VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT

on

OPERATING SYSTEMS

Submitted by

VARSHA A Y (1WA23CS032)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)

BENGALURU-560019

Feb-2025 to June-2025

B. M. S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "OPERATING SYSTEMS – 23CS4PCOPS" carried out by Student name (USN), who is Bonafide student of B. M. S. College of Engineering. It is in partial fulfilment for the award of Bachelor of Engineering in Computer Science and Engineering of the Visvesvaraya Technological University, Belgaum during the year Feb 2025- June 2025. The Lab report has been approved as it satisfies the academic requirements in respect of a OPERATING SYSTEMS - (23CS4PCOPS) work prescribed for the said degree.

Faculty Incharge Name	Dr. Kavitha Sooda
Assistant Professor	Professor and Head
Department of CSE	Department of CSE
BMSCE, Bengaluru	BMSCE, Bengaluru

Index Sheet

Course Outcomes

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

Program-1

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

FCFS:

```
#include <stdio.h>

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

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

void fcfs(Process p[], int n, float *avgTAT, float *avgWT) {
    sortByArrival(p, n);
    int time = 0, totalTAT = 0, totalWT = 0;
    for (int i = 0; i < n; i++) {
        if (time < p[i].arrival) time = p[i].arrival;
        p[i].completion = time + p[i].burst;
        p[i].turnaround = p[i].completion - p[i].arrival;
        p[i].waiting = p[i].turnaround - p[i].burst;
        time = p[i].completion;
        totalTAT += p[i].turnaround;
```

```
        totalWT += p[i].waiting;
    }
    *avgTAT = (float)totalTAT / n;
    *avgWT = (float)totalWT / n;
}

void display(Process p[], int n, float avgTAT, float avgWT) {
    printf("\nPID Arrival Burst Completion Turnaround Waiting\n");
    for (int i = 0; i < n; i++) {
        printf("%3d %7d %6d %10d %10d %8d\n", p[i].id, p[i].arrival, p[i].burst,
p[i].completion, p[i].turnaround, p[i].waiting);
    }
    printf("\nAverage Turnaround Time: %.2f", avgTAT);
    printf("\nAverage Waiting Time: %.2f\n", avgWT);
}

int main() {
    int n;
    float avgTAT, avgWT;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    Process p[n];

    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        p[i].id = i + 1;
        printf("P[%d]: ", i + 1);
        scanf("%d %d", &p[i].arrival, &p[i].burst);
    }

    printf("(FCFS)");
    fcfs(p, n, &avgTAT, &avgWT);
    display(p, n, avgTAT, avgWT);

    return 0;

}
```

OUTPUT:

```
Enter number of processes: 4
Enter Arrival Time and Burst Time for each process:
P[1]: 0 7
P[2]: 0 3
P[3]: 0 4
P[4]: 0 6
(FCFS)
PID Arrival Burst Completion Turnaround Waiting
  1       0      7          7          7       0
  2       0      3         10         10       7
  3       0      4         14         14      10
  4       0      6         20         20      14

Average Turnaround Time: 12.75
Average Waiting Time: 7.75

Process returned 0 (0x0)   execution time : 27.578 s
Press any key to continue.
```

SJF PREEMPTIVE:

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

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

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

```
    }
}

void sjf_preemptive(Process p[], int n, float *avgTAT, float *avgWT) {
    int completed = 0, time = 0, minIdx, totalTAT = 0, totalWT = 0;
    int isCompleted[n];
    for (int i = 0; i < n; i++) {
        isCompleted[i] = 0;
        p[i].remaining = p[i].burst;
    }

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

        p[minIdx].remaining--;
        time++;
        if (p[minIdx].remaining == 0) {
            p[minIdx].completion = time;
            p[minIdx].turnaround = p[minIdx].completion - p[minIdx].arrival;
            p[minIdx].waiting = p[minIdx].turnaround - p[minIdx].burst;
            isCompleted[minIdx] = 1;
            totalTAT += p[minIdx].turnaround;
            totalWT += p[minIdx].waiting;
            completed++;
        }
    }
    *avgTAT = (float)totalTAT / n;
    *avgWT = (float)totalWT / n;
}

void display(Process p[], int n, float avgTAT, float avgWT) {
```

```
    printf("\nPID Arrival Burst Completion Turnaround Waiting\n");
    for (int i = 0; i < n; i++) {
        printf("%3d %7d %6d %10d %10d %8d\n", p[i].id, p[i].arrival, p[i].burst,
p[i].completion, p[i].turnaround, p[i].waiting);
    }
    printf("\nAverage Turnaround Time: %.2f", avgTAT);
    printf("\nAverage Waiting Time: %.2f\n", avgWT);
}

int main() {
    int n;
    float avgTAT, avgWT;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    Process p[n];

    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        p[i].id = i + 1;
        printf("P[%d]: ", i + 1);
        scanf("%d %d", &p[i].arrival, &p[i].burst);
    }

    printf("\nShortest Job First (Preemptive) Scheduling\n");
    sjf_preemptive(p, n, &avgTAT, &avgWT);
    display(p, n, avgTAT, avgWT);

    return 0;
}
```

OUTPUT:

```
Enter number of processes: 4
Enter Arrival Time and Burst Time for each process:
P[1]: 0 7
P[2]: 8 3
P[3]: 3 4
P[4]: 5 6

Shortest Job First (Preemptive) Scheduling

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

Average Turnaround Time: 9.00
Average Waiting Time: 4.00

Process returned 0 (0x0)   execution time : 74.341 s
Press any key to continue.
```

SJF NON-PREEMPTIVE:

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

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

```
void sjf_non_preemptive(Process p[], int n, float *avgTAT, float *avgWT) {
    int completed = 0, time = 0, minIdx, totalTAT = 0, totalWT = 0;
    int isCompleted[n];
    for (int i = 0; i < n; i++) isCompleted[i] = 0;

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

        p[minIdx].completion = time + p[minIdx].burst;
        p[minIdx].turnaround = p[minIdx].completion - p[minIdx].arrival;
        p[minIdx].waiting = p[minIdx].turnaround - p[minIdx].burst;
        time = p[minIdx].completion;
        isCompleted[minIdx] = 1;
        totalTAT += p[minIdx].turnaround;
        totalWT += p[minIdx].waiting;
        completed++;
    }
    *avgTAT = (float)totalTAT / n;
    *avgWT = (float)totalWT / n;
}

void display(Process p[], int n, float avgTAT, float avgWT) {
    printf("\nPID Arrival Burst Completion Turnaround Waiting\n");
    for (int i = 0; i < n; i++) {
        printf("%3d %7d %6d %10d %10d %8d\n", p[i].id, p[i].arrival, p[i].burst,
p[i].completion, p[i].turnaround, p[i].waiting);
    }
    printf("\nAverage Turnaround Time: %.2f", avgTAT);
    printf("\nAverage Waiting Time: %.2f\n", avgWT);
}
```

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

    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        p[i].id = i + 1;
        printf("P[%d]: ", i + 1);
        scanf("%d %d", &p[i].arrival, &p[i].burst);
    }

    printf("\nShortest Job First (Non-Preemptive) Scheduling\n");
    sjf_non_preemptive(p, n, &avgTAT, &avgWT);
    display(p, n, avgTAT, avgWT);

    return 0;
}
```

OUTPUT:

```
Enter number of processes: 4
Enter Arrival Time and Burst Time for each process:
P[1]: 0 7
P[2]: 0 3
P[3]: 0 4
P[4]: 0 6

Shortest Job First (Non-Preemptive) Scheduling

PID Arrival Burst Completion Turnaround Waiting
  1       0      7         20         20       13
  2       0      3          3          3        0
  3       0      4          7          7        3
  4       0      6         13         13        7

Average Turnaround Time: 10.75
Average Waiting Time: 5.75

Process returned 0 (0x0)   execution time : 29.969 s
Press any key to continue.
```

Program-2

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

PRIORITY PREEMPTIVE AND NON-PREEMPTIVE:

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

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

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

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

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

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

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

        if (p[selected].remaining_bt == 0) {
            p[selected].ct = time;
            p[selected].tat = p[selected].ct - p[selected].at;
            p[selected].wt = p[selected].tat - p[selected].bt;
            completed++;
        }
    }
}
void displayProcesses(Process p[], int n) {
    float avg_tat = 0, avg_wt = 0, avg_rt = 0;
    printf("\nPID\tAT\tBT\tPriority\tCT\tTAT\tWT\tRT\n");
    for (int i = 0; i < n; i++) {
```

```
        printf("%d\t%d\t%d\t%d\t\t%d\t%d\t%d\t%d\n",
               p[i].pid, p[i].at, p[i].bt, p[i].pt, p[i].ct, p[i].tat, p[i].wt, p[i].rt);
        avg_tat += p[i].tat;
        avg_wt += p[i].wt;
        avg_rt += p[i].rt;
    }
    printf("\nAverage TAT: %.2f", avg_tat / n);
    printf("\nAverage WT: %.2f", avg_wt / n);
    printf("\nAverage RT: %.2f\n", avg_rt / n);
}
int main() {
    Process p[MAX];
    int n, choice;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("\nEnter Arrival Time, Burst Time, and Priority for Process %d:\n", p[i].pid);
        printf("Arrival Time: ");
        scanf("%d", &p[i].at);
        printf("Burst Time: ");
        scanf("%d", &p[i].bt);
        printf("Priority (lower number means higher priority): ");
        scanf("%d", &p[i].pt);
        p[i].remaining_bt = p[i].bt;
        p[i].is_completed = 0;
        p[i].rt = -1;
    }
    while (1) {
        printf("\nPriority Scheduling Menu:\n");
        printf("1. Non-Preemptive Priority Scheduling\n");
        printf("2. Preemptive Priority Scheduling\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                nonPreemptivePriority(p, n);
                printf("Non-Preemptive Scheduling Completed!\n");
```

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

OUTPUT:

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

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

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

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

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

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

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

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

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

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

ROUND ROBIN:

```
#include <stdio.h>

#define MAX 100

void roundRobin(int n, int at[], int bt[], int quant) {
    int ct[n], tat[n], wt[n], rem_bt[n];
    int queue[MAX], front = 0, rear = 0;
    int time = 0, completed = 0, visited[n];

    for (int i = 0; i < n; i++) {
        rem_bt[i] = bt[i];
        visited[i] = 0;
    }
```

```
queue[rear++] = 0;
visited[0] = 1;

while (completed < n) {
    int index = queue[front++];

    if (rem_bt[index] > quant) {
        time += quant;
        rem_bt[index] -= quant;
    } else {
        time += rem_bt[index];
        rem_bt[index] = 0;
        ct[index] = time;
        completed++;
    }

    for (int i = 0; i < n; i++) {
        if (at[i] <= time && rem_bt[i] > 0 && !visited[i]) {
            queue[rear++] = i;
            visited[i] = 1;
        }
    }

    if (rem_bt[index] > 0) {
        queue[rear++] = index;
    }

    if (front == rear) {
        for (int i = 0; i < n; i++) {
            if (rem_bt[i] > 0) {
                queue[rear++] = i;
                visited[i] = 1;
                break;
            }
        }
    }
}

float total_tat = 0, total_wt = 0;
printf("P#\tAT\tBT\tCT\tTAT\tWT\n");
```

```
    for (int i = 0; i < n; i++) {
        tat[i] = ct[i] - at[i];
        wt[i] = tat[i] - bt[i];
        total_tat += tat[i];
        total_wt += wt[i];
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", i + 1, at[i], bt[i], ct[i], tat[i], wt[i]);
    }

    printf("Average TAT: %.2f\n", total_tat / n);
    printf("Average WT: %.2f\n", total_wt / n);
}

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

    int at[n], bt[n];
    for (int i = 0; i < n; i++) {
        printf("Enter AT and BT for process %d: ", i + 1);
        scanf("%d %d", &at[i], &bt[i]);
    }

    printf("Enter time quantum: ");
    scanf("%d", &quant);

    roundRobin(n, at, bt, quant);
    return 0;
}
```

OUTPUT:

```
PS C:\Users\trish\OneDrive\Desktop\OS_LAB> gcc Round.c -o Round
PS C:\Users\trish\OneDrive\Desktop\OS_LAB> ./Round
Enter number of processes: 5
Enter AT and BT for process 1: 0
8
Enter AT and BT for process 2: 5
2
Enter AT and BT for process 3: 1
7
Enter AT and BT for process 4: 6
3
Enter AT and BT for process 5: 8
5
Enter time quantum: 3
P#      AT      BT      CT      TAT     WT
1       0       8       22      22      14
2       5       2       11      6       4
3       1       7       23      22      15
4       6       3       14      8       5
5       8       5       25      17      12
Average TAT: 15.00
Average WT: 10.00
PS C:\Users\trish\OneDrive\Desktop\OS_LAB>
```

Program-3

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.

```
#include <stdio.h>

#define MAX_PROCESSES 10
#define TIME_QUANTUM 2

typedef struct {
   int burst_time, arrival_time, queue_type, waiting_time, turnaround_time, response_time,
remaining_time;
} Process;

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

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

int main() {
    Process processes[MAX_PROCESSES], system_queue[MAX_PROCESSES],
user_queue[MAX_PROCESSES];
    int n, sys_count = 0, user_count = 0, time = 0;
    float avg_waiting = 0, avg_turnaround = 0, avg_response = 0, throughput;

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

    for (int i = 0; i < n; i++) {
        printf("Enter Burst Time, Arrival Time and Queue of P%d: ", i + 1);
        scanf("%d %d %d", &processes[i].burst_time, &processes[i].arrival_time,
&processes[i].queue_type);
        processes[i].remaining_time = processes[i].burst_time;

        if (processes[i].queue_type == 1) {
            system_queue[sys_count++] = processes[i];
        } else {
            user_queue[user_count++] = processes[i];
        }
    }

    // Sort user processes by arrival time for FCFS
    for (int i = 0; i < user_count - 1; i++) {
        for (int j = 0; j < user_count - i - 1; j++) {
            if (user_queue[j].arrival_time > user_queue[j + 1].arrival_time) {
                Process temp = user_queue[j];
```

```
            user_queue[j] = user_queue[j + 1];
            user_queue[j + 1] = temp;
        }
    }
}

printf("\nQueue 1 is System Process\nQueue 2 is User Process\n");
round_robin(system_queue, sys_count, TIME_QUANTUM, &time);
fcfs(user_queue, user_count, &time);

printf("\nProcess  Waiting Time  Turn Around Time  Response Time\n");

for (int i = 0; i < sys_count; i++) {
    avg_waiting += system_queue[i].waiting_time;
    avg_turnaround += system_queue[i].turnaround_time;
    avg_response += system_queue[i].response_time;
    printf("%d        %d           %d              %d\n", i + 1, system_queue[i].waiting_time,
system_queue[i].turnaround_time, system_queue[i].response_time);
}

for (int i = 0; i < user_count; i++) {
    avg_waiting += user_queue[i].waiting_time;
    avg_turnaround += user_queue[i].turnaround_time;
    avg_response += user_queue[i].response_time;
    printf("%d        %d           %d              %d\n", i + 1 + sys_count,
user_queue[i].waiting_time, user_queue[i].turnaround_time, user_queue[i].response_time);
}

avg_waiting /= n;
avg_turnaround /= n;
avg_response /= n;
throughput = (float)n / time;

printf("\nAverage Waiting Time: %.2f", avg_waiting);
printf("\nAverage Turn Around Time: %.2f", avg_turnaround);
printf("\nAverage Response Time: %.2f", avg_response);
printf("\nThroughput: %.2f", throughput);
printf("\nProcess returned %d (0x%d) execution time: %.3f s\n", time, time, (float)time);

return 0;
```

}

OUTPUT:

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

Queue 1 is System Process
Queue 2 is User Process

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

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

Process returned 0 (0x0)   execution time : 21.307 s
Press any key to continue.
```

Program-4

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

a) RATE MONOTONIC:

```
#include <stdio.h>

#define MAX_PROCESSES 10

typedef struct {
    int id;
    int burst_time;
    int period;
    int remaining_time;
    int next_deadline;
} Process;

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

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

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

```
}

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

double utilization_factor(Process processes[], int n) {
    double sum = 0;
    for (int i = 0; i < n; i++) {
        sum += (double)processes[i].burst_time / processes[i].period;
    }
    return sum;
}

double rms_threshold(int n) {
    return n * (pow(2.0, 1.0 / n) - 1);
}

void rate_monotonic_scheduling(Process processes[], int n) {
    int lcm_period = calculate_lcm(processes, n);
    printf("LCM=%d\n\n", lcm_period);

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

    double utilization = utilization_factor(processes, n);
    double threshold = rms_threshold(n);
    printf("\n%.6f <= %.6f => %s\n", utilization, threshold, (utilization <= threshold) ?
"true" : "false");

    if (utilization > threshold) {
        printf("\nSystem may not be schedulable!\n");
```

```
        return;
    }

    int timeline = 0, executed = 0;
    while (timeline < lcm_period) {
        int selected = -1;
        for (int i = 0; i < n; i++) {
            if (timeline % processes[i].period == 0) {
                processes[i].remaining_time = processes[i].burst_time;
            }
            if (processes[i].remaining_time > 0) {
                selected = i;
                break;
            }
        }
        if (selected != -1) {
            printf("Time %d: Process %d is running\n", timeline, processes[selected].id);
            processes[selected].remaining_time--;
            executed++;
        } else {
            printf("Time %d: CPU is idle\n", timeline);
        }
        timeline++;
    }
}

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

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

    printf("Enter the CPU burst times:\n");
    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        scanf("%d", &processes[i].burst_time);
        processes[i].remaining_time = processes[i].burst_time;
    }
```

```
    printf("Enter the time periods:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &processes[i].period);
    }

    sort_by_period(processes, n);
    rate_monotonic_scheduling(processes, n);

    return 0;
}
```

OUTPUT:

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

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

4.100000 <= 0.779763 => false

System may not be schedulable!

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

b) EARLIEST DEADLINE:

```
#include <stdio.h>

int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

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

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

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

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

```
        if (earliest == -1) break;

        printf("%dms: Task %d is running.\n", time, p[earliest].id);
        p[earliest].burst_time--;
        time++;
    }
}

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

    struct Process processes[n];
    printf("Enter the CPU burst times:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &processes[i].burst_time);
        processes[i].id = i + 1;
    }

    printf("Enter the deadlines:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &processes[i].deadline);
    }

    printf("Enter the time periods:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &processes[i].period);
    }

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

    printf("\nSystem will execute for hyperperiod (LCM of periods): %d ms\n",
hyperperiod);

    earliest_deadline_first(processes, n, hyperperiod);
```

```
    return 0;
}
```

OUTPUT:

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

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

Scheduling occurs for 6 ms
0ms: Task 1 is running.
1ms: Task 1 is running.
2ms: Task 2 is running.
3ms: Task 2 is running.
4ms: Task 2 is running.
5ms: Task 3 is running.
```

c) PROPORTIONAL SCHEDULING:

```
#include <stdio.h>

struct Process {
   int id;
   int weight;
   int executed_time;
};

void proportionalScheduling(struct Process p[], int n, int total_time) {
   int i, total_weight = 0;

   for (i = 0; i < n; i++)
      total_weight += p[i].weight;

   printf("\n-- Proportional Scheduling Execution --\n");

   for (i = 0; i < n; i++) {
      int time_slice = (p[i].weight * total_time) / total_weight;
      p[i].executed_time = time_slice;
      printf("Process %d (Weight %d): Executed for %d units\n", p[i].id, p[i].weight,
p[i].executed_time);
   }
}

int main() {
   int i, n, total_time;

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

   struct Process p[n];

   for (i = 0; i < n; i++) {
      p[i].id = i + 1;
      printf("Enter weight (priority) for Process %d: ", p[i].id);
      scanf("%d", &p[i].weight);
   }
```

```
    printf("Enter total CPU time available: ");
    scanf("%d", &total_time);

    proportionalScheduling(p, n, total_time);

    return 0;
}
```

OUTPUT:

Program-5

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

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

int mutex = 1;
int full = 0;
int empty = 5;
int item = 0;

int wait(int s);
int signal(int s);
void producer();
void consumer();

int main() {
    int choice;

    printf("Producer-Consumer Problem Simulation\n");

    while (1) {
        printf("\n1. Produce\n2. Consume\n3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                if ((mutex == 1) && (empty != 0)) {
                    producer();
                } else {
                    printf("Buffer is full or mutex is locked. Cannot produce.\n");
                }
                break;
            case 2:
                if ((mutex == 1) && (full != 0)) {
                    consumer();
                } else {
                    printf("Buffer is empty or mutex is locked. Cannot consume.\n");
```

```
            }
            break;
        case 3:
            exit(0);
        default:
            printf("Invalid choice. Try again.\n");
        }
    }

    return 0;
}

int wait(int s) {
    return --s;
}

int signal(int s) {
    return ++s;
}

void producer() {
    mutex = wait(mutex);
    empty = wait(empty);
    full = signal(full);

    item++;
    printf("Produced item %d\n", item);

    mutex = signal(mutex);
}

void consumer() {
    mutex = wait(mutex);
    full = wait(full);
    empty = signal(empty);

    printf("Consumed item %d\n", item);
    item--;

    mutex = signal(mutex);
```

}

OUTPUT:

```
Producer-Consumer Problem Simulation

1. Produce
2. Consume
3. Exit
Enter your choice: 1
Produced item 1

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

1. Produce
2. Consume
3. Exit
Enter your choice: 2
Buffer is empty or mutex is locked. Cannot consume.

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

Program-6

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

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

#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

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

sem_t mutex;
sem_t S[N];

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

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

   sem_init(&mutex, 0, 1);

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

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

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

    return 0;
}

void test(int phnum) {
    if (state[phnum] == HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING) {

        state[phnum] = EATING;

        sleep(2);

        printf("Philosopher %d takes fork %d and %d\n", phnum + 1, LEFT + 1, phnum + 1);
        printf("Philosopher %d is Eating\n", phnum + 1);

        sem_post(&S[phnum]);
    }
}

void take_fork(int phnum) {
    sem_wait(&mutex);

    state[phnum] = HUNGRY;
    printf("Philosopher %d is Hungry\n", phnum + 1);

    test(phnum);

    sem_post(&mutex);

    sem_wait(&S[phnum]);
    sleep(1);
}

void put_fork(int phnum) {
    sem_wait(&mutex);
```

```
    state[phnum] = THINKING;

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

    test(LEFT);
    test(RIGHT);

    sem_post(&mutex);
}

void* philosopher(void* num) {
    int* i = (int*)num;

    while (1) {
        sleep(1);
        take_fork(*i);
        sleep(1);
        put_fork(*i);
    }
}
```

OUTPUT:

```
PS C:\Users\Admin\Desktop\1wa23cs023> gcc dinning_phi.c -o dinning_phi
PS C:\Users\Admin\Desktop\1wa23cs023> ./dinning_phi
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 5 is Hungry
Philosopher 4 is Hungry
Philosopher 2 is Hungry
Philosopher 3 is Hungry
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 1 is Hungry
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 3 is Hungry
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 1 is Hungry
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 4 is Hungry
Philosopher 5 putting fork 4 and 5 down
```

Program-7

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

```
#include <stdio.h>
#define MAX 10
#define RESOURCE_TYPES 3

void calculateNeed(int need[MAX][RESOURCE_TYPES], int
max[MAX][RESOURCE_TYPES], int allocation[MAX][RESOURCE_TYPES], int
numProcesses, int numResources) {
   for (int i = 0; i < numProcesses; i++) {
      for (int j = 0; j < numResources; j++) {
         need[i][j] = max[i][j] - allocation[i][j];
      }
   }
}

int isSafeState(int processes, int resources, int available[], int
max[][RESOURCE_TYPES], int allot[][RESOURCE_TYPES], int safeSeq[]) {
   int finish[processes], work[resources];
   int count = 0;
   for (int i = 0; i < processes; i++) {
      finish[i] = 0;
   }

   for (int i = 0; i < resources; i++) {
      work[i] = available[i];
   }
   while (count < processes) {
      int found = 0;
      for (int p = 0; p < processes; p++) {
         if (finish[p] == 0) {
            int canAllocate = 1;
            for (int r = 0; r < resources; r++) {
               if (max[p][r] - allot[p][r] > work[r]) {
                   canAllocate = 0;
                  break;
```

```
                }
            }
            if (canAllocate) {
                for (int r = 0; r < resources; r++) {
                    work[r] += allot[p][r];
                }
                safeSeq[count++] = p;
                finish[p] = 1;
                found = 1;
            }
        }
    }
    if (!found) {
        return 0;
    }
  }
  return 1;
}

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

  int max[numProcesses][numResources], allocation[numProcesses][numResources],
available[numResources];
  int need[numProcesses][numResources];
  int safeSeq[numProcesses];

  printf("\nEnter the maximum resources required for each process:\n");
  for (int i = 0; i < numProcesses; i++) {
    printf("Process P%d: ", i);
    for (int j = 0; j < numResources; j++) {
      scanf("%d", &max[i][j]);
    }
  }
  printf("\nEnter the resources currently allocated to each process:\n");
  for (int i = 0; i < numProcesses; i++) {
```

```
        printf("Process P%d: ", i);
        for (int j = 0; j < numResources; j++) {
            scanf("%d", &allocation[i][j]);
        }
    }
    printf("\nEnter the available resources:\n");
    for (int i = 0; i < numResources; i++) {
        scanf("%d", &available[i]);
    }

    calculateNeed(need, max, allocation, numProcesses, numResources);

    if (isSafeState(numProcesses, numResources, available, max, allocation, safeSeq)) {
        printf("\nThe system is in a safe state.\n");
        printf("Safe Sequence: ");
        for (int i = 0; i < numProcesses; i++) {
            printf("P%d ", safeSeq[i]);
        }
        printf("\n");
    } else {
        printf("\nThe system is not in a safe state.\n");
    }

    return 0;
}
```

OUTPUT:

```
Enter the number of processes: 5
Enter the number of resources: 3

Enter the maximum resources required for each process:
Process P0: 7 5 3
Process P1: 3 2 2
Process P2: 9 0 2
Process P3: 2 2 2
Process P4: 4 3 3

Enter the resources currently allocated to each process:
Process P0: 0 1 0
Process P1: 2 0 0
Process P2: 3 0 2
Process P3: 2 1 1
Process P4: 0 0 2

Enter the available resources:
3 3 2

The system is in a safe state.
Safe Sequence: P1 P3 P4 P0 P2

Process returned 0 (0x0)   execution time : 89.359 s
```

Program-8

8. Write a C program to simulate deadlock detection.

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

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

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

    int alloc[n][m], request[n][m], avail[m];
    bool finish[n];

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

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

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

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

```
    bool changed;
    do {
        changed = false;
        for (i = 0; i < n; i++) {
            if (!finish[i]) {
                bool can_finish = true;
                for (j = 0; j < m; j++) {
                    if (request[i][j] > avail[j]) {
                        can_finish = false;
                        break;
                    }
                }

                if (can_finish) {
                    for (k = 0; k < m; k++)
                        avail[k] += alloc[i][k];
                    finish[i] = true;
                    changed = true;
                    printf("Process %d can finish.\n", i);
                }
            }
        }
    } while (changed);

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

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

    return 0;
}
```

OUTPUT:

```
PS C:\Users\Admin\Desktop\1wa23cs023> gcc deadlock2.c -o deadlock2
PS C:\Users\Admin\Desktop\1wa23cs023> ./deadlock2
Enter number of processes and resources:
5 3
Enter allocation matrix:
0 1 0
2 0 0
3 0 3
2 1 1
0 0 2
Enter request matrix:
0 0 0
2 0 2
0 0 1
1 0 0
0 0 2
Enter available matrix:
0 0 0
Process 0 can finish.
System is in a deadlock state.
```

Program-9

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

a) Best-fit
b) Worst-fit
c) First-fit

```
#include <stdio.h>

struct Block {
    int block_no;
    int block_size;
    int is_free;
};

struct File {
    int file_no;
    int file_size;
};

void bestFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
    printf("\nMemory Management Scheme - Best Fit\n");
    printf("File_no\tFile_size\tBlock_no\tBlock_size\tFragment\n");

    for (int i = 0; i < n_files; i++) {
        int best_fit_block = -1;
        int min_fragment = 100000;

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

```
            if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
                int fragment = blocks[j].block_size - files[i].file_size;
                if (fragment < min_fragment) {
                    min_fragment = fragment;
                    best_fit_block = j;
                }
            }
        }

        if (best_fit_block != -1) {
            blocks[best_fit_block].is_free = 0;
            printf("%d\t%d\t\t%d\t\t%d\t\t%d\n",
                   files[i].file_no, files[i].file_size,
                   blocks[best_fit_block].block_no,
                   blocks[best_fit_block].block_size, min_fragment);
        } else {
            printf("%d\t%d\t\tNot Allocated\n", files[i].file_no, files[i].file_size);
        }
    }
}

void worstFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
    printf("\nMemory Management Scheme - Worst Fit\n");
    printf("File_no\tFile_size\tBlock_no\tBlock_size\tFragment\n");

    for (int i = 0; i < n_files; i++) {
        int worst_fit_block = -1;
```

```
        int max_fragment = -1;

        for (int j = 0; j < n_blocks; j++) {
            if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
                int fragment = blocks[j].block_size - files[i].file_size;
                if (fragment > max_fragment) {
                    max_fragment = fragment;
                    worst_fit_block = j;
                }
            }
        }

        if (worst_fit_block != -1) {
            blocks[worst_fit_block].is_free = 0;
            printf("%d\t%d\t\t%d\t\t%d\t\t%d\n",
                   files[i].file_no, files[i].file_size,
                   blocks[worst_fit_block].block_no,
                   blocks[worst_fit_block].block_size, max_fragment);
        } else {
            printf("%d\t%d\t\tNot Allocated\n", files[i].file_no, files[i].file_size);
        }
    }
}

void firstFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
    printf("\nMemory Management Scheme - First Fit\n");
    printf("File_no\tFile_size\tBlock_no\tBlock_size\tFragment\n");
```

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

        for (int j = 0; j < n_blocks; j++) {
            if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
                int fragment = blocks[j].block_size - files[i].file_size;
                blocks[j].is_free = 0;
                printf("%d\t%d\t\t%d\t\t%d\t\t%d\n",
                       files[i].file_no, files[i].file_size,
                       blocks[j].block_no, blocks[j].block_size, fragment);
                allocated = 1;
                break;
            }
        }

        if (!allocated) {
            printf("%d\t%d\t\tNot Allocated\n", files[i].file_no, files[i].file_size);
        }
    }
}

void resetBlocks(struct Block blocks[], int n_blocks) {
    for (int i = 0; i < n_blocks; i++) {
        blocks[i].is_free = 1;
    }
}
```

```
int main() {
    int n_blocks, n_files, choice;

    printf("Enter the number of blocks: ");
    scanf("%d", &n_blocks);
    struct Block blocks[n_blocks];

    for (int i = 0; i < n_blocks; i++) {
        blocks[i].block_no = i + 1;
        printf("Enter the size of block %d: ", i + 1);
        scanf("%d", &blocks[i].block_size);
        blocks[i].is_free = 1;
    }

    printf("Enter the number of files: ");
    scanf("%d", &n_files);
    struct File files[n_files];

    for (int i = 0; i < n_files; i++) {
        files[i].file_no = i + 1;
        printf("Enter the size of file %d: ", i + 1);
        scanf("%d", &files[i].file_size);
    }

    printf("\nChoose Memory Allocation Technique:\n");
    printf("1. First Fit\n");
```

```
    printf("2. Best Fit\n");
    printf("3. Worst Fit\n");
    printf("Enter choice (1/2/3): ");
    scanf("%d", &choice);

    resetBlocks(blocks, n_blocks);

    switch (choice) {
        case 1:
            firstFit(blocks, n_blocks, files, n_files);
            break;
        case 2:
            bestFit(blocks, n_blocks, files, n_files);
            break;
        case 3:
            worstFit(blocks, n_blocks, files, n_files);
            break;
        default:
            printf("Invalid choice\n");
    }

    return 0;
}
```

OUTPUT:

```
Enter the size of the blocks:
Block 1: 100
Block 2: 500
Block 3: 200
Block 4: 300
Block 5: 600
Enter the size of the files:
File 1: 212
File 2: 417
File 3: 112
File 4: 426

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

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

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

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

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

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

Program-10

10. Write a C program to simulate page replacement algorithms
a) FIFO
d) LRU
e) Optimal

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

int findOptimal(int pages[], int n, int frame[], int frameSize, int index) {
   int farthest = index, pos = -1;
   for (int i = 0; i < frameSize; i++) {
      int j;
      for (j = index; j < n; j++) {
         if (frame[i] == pages[j]) {
            if (j > farthest) {
               farthest = j;
               pos = i;
            }
            break;
         }
      }
      if (j == n) return i;
   }
   return (pos == -1) ? 0 : pos;
}

void fifo(int pages[], int n, int frameSize) {
   int frame[frameSize];
   int front = 0, pageFaults = 0;
   int count = 0;

   printf("\nFIFO Page Replacement:\n");

   for (int i = 0; i < frameSize; i++) frame[i] = -1;

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

```
            if (frame[j] == pages[i]) {
                found = 1;
                break;
            }
        }

        if (!found) {
            frame[front] = pages[i];
            front = (front + 1) % frameSize;
            pageFaults++;
        }

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

    printf("Total Page Faults (FIFO): %d\n", pageFaults);
}

void lru(int pages[], int n, int frameSize) {
    int frame[frameSize], count[frameSize];
    int time = 0, pageFaults = 0;

    printf("\nLRU Page Replacement:\n");

    for (int i = 0; i < frameSize; i++) {
        frame[i] = -1;
        count[i] = 0;
    }

    for (int i = 0; i < n; i++) {
        int found = 0, pos = 0, min = time;

        for (int j = 0; j < frameSize; j++) {
            if (frame[j] == pages[i]) {
                found = 1;
                count[j] = time++;
                break;
```

```
            }
        }

        if (!found) {
            for (int j = 0; j < frameSize; j++) {
                if (frame[j] == -1) {
                    pos = j;
                    break;
                }
                if (count[j] < min) {
                    min = count[j];
                    pos = j;
                }
            }
            frame[pos] = pages[i];
            count[pos] = time++;
            pageFaults++;
        }

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

    printf("Total Page Faults (LRU): %d\n", pageFaults);
}

void optimal(int pages[], int n, int frameSize) {
    int frame[frameSize];
    int pageFaults = 0;

    printf("\nOptimal Page Replacement:\n");

    for (int i = 0; i < frameSize; i++) frame[i] = -1;

    for (int i = 0; i < n; i++) {
        int found = 0;
        for (int j = 0; j < frameSize; j++) {
            if (frame[j] == pages[i]) {
```

```
                found = 1;
                break;
            }
        }

        if (!found) {
            int pos = -1;
            for (int j = 0; j < frameSize; j++) {
                if (frame[j] == -1) {
                    pos = j;
                    break;
                }
            }

            if (pos == -1) {
                pos = findOptimal(pages, n, frame, frameSize, i + 1);
            }

            frame[pos] = pages[i];
            pageFaults++;
        }

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

    printf("Total Page Faults (Optimal): %d\n", pageFaults);
}

int main() {
    int n, frameSize, choice;
    printf("Enter number of pages: ");
    scanf("%d", &n);

    int pages[n];
    printf("Enter page reference string: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &pages[i]);
```

```
    }

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

    do {
        printf("\n--- Page Replacement Algorithms ---\n");
        printf("1. FIFO\n2. LRU\n3. Optimal\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                fifo(pages, n, frameSize);
                break;
            case 2:
                lru(pages, n, frameSize);
                break;
            case 3:
                optimal(pages, n, frameSize);
                break;
            case 4:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice!\n");
        }

    } while (choice != 4);

    return 0;
}
```

OUTPUT:

```
Enter number of pages: 7
Enter page reference string: 1 3 0 3 5 6 3
Enter number of frames: 3

--- Page Replacement Algorithms ---
1. FIFO
2. LRU
3. Optimal
4. Exit
Enter your choice: 1

FIFO Page Replacement:
Frames: 1 - -
Frames: 1 3 -
Frames: 1 3 0
Frames: 1 3 0
Frames: 5 3 0
Frames: 5 6 0
Frames: 5 6 3
Total Page Faults (FIFO): 6

--- Page Replacement Algorithms ---
1. FIFO
2. LRU
3. Optimal
4. Exit
Enter your choice: 2

LRU Page Replacement:
Frames: 1 - -
Frames: 1 3 -
Frames: 1 3 0
Frames: 1 3 0
Frames: 5 3 0
Frames: 5 3 6
Frames: 5 3 6
Total Page Faults (LRU): 5
```

```
--- Page Replacement Algorithms ---
1. FIFO
2. LRU
3. Optimal
4. Exit
Enter your choice: 3

Optimal Page Replacement:
Frames: 1 - -
Frames: 1 3 -
Frames: 1 3 0
Frames: 1 3 0
Frames: 5 3 0
Frames: 6 3 0
Frames: 6 3 0
Total Page Faults (Optimal): 5

--- Page Replacement Algorithms ---
1. FIFO
2. LRU
3. Optimal
4. Exit
Enter your choice: 4
Exiting...
```