f\n", missRatio);
    printf("Hit Ratio = %.2f\n", hitRatio);
}

void optimal(int pages[], int n, int frames) {
    int frame[frames];
    for (int i = 0; i < frames; i++)
        frame[i] = -1;

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

```
        bool found = false;
        for (int j = 0; j < frames; j++) {
            if (frame[j] == pages[i]) {
                found = true;
                pageHits++;
                break;
            }
        }

        if (!found) {
            int replaceIndex = -1, farthest = i;
            for (int j = 0; j < frames; j++) {
                if (frame[j] == -1) {
                    replaceIndex = j;
                    break;
                }

                int k;
                for (k = i + 1; k < n; k++) {
                    if (frame[j] == pages[k])
                        break;
                }
                if (k == n) {
                    replaceIndex = j;
                    break;
                } else if (k > farthest) {
                    farthest = k;
                    replaceIndex = j;
                }
            }
            frame[replaceIndex] = pages[i];
            pageFaults++;
        }

        printf("Frame: ");
        for (int j = 0; j < frames; j++) {
            if (frame[j] != -1)
                printf("%d ", frame[j]);
            else
                printf("- ");
        }
        printf("\n");
    }
```

```
    double missRatio = (double)pageFaults / n;
    double hitRatio = (double)pageHits / n;

    printf("Total Page Faults = %d\n", pageFaults);
    printf("Total Page Hits = %d\n", pageHits);
    printf("Miss Ratio = %.2f\n", missRatio);
    printf("Hit Ratio = %.2f\n", hitRatio);
}

void lru(int pages[], int n, int frames) {
    int frame[frames];
    for (int i = 0; i < frames; i++)
        frame[i] = -1;

    int pageFaults = 0, pageHits = 0, time[frames];
    for (int i = 0; i < frames; i++)
        time[i] = 0;

    for (int i = 0; i < n; i++) {
        bool found = false;
        for (int j = 0; j < frames; j++) {
            if (frame[j] == pages[i]) {
                found = true;
                time[j] = i + 1;
                pageHits++;
                break;
            }
        }

        if (!found) {
            int least = 0;
            for (int j = 1; j < frames; j++) {
                if (time[j] < time[least])
                    least = j;
            }
            frame[least] = pages[i];
            time[least] = i + 1;
            pageFaults++;
        }

        printf("Frame: ");
        for (int j = 0; j < frames; j++) {
```

```
            if (frame[j] != -1)
                printf("%d ", frame[j]);
            else
                printf("- ");
        }
        printf("\n");
    }

    double missRatio = (double)pageFaults / n;
    double hitRatio = (double)pageHits / n;

    printf("Total Page Faults = %d\n", pageFaults);
    printf("Total Page Hits = %d\n", pageHits);
    printf("Miss Ratio = %.2f\n", missRatio);
    printf("Hit Ratio = %.2f\n", hitRatio);
}
```

Output:

```
Enter the number of pages: 15
Enter the page sequence:
Page 1: 1
Page 2: 2
Page 3: 3
Page 4: 4
Page 5: 2
Page 6: 1
Page 7: 5
Page 8: 6
Page 9: 2
Page 10: 1
Page 11: 2
Page 12: 3
Page 13: 7
Page 14: 6
Page 15: 3
Enter the number of frames: 4

FIFO Page Replacement:
Frame: 1 - - -
Frame: 1 2 - -
Frame: 1 2 3 -
Frame: 1 2 3 4
Frame: 1 2 3 4
Frame: 1 2 3 4
Frame: 5 2 3 4
Frame: 5 6 3 4
Frame: 5 6 2 4
Frame: 5 6 2 1
Frame: 5 6 2 1
Frame: 3 6 2 1
Frame: 3 7 2 1
Frame: 3 7 6 1
Frame: 3 7 6 1
Total Page Faults = 11
Total Page Hits = 4
Miss Ratio = 0.73
Hit Ratio = 0.27
```

```
Optimal Page Replacement:
Frame: 1 - - -
Frame: 1 2 - -
Frame: 1 2 3 -
Frame: 1 2 3 4
Frame: 1 2 3 4
Frame: 1 2 3 4
Frame: 1 2 3 5
Frame: 1 2 3 6
Frame: 1 2 3 6
Frame: 1 2 3 6
Frame: 1 2 3 6
Frame: 1 2 3 6
Frame: 7 2 3 6
Frame: 7 2 3 6
Frame: 7 2 3 6
Total Page Faults = 7
Total Page Hits = 8
Miss Ratio = 0.47
Hit Ratio = 0.53

LRU Page Replacement:
Frame: 1 - - -
Frame: 1 2 - -
Frame: 1 2 3 -
Frame: 1 2 3 4
Frame: 1 2 3 4
Frame: 1 2 3 4
Frame: 1 2 5 4
Frame: 1 2 5 6
Frame: 1 2 5 6
Frame: 1 2 5 6
Frame: 1 2 5 6
Frame: 1 2 3 6
Frame: 1 2 3 7
Frame: 6 2 3 7
Frame: 6 2 3 7
Total Page Faults = 9
Total Page Hits = 6
Miss Ratio = 0.60
Hit Ratio = 0.40
```