VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS

Submitted by

GAURAV RAMACHANDRA
(1BM22CS100)

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 **GAURAV RAMACHANDRA (1BM22CS100),** 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.

Dr. Selva Kumar S

Associate Professor
Department of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak

Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

8.	Write a C program to simulate deadlock detection	37-40
9.	Write a C program to simulate the following contiguous memory allocation techniques a) Worst-fit b) Best-fit c) First-fit	41-47
10.	Write a C program to simulate page replacement algorithms a) FIFO b) LRU c) Optimal	48-43

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

Question:

Write a C program to stimulate the following non-pre-emptive CPU scheduling algorithm to find turnaround and waiting time

1. FCFS
2. SJF (pre-emptive and non-preemptive)

Code:

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

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

void sort_by_arrival_time(Process *processes, int n) {
   // Bubble sort 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) {
               // Swap
               Process temp = processes[j];
               processes[j] = processes[j + 1];
               processes[j + 1] = temp;
           }
       }
   }
}

void compute_completion_time(Process *processes, int n) {
   int temp_timestamp = processes[0].arrival_time;
   for (int i = 0; i < n; i++) {
       if (processes[i].arrival_time > temp_timestamp) {
           // If the next process arrives after the current time, move the timestamp forward
           temp_timestamp = processes[i].arrival_time;
       }
       processes[i].completion_time = temp_timestamp + processes[i].burst_time;
       temp_timestamp = processes[i].completion_time;
   }
}

void compute_turnaround_waiting_time(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 display_table(Process *processes, int n) {
  printf("Process   Arrival Time   Burst Time   Completion Time   Turnaround Time   Waiting
Time\n");
  printf("-------   ------------   ----------   ---------------   ---------------   ------------\n");
  for (int i = 0; i < n; i++) {
      printf("  %c\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].process_name,
                                        processes[i].arrival_time,
                                        processes[i].burst_time,
                                        processes[i].completion_time,
                                        processes[i].turnaround_time,
                                        processes[i].waiting_time);
  }
}

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

  Process *processes = (Process *)malloc(n * sizeof(Process));

  // Input process details
  for (int i = 0; i < n; i++) {
      printf("Enter details for process %d (Name Arrival Burst): ", i + 1);
      scanf(" %c %d %d", &processes[i].process_name, &processes[i].arrival_time,
&processes[i].burst_time);
  }

  // Sort processes by arrival time
  sort_by_arrival_time(processes, n);

  // Compute completion time
  compute_completion_time(processes, n);

  // Compute turnaround and waiting time
  compute_turnaround_waiting_time(processes, n);

  // Display table
  display_table(processes, n);

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

Result:

```
"C:\Users\gau68\OneDrive\De

Enter details for process 3 (Name Arrival Burst): 3 3 2
Enter details for process 4 (Name Arrival Burst): 4 5 6

FCFS Scheduling:
Process    Arrival Time   Burst Time   Completion Time   Turnaround Time   Waiting Time
-------    ------------   ----------   ---------------   ---------------   ------------
  1            0              7              7                7                 0
  3            3              2              9                6                 4
  4            5              6              15               10                4
  2            8              3              18               10                7

SJF Non-Preemptive Scheduling:
Process    Arrival Time   Burst Time   Completion Time   Turnaround Time   Waiting Time
-------    ------------   ----------   ---------------   ---------------   ------------
  1            0              7              7                7                 0
  3            3              2              9                6                 4
  4            5              6              18               13                7
  2            8              3              12               4                 1

SRTF Preemptive Scheduling:
Process    Arrival Time   Burst Time   Completion Time   Turnaround Time   Waiting Time
-------    ------------   ----------   ---------------   ---------------   ------------
  1            0              7              9                9                 2
  3            3              2              5                2                 0
  4            5              6              18               13                7
  2            8              3              12               4                 1

Process returned 0 (0x0)   execution time : 18.211 s
Press any key to continue.
```

Program -2

Question:

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

1. Priority (pre-emptive & Non-pre-emptive)
2. Round Robin (Experiment with different quantum sizes for RR algorithm)

Code:

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



typedef struct {

    char process_name;

    int arrival_time;

    int burst_time;

    int priority;

    int completion_time;

    int turnaround_time;

    int waiting_time;

    int remaining_time;

} Process;



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

void compute_completion_time_priority_nonpreemptive(Process *processes, int n) {
    int current_time = 0;

    for (int i = 0; i < n; i++) {
        if (processes[i].arrival_time > current_time) {
            current_time = processes[i].arrival_time;
```

```
        }
        processes[i].completion_time = current_time + processes[i].burst_time;
        current_time = processes[i].completion_time;
    }
}

void compute_completion_time_priority_preemptive(Process *processes, int n) {
    int current_time = 0, completed = 0, min_priority, highest_priority, next_process;
    int *remaining_time = (int *)malloc(n * sizeof(int));

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

    while (completed != n) {
        min_priority = INT_MAX;
        highest_priority = -1;
        next_process = -1;
        for (int i = 0; i < n; i++) {
            if (processes[i].arrival_time <= current_time && remaining_time[i] > 0 && processes[i].priority < min_priority) {
                min_priority = processes[i].priority;
                highest_priority = processes[i].priority;
                next_process = i;
            }
```

```
        }

        if (next_process == -1) {
            current_time++;
        } else {
            remaining_time[next_process]--;
            current_time++;

            if (remaining_time[next_process] == 0) {
                processes[next_process].completion_time = current_time;
                completed++;
            }
        }
    }
    free(remaining_time);
}

void compute_turnaround_waiting_time(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 round_robin(Process *processes, int n, int quantum) {
    int *remaining_time = (int *)malloc(n * sizeof(int));
    int *completion = (int *)malloc(n * sizeof(int));
    int current_time = 0;
    int completed = 0;

    for (int i = 0; i < n; i++) {
        remaining_time[i] = processes[i].burst_time;
        completion[i] = 0;
    }
    while (completed != n) {
        int flag = 0;
        for (int i = 0; i < n; i++) {
            if (remaining_time[i] > 0) {
                flag = 1;
                if (remaining_time[i] > quantum) {
                    current_time += quantum;
                    remaining_time[i] -= quantum;
                } else {
                    current_time += remaining_time[i];
                    completion[i] = current_time;
                    remaining_time[i] = 0;
                    completed++;
                }
            }
```

```
        }

        if (flag == 0) {
            current_time++;
        }
    }

    for (int i = 0; i < n; i++) {
        processes[i].completion_time = completion[i];
    }

    free(remaining_time);
    free(completion);
}

void display_table(Process *processes, int n, const char *algorithm) {
    printf("\n%s Scheduling:\n", algorithm);
    printf("Process   Arrival Time   Burst Time   Priority   Completion Time   Turnaround
Time   Waiting Time\n");
    printf("-------   ------------   ----------   --------   ---------------   ---------------   ------------\n");
    for (int i = 0; i < n; i++) {
        printf("  %c\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].process_name,
               processes[i].arrival_time,
               processes[i].burst_time,
               processes[i].priority,
```

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

void reset_process_times(Process *processes, int n) {
    for (int i = 0; i < n; i++) {
        processes[i].completion_time = 0;
        processes[i].turnaround_time = 0;
        processes[i].waiting_time = 0;
    }
}

int main() {
    int n, quantum;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    Process *processes = (Process *)malloc(n * sizeof(Process));
    // Input process details
    for (int i = 0; i < n; i++) {
        printf("Enter details for process %d (Name Arrival Burst Priority): ", i + 1);
        scanf(" %c %d %d %d", &processes[i].process_name, &processes[i].arrival_time,
&processes[i].burst_time, &processes[i].priority);
        processes[i].completion_time = 0;
```

```
    processes[i].turnaround_time = 0;
    processes[i].waiting_time = 0;
    processes[i].remaining_time = processes[i].burst_time;
}

// Sort processes by arrival time for proper execution order
sort_by_arrival_time(processes, n);

// Priority Scheduling - Non-preemptive
compute_completion_time_priority_nonpreemptive(processes, n);
compute_turnaround_waiting_time(processes, n);
display_table(processes, n, "Priority (Non-preemptive)");
// Reset process times for next algorithm
reset_process_times(processes, n);
// Priority Scheduling - Preemptive
compute_completion_time_priority_preemptive(processes, n);
compute_turnaround_waiting_time(processes, n);
display_table(processes, n, "Priority (Preemptive)");

// Reset process times for next algorithm
reset_process_times(processes, n);

// Round Robin Scheduling with different quantum sizes
printf("\nEnter the quantum size for Round Robin scheduling: ");
scanf("%d", &quantum);
```

```
    round_robin(processes, n, quantum);

    compute_turnaround_waiting_time(processes, n);

    display_table(processes, n, "Round Robin (Quantum)");


    free(processes);

    return 0;

}
```

Result:

```
"C:\Users\gau68\OneDrive\D

-------   ------------   ----------   --------   ---------------   ---------------   ------------
  1           0              3            5             3                3                0
  2           2              2            3             5                3                1
  3           3              5            2             10               7                2
  4           4              4            4             14               10               6
  5           6              1            1             15               9                8

Priority (Preemptive) Scheduling:
Process   Arrival Time   Burst Time   Priority   Completion Time   Turnaround Time   Waiting Time
-------   ------------   ----------   --------   ---------------   ---------------   ------------
  1           0              3            5             15               15               12
  2           2              2            3             10               8                6
  3           3              5            2             9                6                1
  4           4              4            4             14               10               6
  5           6              1            1             7                1                0

Enter the quantum size for Round Robin scheduling: 3

Round Robin (Quantum) Scheduling:
Process   Arrival Time   Burst Time   Priority   Completion Time   Turnaround Time   Waiting Time
-------   ------------   ----------   --------   ---------------   ---------------   ------------
  1           0              3            5             3                3                0
  2           2              2            3             5                3                1
  3           3              5            2             14               11               6
  4           4              4            4             15               11               7
  5           6              1            1             12               6                5

Process returned 0 (0x0)   execution time : 101.899 s
Press any key to continue.
```

Program -3

Question:

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>

#include <stdlib.h>



typedef struct {

    char process_name;

    int arrival_time;

    int burst_time;

    int completion_time;

    int turnaround_time;

    int waiting_time;

} Process;



void fcfs_scheduling(Process *processes, int n) {

    int current_time = 0;



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

        if (processes[i].arrival_time > current_time) {

            current_time = processes[i].arrival_time;

        }

        processes[i].completion_time = current_time + processes[i].burst_time;
```

```
        current_time = processes[i].completion_time;
    }
}

void compute_turnaround_waiting_time(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 display_table(Process *processes, int n) {
    printf("Process   Arrival Time   Burst Time   Completion Time   Turnaround Time
Waiting Time\n");
    for (int i = 0; i < n; i++) {
        printf("  %c\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].process_name,
               processes[i].arrival_time,
               processes[i].burst_time,
               processes[i].completion_time,
               processes[i].turnaround_time,
               processes[i].waiting_time);
    }
}

int main() {
    int n_sys, n_user;
```

```
    printf("Enter the number of system processes: ");

    scanf("%d", &n_sys);

    printf("Enter the number of user processes: ");

    scanf("%d", &n_user);


    Process *sys_processes = (Process *)malloc(n_sys * sizeof(Process));

    Process *user_processes = (Process *)malloc(n_user * sizeof(Process));


    // Input system process details

    printf("\nEnter details for system processes (Name Arrival Burst):\n");

    for (int i = 0; i < n_sys; i++) {

        printf("System Process %d: ", i + 1);

        scanf(" %c %d %d", &sys_processes[i].process_name, &sys_processes[i].arrival_time,
&sys_processes[i].burst_time);

        sys_processes[i].completion_time = 0;

        sys_processes[i].turnaround_time = 0;

        sys_processes[i].waiting_time = 0;

    }


    // Input user process details

    printf("\nEnter details for user processes (Name Arrival Burst):\n");

    for (int i = 0; i < n_user; i++) {

        printf("User Process %d: ", i + 1);

        scanf(" %c %d %d", &user_processes[i].process_name,
&user_processes[i].arrival_time, &user_processes[i].burst_time);

        user_processes[i].completion_time = 0;
```

```
        user_processes[i].turnaround_time = 0;
        user_processes[i].waiting_time = 0;
    }

    // Sort processes by arrival time for each category (FCFS)
    fcfs_scheduling(sys_processes, n_sys);
    fcfs_scheduling(user_processes, n_user);

    // Compute turnaround and waiting time for system processes
    compute_turnaround_waiting_time(sys_processes, n_sys);
    // Compute turnaround and waiting time for user processes
    compute_turnaround_waiting_time(user_processes, n_user);

    // Display table for system processes
    printf("\nSystem Processes:\n");
    display_table(sys_processes, n_sys);

    // Display table for user processes
    printf("\nUser Processes:\n");
    display_table(user_processes, n_user);

    free(sys_processes);
    free(user_processes);
    return 0;
}
```

Result:

```
"C:\Users\gau68\OneDrive\D

Enter the number of system processes: 3
Enter the number of user processes: 2

Enter details for system processes (Name Arrival Burst):
System Process 1: A 0 5
System Process 2: B 1 3
System Process 3: C 2 4

Enter details for user processes (Name Arrival Burst):
User Process 1: X 0 2
User Process 2: Y 1 6

System Processes:
Process   Arrival Time   Burst Time   Completion Time   Turnaround Time   Waiting Time
  A            0              5              5                5               0
  B            1              3              8                7               4
  C            2              4              12               10              6

User Processes:
Process   Arrival Time   Burst Time   Completion Time   Turnaround Time   Waiting Time
  X            0              2              2                2               0
  Y            1              6              8                7               1

Process returned 0 (0x0)   execution time : 31.399 s
Press any key to continue.
```

Program -4

Question:

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

a) Rate- Monotonic

b) Earliest-deadline First

c) Proportional scheduling

Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>


#define MAX_PROCESSES 10

#define MAX_TIME 100


struct Process {

   int id;

   int burst_time;

   int arrival_time;

   int period;

   int deadline;

   int remaining_time;

   int share;

   int priority; // Dynamic priority for RM and EDF

};


// Function prototypes
```

```
void rateMonotonic(struct Process processes[], int n);
void earliestDeadlineFirst(struct Process processes[], int n);
void proportionalShare(struct Process processes[], int n, int total_shares);

void simulate(struct Process processes[], int n) {
    for (int time = 0; time < MAX_TIME; time++) {
        // 1. Check for arrivals
        for (int i = 0; i < n; i++) {
            if (processes[i].arrival_time == time) {
                processes[i].remaining_time = processes[i].burst_time;
            }
        }

        // 2. Determine highest priority process (algorithm-specific)
        int highestPriorityProcess = -1;
        for (int i = 0; i < n; i++) {
            if (processes[i].remaining_time > 0 && (highestPriorityProcess == -1 ||
                processes[i].priority < processes[highestPriorityProcess].priority)) {
                highestPriorityProcess = i;
            }
        }

        // 3. Execute highest priority process
        if (highestPriorityProcess != -1) {
            processes[highestPriorityProcess].remaining_time--;
```

```
            printf("Time %d: P%d\n", time, processes[highestPriorityProcess].id);


            // 4. Handle completion and periodic tasks
            if (processes[highestPriorityProcess].remaining_time == 0) {
                if (processes[highestPriorityProcess].period > 0) { // Periodic
                    processes[highestPriorityProcess].remaining_time =
processes[highestPriorityProcess].burst_time;
                    processes[highestPriorityProcess].deadline +=
processes[highestPriorityProcess].period;
                } else { // Non-periodic
                    // Mark as completed (or remove from consideration)
                }
            }
        } else {
            printf("Time %d: IDLE\n", time);
        }
    }
}


// Scheduling algorithm implementations


void rateMonotonic(struct Process processes[], int n) {
    // Smaller period -> Higher priority
    for (int i = 0; i < n; i++) {
        processes[i].priority = processes[i].period;
    }
```

```
    simulate(processes, n);
}

void earliestDeadlineFirst(struct Process processes[], int n) {
    // Earlier absolute deadline -> Higher priority
    for (int i = 0; i < n; i++) {
        processes[i].priority = processes[i].deadline;
    }
    simulate(processes, n);
}

void proportionalShare(struct Process processes[], int n, int total_shares) {
    // Simplified PS: Assign shares, calculate deadlines based on shares
    for (int i = 0; i < n; i++) {
        processes[i].share = total_shares / n; // Equal shares for simplicity
        processes[i].priority = processes[i].deadline; // Initial priority based on deadline
    }
    simulate(processes, n);
}

// ... main function (input/output handling)
int main() {
    struct Process processes[MAX_PROCESSES];
    int n, total_shares, choice;
```

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

if (n <= 0 || n > MAX_PROCESSES) {
    fprintf(stderr, "Invalid number of processes.\n");
    return 1;
}
printf("Enter details for each process:\n");
for (int i = 0; i < n; i++) {
    processes[i].id = i + 1;
    printf("Process P%d:\n", i + 1);
    printf("  Burst Time: ");
    scanf("%d", &processes[i].burst_time);
    printf("  Arrival Time: ");
    scanf("%d", &processes[i].arrival_time);

    // Determine period and deadline based on algorithm
    printf("Choose type of process (1: Periodic, 2: Non-Periodic): ");
    int processType;
    scanf("%d", &processType);

    if (processType == 1) {
        printf("  Period: ");
        scanf("%d", &processes[i].period);
        processes[i].deadline = processes[i].arrival_time + processes[i].period;
```

```
    } else {
        processes[i].period = 0;
        printf("  Deadline: ");
        scanf("%d", &processes[i].deadline);
    }
}
printf("\nChoose a scheduling algorithm:\n");
printf("1. Rate Monotonic\n");
printf("2. Earliest Deadline First\n");
printf("3. Proportional Share\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
    case 1:
        rateMonotonic(processes, n);
        break;
    case 2:
        earliestDeadlineFirst(processes, n);
        break;
    case 3:
        printf("Enter total number of shares: ");
        scanf("%d", &total_shares);
        proportionalShare(processes, n, total_shares);
        break;
```

```
        default:

            fprintf(stderr, "Invalid choice.\n");

            return 1;

    }



    return 0;

}
```

Result:

```
"C:\Users\gau68\OneDrive\De
Enter the number of processes (max 10): 3
Enter details for each process:
Process P1:
  Burst Time: 3
  Arrival Time: 0
Choose type of process (1: Periodic, 2: Non-Periodic): 1
  Period: 8
Process P2:
  Burst Time: 2
  Arrival Time: 1
Choose type of process (1: Periodic, 2: Non-Periodic): 1
  Period: 5
Process P3:
  Burst Time: 4
  Arrival Time: 2
Choose type of process (1: Periodic, 2: Non-Periodic): 2
  Deadline: 7

Choose a scheduling algorithm:
1. Rate Monotonic
2. Earliest Deadline First
3. Proportional Share
Enter your choice: 1
Time 0: P1
Time 1: P2
Time 2: P3
Time 3: P3
Time 4: P3
Time 5: P3
Time 6: P2
Time 7: P2
Time 8: P2
Time 9: P2
Time 10: P2
Time 11: P2
Time 12: P2
Time 13: P2
Time 14: P2
Time 15: P2
Time 16: P2
Time 17: P2
```

Program -5

Question:

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

Code:

```
#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>


#define BUFFER_SIZE 5


int buffer[BUFFER_SIZE];

int in = 0, out = 0;

sem_t empty, full;

pthread_mutex_t mutex;


void *producer(void *param) {

    int item;

    while (1) {

        item = rand() % 100; // Produce a random item

        sem_wait(&empty);

        pthread_mutex_lock(&mutex);

        // Add item to the buffer

        buffer[in] = item;
```

```
    }

        printf("Producer produced %d\n", item);
        in = (in + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex);
        sem_post(&full);

        sleep(1); // Sleep for a while
    }
}

void *consumer(void *param) {
    int item;
    while (1) {
        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        // Remove item from the buffer
        item = buffer[out];
        printf("Consumer consumed %d\n", item);
        out = (out + 1) % BUFFER_SIZE;
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
        sleep(1); // Sleep for a while
    }
}
```

```
int main() {
    pthread_t tid1, tid2;

    // Initialize the semaphores
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);

    // Initialize the mutex
    pthread_mutex_init(&mutex, NULL);

    // Create the producer and consumer threads
    pthread_create(&tid1, NULL, producer, NULL);
    pthread_create(&tid2, NULL, consumer, NULL);

    // Wait for the threads to finish
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    // Destroy the semaphores and mutex
    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

Result:

```
"C:\Users\gau68\OneDrive\De
Producer produced 41
Consumer consumed 41
Producer produced 67
Consumer consumed 67
Producer produced 34
Consumer consumed 34
Producer produced 0
Consumer consumed 0
Producer produced 69
Consumer consumed 69
Producer produced 24
Consumer consumed 24
Producer produced 78
Consumer consumed 78
Producer produced 58
Consumer consumed 58
Producer produced 62
Consumer consumed 62
Producer produced 64
Consumer consumed 64
Producer produced 5
Consumer consumed 5
Producer produced 45
Consumer consumed 45
Producer produced 81
Consumer consumed 81
Producer produced 27
Consumer consumed 27
Producer produced 61
Consumer consumed 61
Producer produced 91
Consumer consumed 91
Producer produced 95
Consumer consumed 95
Producer produced 42
Consumer consumed 42
Producer produced 27
Consumer consumed 27
```

Program -6

Question:

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

Code:

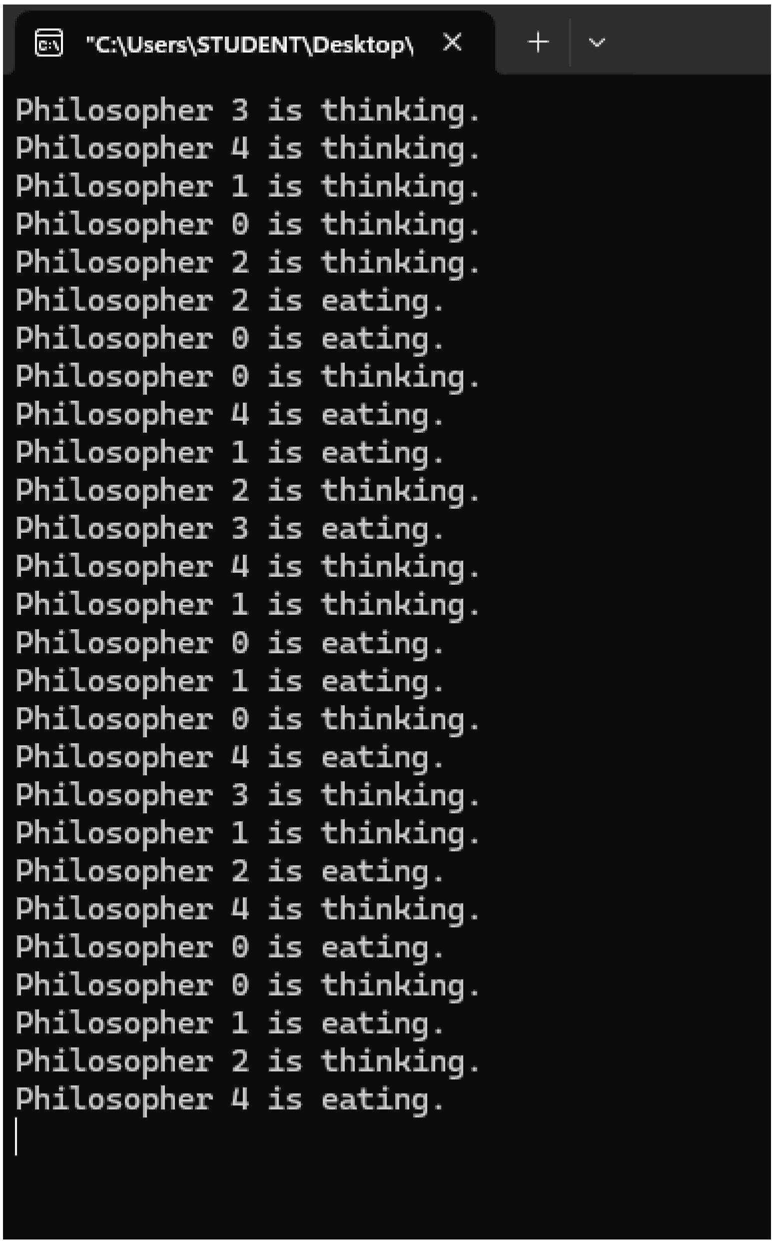
```
#include <stdio.h>

#include <pthread.h>

#include <unistd.h>


#define N 5  // Number of philosophers


// Enum to define states of a philosopher

typedef enum { THINKING, HUNGRY, EATING } state_t;


// Array to hold the states of all philosophers

state_t state[N];


// Condition variables for each philosopher

pthread_cond_t self[N];


// Mutex for critical sections

pthread_mutex_t mutex;


// Function prototypes

void *philosopher(void *arg);

void pickup(int i);

void putdown(int i);

void test(int i);

void initialization_code();


void initialization_code() {

   int i;
```

```
    // Initialize all philosophers as thinking
    for (i = 0; i < N; i++)
        state[i] = THINKING;
}

void pickup(int i) {
    pthread_mutex_lock(&mutex);
    state[i] = HUNGRY;
    test(i);
    if (state[i] != EATING)
        pthread_cond_wait(&self[i], &mutex);
    pthread_mutex_unlock(&mutex);
}

void putdown(int i) {
    pthread_mutex_lock(&mutex);
    state[i] = THINKING;
    test((i + 4) % N);
    test((i + 1) % N);
    pthread_mutex_unlock(&mutex);
}

void test(int i) {
    if (state[(i + 4) % N] != EATING &&
        state[i] == HUNGRY &&
        state[(i + 1) % N] != EATING) {
        state[i] = EATING;
        pthread_cond_signal(&self[i]);
    }
}
```

```
void *philosopher(void *arg) {
    int id = *((int *) arg);
    while (1) {
        printf("Philosopher %d is thinking.\n", id);
        sleep(1); // Thinking
        pickup(id);
        printf("Philosopher %d is eating.\n", id);
        sleep(2); // Eating
        putdown(id);
    }
}

int main() {
    pthread_t tid[N];
    int i;
    int ids[N];
    // Initialize mutex and condition variables
    pthread_mutex_init(&mutex, NULL);
    for (i = 0; i < N; i++)
        pthread_cond_init(&self[i], NULL);
    // Initialize philosophers' states
    initialization_code();

    // Create philosopher threads
    for (i = 0; i < N; i++) {
        ids[i] = i;
        pthread_create(&tid[i], NULL, philosopher, &ids[i]);
    }

    // Join threads
    for (i = 0; i < N; i++)
```

```
        pthread_join(tid[i], NULL);
    // Cleanup
    pthread_mutex_destroy(&mutex);
    for (i = 0; i < N; i++)
        pthread_cond_destroy(&self[i]);


    return 0;
}
```

Result:



```
"C:\Users\STUDENT\Desktop\
Philosopher 3 is thinking.
Philosopher 4 is thinking.
Philosopher 1 is thinking.
Philosopher 0 is thinking.
Philosopher 2 is thinking.
Philosopher 2 is eating.
Philosopher 0 is eating.
Philosopher 0 is thinking.
Philosopher 4 is eating.
Philosopher 1 is eating.
Philosopher 2 is thinking.
Philosopher 3 is eating.
Philosopher 4 is thinking.
Philosopher 1 is thinking.
Philosopher 0 is eating.
Philosopher 1 is eating.
Philosopher 0 is thinking.
Philosopher 4 is eating.
Philosopher 3 is thinking.
Philosopher 1 is thinking.
Philosopher 2 is eating.
Philosopher 4 is thinking.
Philosopher 0 is eating.
Philosopher 0 is thinking.
Philosopher 1 is eating.
Philosopher 2 is thinking.
Philosopher 4 is eating.
```

Program -7

Question:

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

Code:

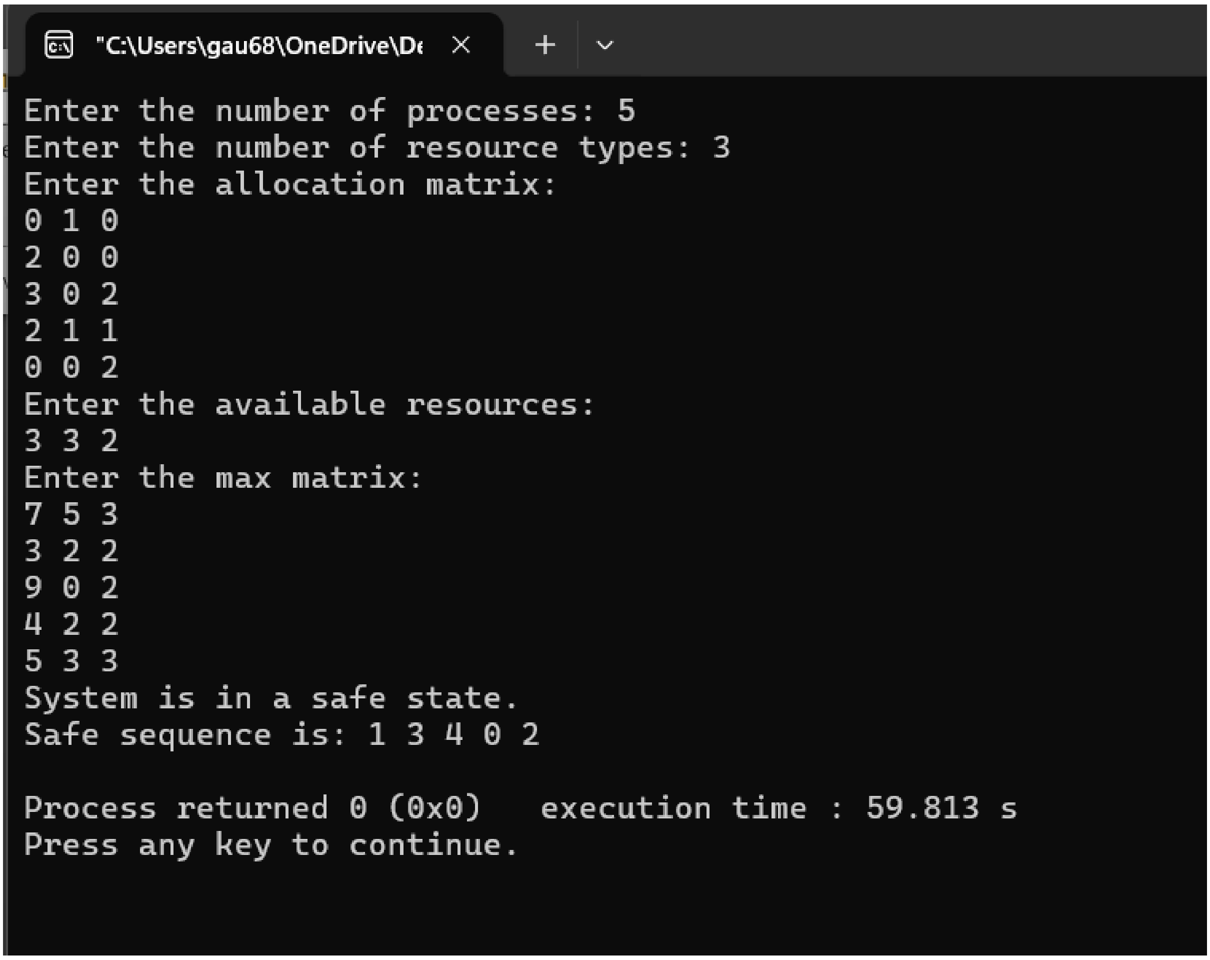
```
#include <stdio.h>

#include <stdbool.h>


// Function to check if the system is in a safe state

bool isSafeState(int n, int m, int available[], int max[][m], int allocation[][m], int need[][m], int
safeSequence[]) {

   bool finish[n];

   int work[m];

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

      work[i] = available[i];

   }

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

      finish[i] = false;

   }


   int count = 0;

   while (count < n) {

      bool found = false;

      for (int p = 0; p < n; p++) {

         if (finish[p] == false) {

            int j;

            for (j = 0; j < m; j++) {

               if (need[p][j] > work[j])

                  break;

            }

            if (j == m) {

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

```
                    work[k] += allocation[p][k];
                }
                safeSequence[count++] = p;
                finish[p] = true;
                found = true;
            }
        }
    }
    if (found == false) {
        return false;
    }
  }
  return true;
}

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

  int allocation[n][m], max[n][m], available[m];
  int need[n][m], safeSequence[n];

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

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

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



// Calculate the need matrix
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        need[i][j] = max[i][j] - allocation[i][j];
    }
}

if (isSafeState(n, m, available, max, allocation, need, safeSequence)) {
    printf("System is in a safe state.\nSafe sequence is: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", safeSequence[i]);
    }
    printf("\n");
} else {
    printf("System is not in a safe state.\n");
}
```

```
    return 0;
}
```

Result:



```
"C:\Users\gau68\OneDrive\De
Enter the number of processes: 5
Enter the number of resource types: 3
Enter the allocation matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter the available resources:
3 3 2
Enter the max matrix:
7 5 3
3 2 2
9 0 2
4 2 2
5 3 3
System is in a safe state.
Safe sequence is: 1 3 4 0 2

Process returned 0 (0x0)   execution time : 59.813 s
Press any key to continue.
```

Program -8

Question:

Write a C program to simulate deadlock detection

Code:

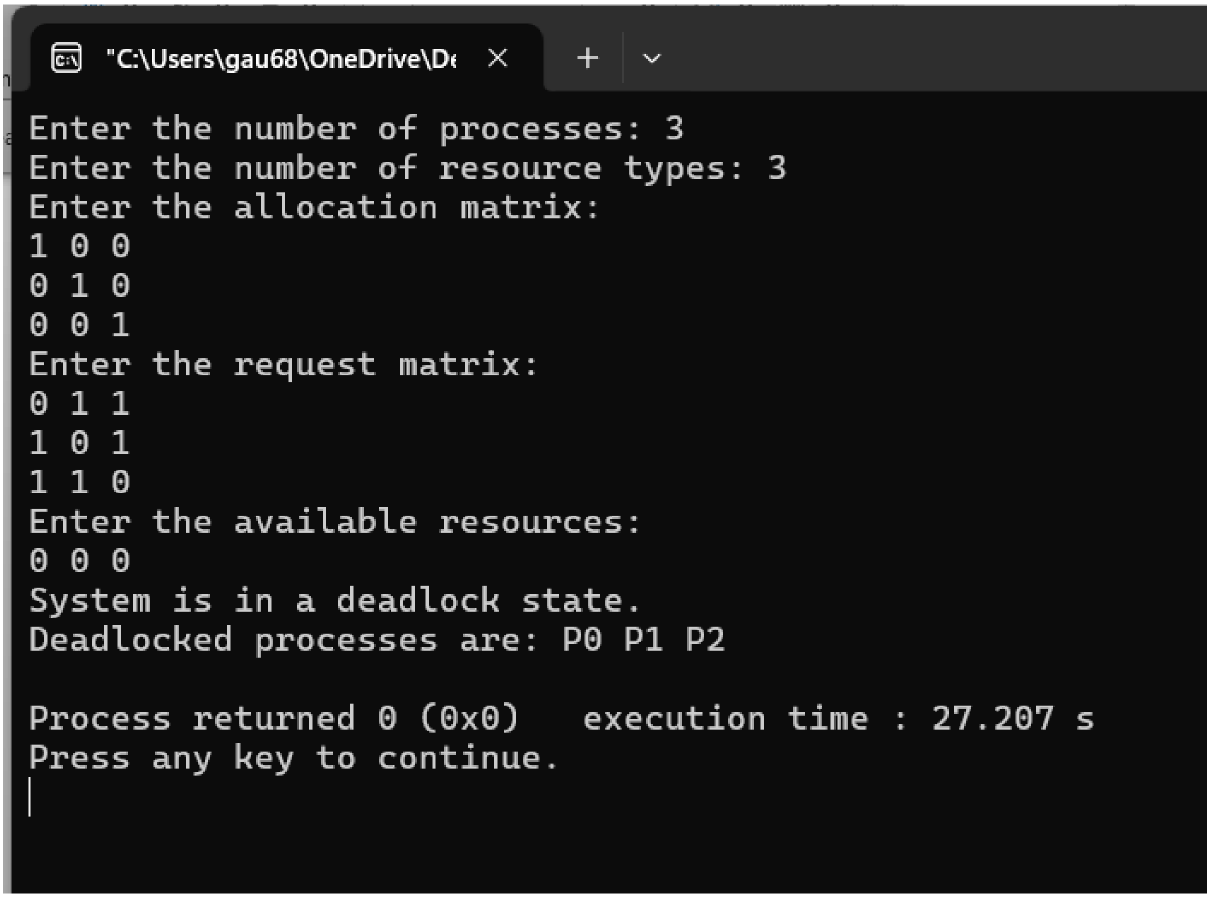
```
#include <stdio.h>

#include <stdbool.h>


// Function to check for deadlock in the system

bool detectDeadlock(int n, int m, int available[], int allocation[][m], int request[][m], int work[], bool finish[]) {

  // Initialize the work vector and finish vector

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

    work[i] = available[i];

  }

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

    finish[i] = false;

  }


  // Main deadlock detection algorithm

  while (true) {

    bool found = false;

    for (int p = 0; p < n; p++) {

      if (!finish[p]) {

        bool possible = true;

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

          if (request[p][j] > work[j]) {

            possible = false;

            break;

          }

        }

        if (possible) {
```

```
                for (int j = 0; j < m; j++) {
                    work[j] += allocation[p][j];
                }
                finish[p] = true;
                found = true;
            }
        }
    }
    if (!found) {
        break;
    }
}

// Check for any unfinished process indicating deadlock
for (int i = 0; i < n; i++) {
    if (!finish[i]) {
        return true;
    }
}
return false;
}

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

    int allocation[n][m], request[n][m], available[m];
    int work[m];
```

```c
bool finish[n];

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

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

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

if (detectDeadlock(n, m, available, allocation, request, work, finish)) {
    printf("System is in a deadlock state.\n");
    printf("Deadlocked processes are: ");
    for (int i = 0; i < n; i++) {
        if (!finish[i]) {
            printf("P%d ", i);
        }
    }
    printf("\n");
} else {
```

```
        printf("System is not in a deadlock state.\n");
    }

    return 0;
}
```

Result:



```
"C:\Users\gau68\OneDrive\De
Enter the number of processes: 3
Enter the number of resource types: 3
Enter the allocation matrix:
1 0 0
0 1 0
0 0 1
Enter the request matrix:
0 1 1
1 0 1
1 1 0
Enter the available resources:
0 0 0
System is in a deadlock state.
Deadlocked processes are: P0 P1 P2

Process returned 0 (0x0)   execution time : 27.207 s
Press any key to continue.
```

Program -9

Question:

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

a) Worst-fit

b) Best-fit

c) First-fit

Code:

```
#include &lt;stdio.h&gt;

#include &lt;stdlib.h&gt;


#define MAX 25


void firstFit(int nb, int nf, int b[], int f[]) {

int ff[MAX] = {0};

int allocated[MAX] = {0};

for (int i = 0; i &lt; nf; i++) {

ff[i] = -1;

for (int j = 0; j &lt; nb; j++) {

if (allocated[j] == 0 &amp;&amp; b[j] &gt;= f[i]) {

ff[i] = j;

allocated[j] = 1;

break;

}

}

}
```

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

void bestFit(int nb, int nf, int b[], int f[]) {

int ff[MAX] = {0};
int allocated[MAX] = {0};
for (int i = 0; i &lt; nf; i++) {
int best = -1;
ff[i] = -1;
for (int j = 0; j &lt; nb; j++) {
if (allocated[j] == 0 &amp;&amp; b[j] &gt;= f[i]) {
if (best == -1 || b[j] &lt; b[best])
best = j;
}
}
if (best != -1) {
ff[i] = best;
allocated[best] = 1;
```

```
}

}


printf(&quot;\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:&quot;);

for (int i = 0; i &lt; nf; i++) {

if (ff[i] != -1)

printf(&quot;\n%d\t\t%d\t\t%d\t\t%d&quot;, i + 1, f[i], ff[i] + 1, b[ff[i]]);

else

printf(&quot;\n%d\t\t%d\t\t-\t\t-&quot;, i + 1, f[i]);

}

}


void worstFit(int nb, int nf, int b[], int f[]) {

int ff[MAX] = {0};

int allocated[MAX] = {0};


for (int i = 0; i &lt; nf; i++) {


int worst = -1;

ff[i] = -1;

for (int j = 0; j &lt; nb; j++) {

if (allocated[j] == 0 &amp;&amp; b[j] &gt;= f[i]) {

if (worst == -1 || b[j] &gt; b[worst])

worst = j;

}
```

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

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

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

printf(&quot;Memory Management Scheme&quot;);
printf(&quot;\nEnter the number of blocks: &quot;);
scanf(&quot;%d&quot;, &amp;nb);
printf(&quot;Enter the number of files: &quot;);
scanf(&quot;%d&quot;, &amp;nf);
```

```
int b[nb], f[nf];

printf(&quot;\nEnter the size of the blocks:\n&quot;);

for (int i = 0; i &lt; nb; i++) {

printf(&quot;Block %d: &quot;, i + 1);

scanf(&quot;%d&quot;, &amp;b[i]);

}

printf(&quot;Enter the size of the files:\n&quot;);

for (int i = 0; i &lt; nf; i++) {

printf(&quot;File %d: &quot;, i + 1);

scanf(&quot;%d&quot;, &amp;f[i]);

}


while (1) {

printf(&quot;\n1. First Fit\n2. Best Fit\n3. Worst Fit\n4. Exit\n&quot;);

printf(&quot;Enter your choice: &quot;);

scanf(&quot;%d&quot;, &amp;choice);

switch (choice) {

case 1:

printf(&quot;\n\tMemory Management Scheme - First Fit\n&quot;);

firstFit(nb, nf, b, f);

break;

case 2:

printf(&quot;\n\tMemory Management Scheme - Best Fit\n&quot;);

bestFit(nb, nf, b, f);

break;
```

```
case 3:

printf(&quot;\n\tMemory Management Scheme - Worst Fit\n&quot;);

worstFit(nb, nf, b, f);

break;

case 4:

printf(&quot;\nExiting...\n&quot;);


exit(0);

break;

default:

printf(&quot;\nInvalid choice.\n&quot;);

break;

}

}


return 0;

}
```

Result:

```
"C:\Users\STUDENT\Desktop\

Memory Management Scheme
Enter the number of blocks: 4
Enter the number of files: 3

Enter the size of the blocks:
Block 1: 5
Block 2: 7
Block 3: 4
Block 4: 9
Enter the size of the files:
File 1: 4
File 2: 3
File 3: 6

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

        Memory Management Scheme - First Fit

File_no:        File_size :     Block_no:       Block_size:
1               4               1               5
2               3               2               7
3               6               4               9
1. First Fit
2. Best Fit
3. Worst Fit
4. Exit
Enter your choice: 2

        Memory Management Scheme - Best Fit

File_no:        File_size :     Block_no:       Block_size:
1               4               3               4
2               3               1               5
3               6               2               7
1. First Fit
2. Best Fit
3. Worst Fit
4. Exit
Enter your choice: 3

        Memory Management Scheme - Worst Fit

File_no:        File_size :     Block_no:       Block_size:
1               4               4               9
2               3               2               7
3               6               -               -
1. First Fit
2. Best Fit
3. Worst Fit
4. Exit
Enter your choice:
```

Program -10

Question:

Write a C program to simulate page replacement algorithms

a) FIFO

b) LRU

c) Optimal

Code:

```
#include <stdio.h>

#include <stdlib.h>


#define MAX_FRAMES 10

#define MAX_PAGES 100


void printFrames(int frames[], int num_frames) {

    for (int i = 0; i < num_frames; i++) {

        if (frames[i] == -1)

            printf("- ");

        else

            printf("%d ", frames[i]);

    }

    printf("\n");

}


int isPageInFrames(int frames[], int num_frames, int page) {

    for (int i = 0; i < num_frames; i++) {

        if (frames[i] == page)
```

```
        return 1;
    }
    return 0;
}

void fifo(int pages[], int num_pages, int num_frames) {
    int frames[MAX_FRAMES], faults = 0, index = 0;
    for (int i = 0; i < num_frames; i++)
        frames[i] = -1;

    printf("FIFO Page Replacement:\n");
    for (int i = 0; i < num_pages; i++) {
        if (!isPageInFrames(frames, num_frames, pages[i])) {
            frames[index] = pages[i];
            index = (index + 1) % num_frames;
            faults++;
            printFrames(frames, num_frames);
        }
    }
    printf("Total Page Faults: %d\n\n", faults);
}

void lru(int pages[], int num_pages, int num_frames) {
    int frames[MAX_FRAMES], faults = 0, time[MAX_FRAMES];
    for (int i = 0; i < num_frames; i++)
```

```
    frames[i] = -1;

printf("LRU Page Replacement:\n");
for (int i = 0; i < num_pages; i++) {
    int found = 0, lru = 0;

    for (int j = 0; j < num_frames; j++) {
        if (frames[j] == pages[i]) {
            found = 1;
            time[j] = i;
            break;
        }
    }

    if (!found) {
        for (int j = 1; j < num_frames; j++) {
            if (time[j] < time[lru])
                lru = j;
        }
        frames[lru] = pages[i];
        time[lru] = i;
        faults++;
        printFrames(frames, num_frames);
    }
}
```

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

void optimal(int pages[], int num_pages, int num_frames) {
    int frames[MAX_FRAMES], faults = 0;
    for (int i = 0; i < num_frames; i++)
        frames[i] = -1;

    printf("Optimal Page Replacement:\n");
    for (int i = 0; i < num_pages; i++) {
        if (!isPageInFrames(frames, num_frames, pages[i])) {
            int replace = 0, farthest = i + 1;
            for (int j = 0; j < num_frames; j++) {
                int k;
                for (k = i + 1; k < num_pages; k++) {
                    if (frames[j] == pages[k]) {
                        if (k > farthest) {
                            farthest = k;
                            replace = j;
                        }
                        break;
                    }
                }
                if (k == num_pages) {
                    replace = j;
```

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

int main() {
    int num_pages, num_frames, pages[MAX_PAGES];

    printf("Enter the number of pages: ");
    scanf("%d", &num_pages);
    printf("Enter the pages: ");
    for (int i = 0; i < num_pages; i++)
        scanf("%d", &pages[i]);

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

    fifo(pages, num_pages, num_frames);
    lru(pages, num_pages, num_frames);
```

```
    optimal(pages, num_pages, num_frames);



    return 0;
}
```

Result:

```
"C:\Users\gau68\OneDrive\De
Enter the number of pages: 6
Enter the pages: 1 2 4 5 6 8
Enter the number of frames: 4
FIFO Page Replacement:
1 - - -
1 2 - -
1 2 4 -
1 2 4 5
6 2 4 5
6 8 4 5
Total Page Faults: 6

LRU Page Replacement:
- 1 - -
- 2 - -
- 2 - 4
- 5 - 4
- 5 - 6
- 8 - 6
Total Page Faults: 6

Optimal Page Replacement:
1 - - -
2 - - -
4 - - -
5 - - -
6 - - -
8 - - -
Total Page Faults: 6


Process returned 0 (0x0)   execution time : 21.291 s
Press any key to continue.
```