VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS

(23CS4PCOPS)

Submitted by

KATHASAGARAM AISHWARYA (1BM22CS123)

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 **KATHASGARAM AISHWARYA (1BM22CS123),** 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.

Radhika A D

Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak

Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

Course Outcome

CO1	Apply the different concepts and functionalities of Operating System
CO2	Analyze 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:

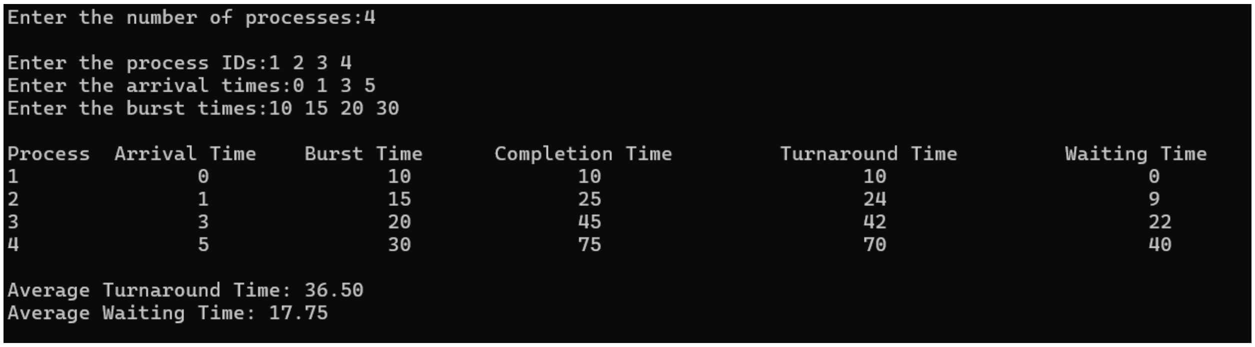
Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time.
→ FCFS
→ SJF (pre-emptive & Non-preemptive)

Code:

FCFS

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

int main(){
    int n;
    int process_id[n],at[n],bt[n],ct[n],tat[n],wt[n];
    printf("\nEnter number of processes: ");
    scanf("%d",&n);
    for(int i =0; i<n;i++){
        process_id[i] = i+1;
        printf("\nArrival Time for %d: ",(i+1));
        scanf("%d",&at[i]);
        printf("\nBurst Time for %d: ",(i+1));
        scanf("%d",&bt[i]);
    }
    int temp = 0;
    int temp2 = 0; int temp3 = 0;
    for(int i = 0; i< n; i++){
        for(int j = i +1; j<n;j++){
            if(at[i] > at[j]){
                temp = at[i];
                at[i] = at[j];
                at[j] = temp;
                temp2 = bt[i];
                bt[i] = bt[j];
                bt[j] = temp2;
                temp3 = process_id[i];
                process_id[i] = process_id[j];
                process_id[j] = temp3;

            }
        }
    }

    int timePassed = 0;
    for(int i = 0; i<n;i++){
        if(at[i] > timePassed){
            timePassed = timePassed + (at[i] - ct[i-1]);
        }
        timePassed += bt[i];
        ct[i] = timePassed;
    }

    for(int i = 0; i<n;i++){
        tat[i] = ct[i] = at[i];
        wt[i] = tat[i] = bt[i];
    }
    printf("\nPID\tAT\tBT\tCT\tTAT\tWT");

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

Result:

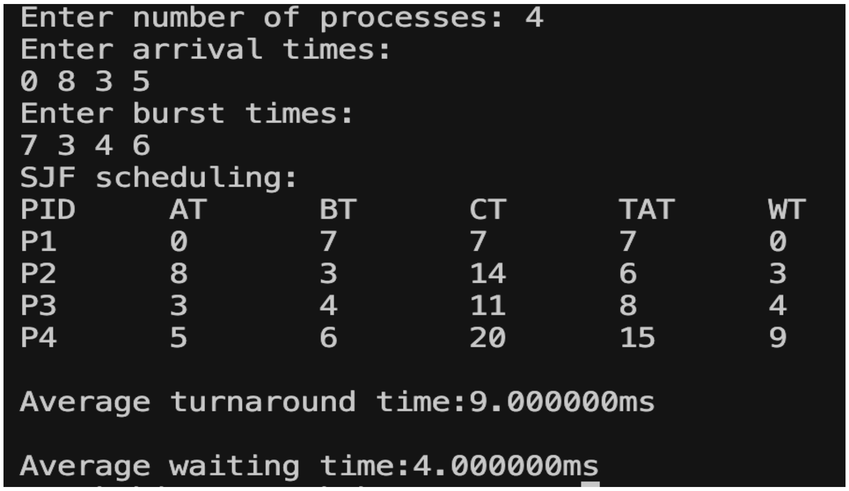


```
Enter the number of processes:4

Enter the process IDs:1 2 3 4
Enter the arrival times:0 1 3 5
Enter the burst times:10 15 20 30

Process  Arrival Time    Burst Time      Completion Time         Turnaround Time         Waiting Time
1               0               10              10                      10                      0
2               1               15              25                      24                      9
3               3               20              45                      42                      22
4               5               30              75                      70                      40

Average Turnaround Time: 36.50
Average Waiting Time: 17.75
```

SJF Non Pre-emptive

```
#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_burst_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].burst_time > processes[j + 1].burst_time) {
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
}
void compute_completion_time(Process *processes, int n) {
    int current_time = 0;
    int index = 0;
    while (index < n) {
        int next_process = -1;
        for (int i = 0; i < n; i++) {
            if (processes[i].arrival_time <= current_time && processes[i].completion_time == 0) {
                if (next_process == -1 || processes[i].burst_time < processes[next_process].burst_time)
{
                    next_process = i;
                }
            }
        }
        if (next_process == -1) {
              current_time = processes[index].arrival_time;
        } else {
            processes[next_process].completion_time = current_time +
processes[next_process].burst_time;
            current_time = processes[next_process].completion_time;
        }
        index++;
    }
}
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;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    Process *processes = (Process *)malloc(n * sizeof(Process));
    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);
        processes[i].completion_time = 0;
        processes[i].turnaround_time = 0;
        processes[i].waiting_time = 0;
    }
     sort_by_burst_time(processes, n);
     compute_completion_time(processes, n);
    compute_turnaround_waiting_time(processes, n);
    display_table(processes, n);
    free(processes);
    return 0;
}
```

Result:



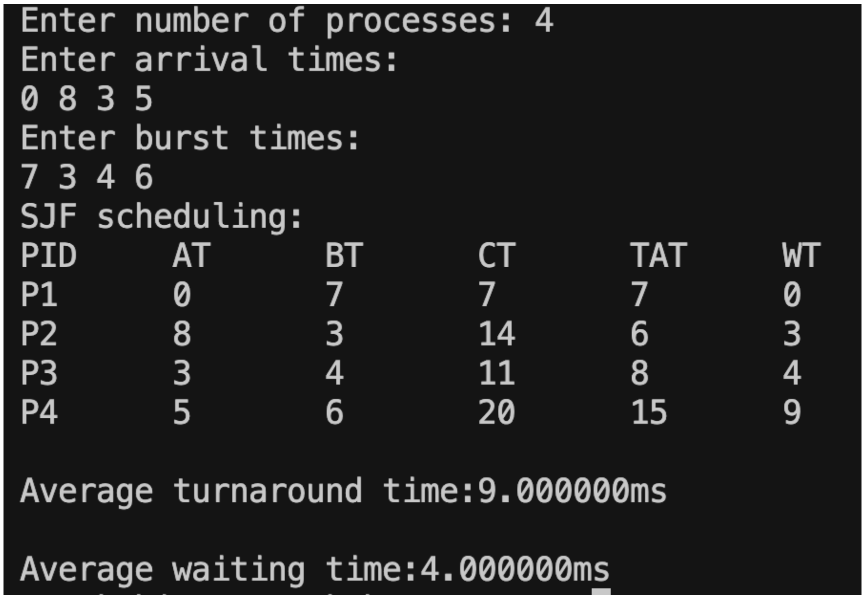
```
Enter number of processes: 4
Enter arrival times:
0 8 3 5
Enter burst times:
7 3 4 6
SJF scheduling:
PID     AT      BT      CT      TAT     WT
P1      0       7       7       7       0
P2      8       3       14      6       3
P3      3       4       11      8       4
P4      5       6       20      15      9

Average turnaround time:9.000000ms

Average waiting time:4.000000ms
```

SJF Pre-emptive

```
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
struct Process {
    int pid;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int completion_time;
    int turnaround_time;
    int waiting_time;
};

int findShortestJob(struct Process processes[], int n, int current_time) {
    int shortest_job_index = -1;
    int shortest_job = INT_MAX;
    for (int i = 0; i < n; i++) {
        if (processes[i].arrival_time <= current_time && processes[i].remaining_time > 0 &&
processes[i].remaining_time < shortest_job) {
            shortest_job_index = i;
            shortest_job = processes[i].remaining_time;
        }
    }
    return shortest_job_index;
}

void SJF(struct Process processes[], int n) {
    int current_time = 0;
    int completed = 0;
    while (completed < n) {
        int shortest_job_index = findShortestJob(processes, n, current_time);
        if (shortest_job_index == -1) {
            current_time++;
        } else {

            processes[shortest_job_index].remaining_time--;
            current_time++;
            if (processes[shortest_job_index].remaining_time == 0) {

                processes[shortest_job_index].completion_time = current_time;
                processes[shortest_job_index].turnaround_time =
processes[shortest_job_index].completion_time - processes[shortest_job_index].arrival_time;
                processes[shortest_job_index].waiting_time =
processes[shortest_job_index].turnaround_time - processes[shortest_job_index].burst_time;
                completed++;
            }
        }
    }
}

int main() {
    int n;
    printf("Enter the total number of processes: ");
    scanf("%d", &n);
    struct Process processes[n];
    printf("Enter Arrival Time and Burst Time for each process:\n");
```

```
  for (int i = 0; i < n; i++) {
    printf("Process %d:\n", i + 1);
    printf("Arrival Time: ");
    scanf("%d", &processes[i].arrival_time);
    printf("Burst Time: ");
    scanf("%d", &processes[i].burst_time);
    processes[i].remaining_time = processes[i].burst_time;
    processes[i].pid = i + 1;
  }
  SJF(processes, n);
  printf("\nProcess\tArrival Time\tBurst Time\tCompletion Time\tWaiting Time\tTurnaround 
Time\n");
  for (int i = 0; i < n; i++) {
    printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].arrival_time, 
processes[i].burst_time, processes[i].completion_time, processes[i].waiting_time, 
processes[i].turnaround_time);
  }
  return 0;
}
```

Result:



```
Enter number of processes: 4
Enter arrival times:
0 8 3 5
Enter burst times:
7 3 4 6
SJF scheduling:
PID     AT      BT      CT      TAT     WT
P1      0       7       7       7       0
P2      8       3       14      6       3
P3      3       4       11      8       4
P4      5       6       20      15      9

Average turnaround time:9.000000ms

Average waiting time:4.000000ms
```

Program - 2

Question:

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

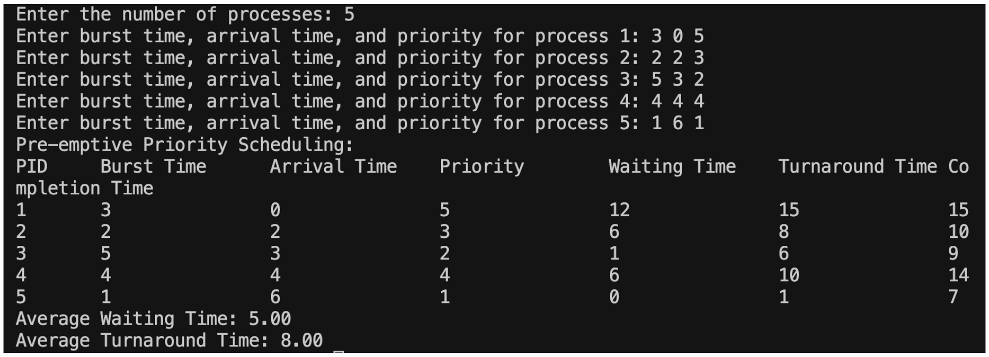
Code:

Priority Pre-emptive Scheduling

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

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

void calculateTimes(Process processes[], int n) {
   int completed = 0, current_time = 0, min_priority_index;
   int total_waiting_time = 0, total_turnaround_time = 0;

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

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

```
        }

        if (min_priority_index != -1) {
            processes[min_priority_index].remaining_time--;

            if (processes[min_priority_index].remaining_time == 0) {
                completed++;
                int finish_time = current_time + 1;
                processes[min_priority_index].completion_time = finish_time; // Set completion time
                processes[min_priority_index].turnaround_time = finish_time -
processes[min_priority_index].arrival_time;
                processes[min_priority_index].waiting_time =
processes[min_priority_index].turnaround_time - processes[min_priority_index].burst_time;
                total_waiting_time += processes[min_priority_index].waiting_time;
                total_turnaround_time += processes[min_priority_index].turnaround_time;
            }
        }

        current_time++;
    }

    printf("Pre-emptive Priority Scheduling:\n");
    printf("PID\tBurst Time\tArrival Time\tPriority\tWaiting Time\tTurnaround Time\tCompletion
Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].burst_time,
processes[i].arrival_time, processes[i].priority, processes[i].waiting_time,
processes[i].turnaround_time, processes[i].completion_time);
    }
    printf("Average Waiting Time: %.2f\n", (float) total_waiting_time / n);
    printf("Average Turnaround Time: %.2f\n", (float) total_turnaround_time / n);
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    Process processes[n];
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        printf("Enter burst time, arrival time, and priority for process %d: ", i + 1);
        scanf("%d %d %d", &processes[i].burst_time, &processes[i].arrival_time,
&processes[i].priority);
        processes[i].remaining_time = processes[i].burst_time;
    }
    calculateTimes(processes, n);
    return
```

Result:

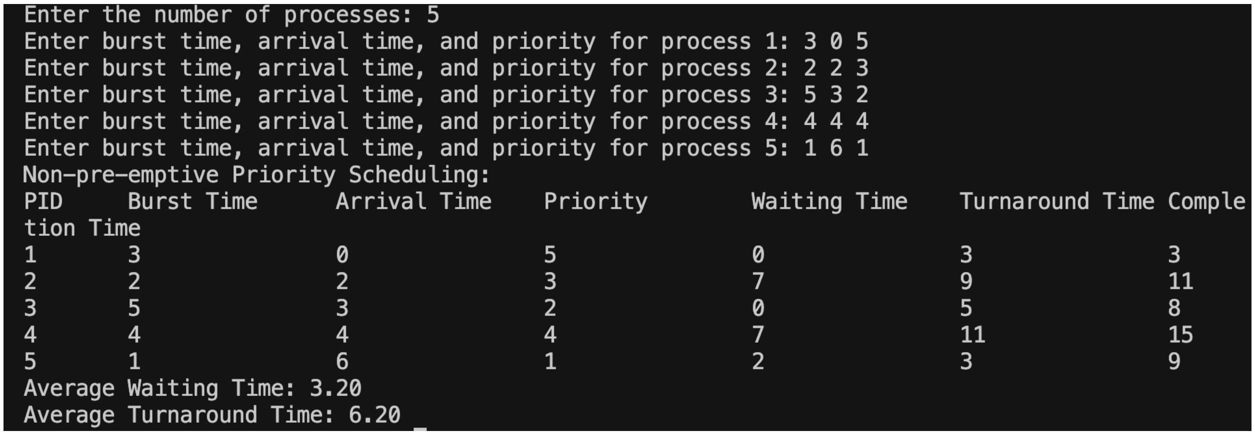


```
Enter the number of processes: 5
Enter burst time, arrival time, and priority for process 1: 3 0 5
Enter burst time, arrival time, and priority for process 2: 2 2 3
Enter burst time, arrival time, and priority for process 3: 5 3 2
Enter burst time, arrival time, and priority for process 4: 4 4 4
Enter burst time, arrival time, and priority for process 5: 1 6 1
Pre-emptive Priority Scheduling:
PID     Burst Time      Arrival Time    Priority        Waiting Time    Turnaround Time Co
mpletion Time
1       3               0               5               12              15              15
2       2               2               3               6               8               10
3       5               3               2               1               6               9
4       4               4               4               6               10              14
5       1               6               1               0               1               7
Average Waiting Time: 5.00
Average Turnaround Time: 8.00 _
```

Priority Non Pre-emptive Scheduling

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

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

void calculateTimes(Process processes[], int n) {
    int completed = 0, current_time = 0;
    int total_waiting_time = 0, total_turnaround_time = 0;

    while (completed != n) {
        int min_priority_index = -1;
        int min_priority = _INT_MAX_;
        for (int i = 0; i < n; i++) {
            if (processes[i].arrival_time <= current_time && !processes[i].completed &&
processes[i].priority < min_priority) {
                min_priority = processes[i].priority;
                min_priority_index = i;
            }
        }
        if (min_priority_index != -1) {
            current_time += processes[min_priority_index].burst_time;
            processes[min_priority_index].waiting_time = current_time -
processes[min_priority_index].arrival_time - processes[min_priority_index].burst_time;
            processes[min_priority_index].turnaround_time = current_time -
processes[min_priority_index].arrival_time;
            processes[min_priority_index].completion_time = current_time; // Set completion time
            processes[min_priority_index].completed = true;
            total_waiting_time += processes[min_priority_index].waiting_time;
            total_turnaround_time += processes[min_priority_index].turnaround_time;
            completed++;
        } else {
            current_time++;
        }
    }

    printf("Non-pre-emptive Priority Scheduling:\n");
    printf("PID\tBurst Time\tArrival Time\tPriority\tWaiting Time\tTurnaround Time\tCompletion
Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].burst_time,
processes[i].arrival_time, processes[i].priority, processes[i].waiting_time,
processes[i].turnaround_time, processes[i].completion_time);
    }
    printf("Average Waiting Time: %.2f\n", (float) total_waiting_time / n);
    printf("Average Turnaround Time: %.2f\n", (float) total_turnaround_time / n);
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    Process processes[n];
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        processes[i].completed = false;
        printf("Enter burst time, arrival time, and priority for process %d: ", i + 1);
        scanf("%d %d %d", &processes[i].burst_time, &processes[i].arrival_time,
&processes[i].priority);
    }
    calculateTimes(processes, n);
    return 0;
}
```

Result:



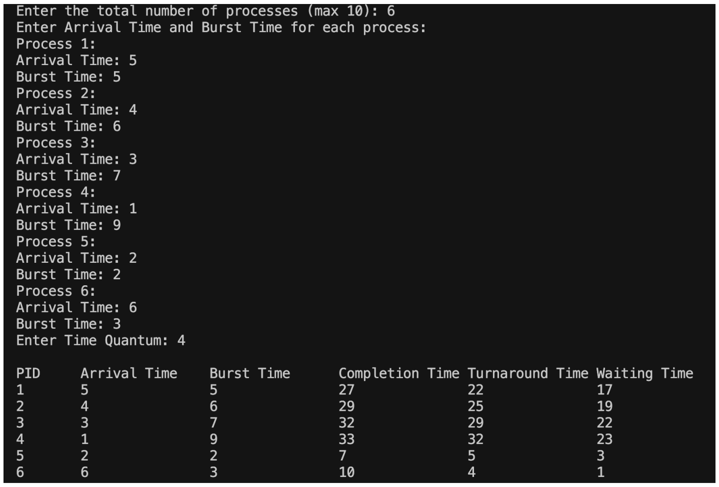
```
Enter the number of processes: 5
Enter burst time, arrival time, and priority for process 1: 3 0 5
Enter burst time, arrival time, and priority for process 2: 2 2 3
Enter burst time, arrival time, and priority for process 3: 5 3 2
Enter burst time, arrival time, and priority for process 4: 4 4 4
Enter burst time, arrival time, and priority for process 5: 1 6 1
Non-pre-emptive Priority Scheduling:
PID     Burst Time      Arrival Time    Priority        Waiting Time    Turnaround Time Comple
tion Time
1       3               0               5               0               3               3
2       2               2               3               7               9               11
3       5               3               2               0               5               8
4       4               4               4               7               11              15
5       1               6               1               2               3               9
Average Waiting Time: 3.20
Average Turnaround Time: 6.20
```

Round Robin

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_PROCESSES 10
struct Process {
    int pid;
    int burst_time;
    int arrival_time;
    int remaining_time;
    int turnaround_time;
    int waiting_time;
    int completion_time;
};
void round_robin(struct Process proc[], int n, int quantum) {
    int current_time = 0;
    int completed_processes = 0;
    while (completed_processes < n) {
        bool process_found = false;
        for (int i = 0; i < n; i++) {
            if (proc[i].remaining_time > 0 && proc[i].arrival_time <= current_time) {
                process_found = true;
                if (proc[i].remaining_time > quantum) {
                    current_time += quantum;
                    proc[i].remaining_time -= quantum;
                } else {
                    current_time += proc[i].remaining_time;
                    proc[i].completion_time = current_time;
                    proc[i].turnaround_time = proc[i].completion_time - proc[i].arrival_time;
                    proc[i].waiting_time = proc[i].turnaround_time - proc[i].burst_time;
                    proc[i].remaining_time = 0;
                    completed_processes++;
                }
            }
        }
        if (!process_found) {
            current_time++;
        }
    }
    // Print the results
    printf("\nPID\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting
Time\n");
    float total_completion_time = 0, total_turnaround_time = 0, total_waiting_time = 0;
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", proc[i].pid, proc[i].arrival_time,
               proc[i].burst_time, proc[i].completion_time, proc[i].turnaround_time,
proc[i].waiting_time);
        total_completion_time += proc[i].completion_time;
        total_turnaround_time += proc[i].turnaround_time;
        total_waiting_time += proc[i].waiting_time;
    }
    // Calculate and display averages
    printf("\nAverage Completion Time: %.2f\n", total_completion_time / n);
    printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);
    printf("Average Waiting Time: %.2f\n", total_waiting_time / n);
```

```
}
int main() {
    int n, quantum;
    printf("Enter the total number of processes (max %d): ", MAX_PROCESSES);
    scanf("%d", &n);
    if (n > MAX_PROCESSES) {
        printf("Number of processes exceeds maximum limit.\n");
        return 1;
    }
    struct Process proc[MAX_PROCESSES];
    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &proc[i].arrival_time);
        printf("Burst Time: ");
        scanf("%d", &proc[i].burst_time);
        proc[i].pid = i + 1;
        proc[i].remaining_time = proc[i].burst_time; // Initialize remaining time
        proc[i].turnaround_time = 0; // Initialize turnaround time
        proc[i].waiting_time = 0; // Initialize waiting time
        proc[i].completion_time = 0; // Initialize completion time
    }
    printf("Enter Time Quantum: ");
    scanf("%d", &quantum);
    round_robin(proc, n, quantum);
    return 0;
}
```

Result:



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

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

Program - 3

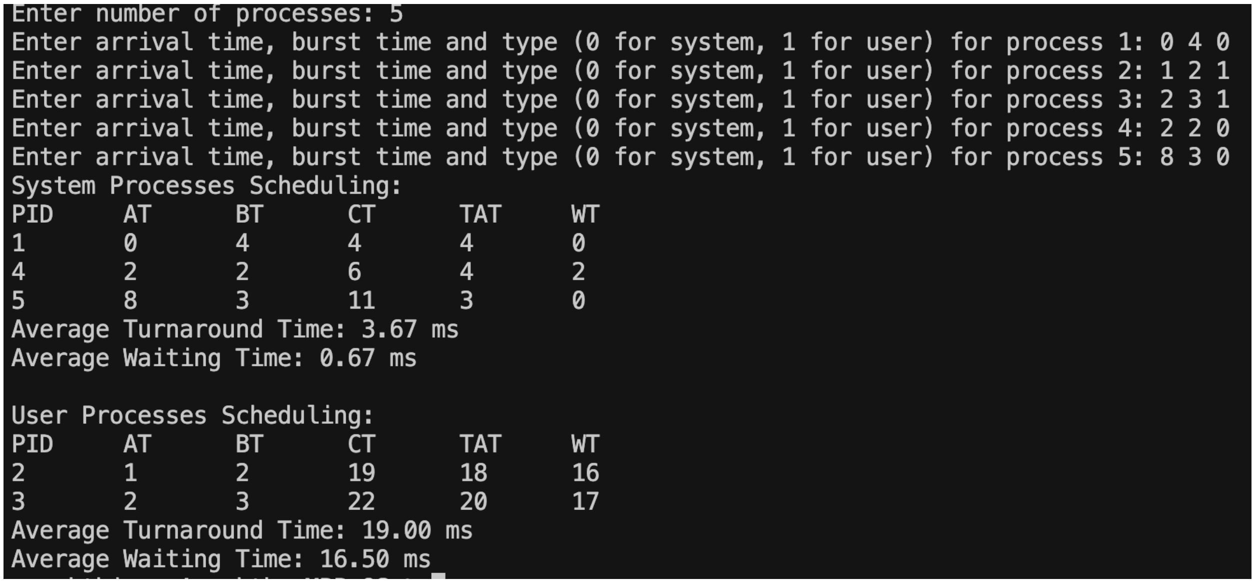
Question :

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

void simulateFCFS(int proc_id[], int at[], int bt[], int n, int start_time) {
    int c = start_time, ct[n], tat[n], wt[n];
    double ttat = 0.0, twt = 0.0;
    for(int i=0; i<n; i++) {
        if(c >= at[i])
            c += bt[i];
        else
            c = at[i] + bt[i];
        ct[i] = c;
    }

    for(int i=0; i<n; i++)
        tat[i] = ct[i] - at[i];

    for(int i=0; i<n; i++)
        wt[i] = tat[i] - bt[i];
    printf("PID\tAT\tBT\tCT\tTAT\tWT\n");
    for(int i=0; i<n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", proc_id[i], at[i], bt[i], ct[i], tat[i], wt[i]);
        ttat += tat[i];
        twt += wt[i];
    }
    printf("Average Turnaround Time: %.2lf ms\n", ttat/n);
    printf("Average Waiting Time: %.2lf ms\n", twt/n);
}

void main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    int proc_id[n], at[n], bt[n], type[n];
    int sys_proc_id[n], sys_at[n], sys_bt[n], user_proc_id[n], user_at[n], user_bt[n];
    int sys_count = 0, user_count = 0;
    for(int i=0; i<n; i++) {
        proc_id[i] = i + 1;
         printf("Enter arrival time, burst time and type (0 for system, 1 for user) for process %d: ",
i+1);
        scanf("%d %d %d", &at[i], &bt[i], &type[i]);
        if(type[i] == 0) {
            sys_proc_id[sys_count] = proc_id[i];
            sys_at[sys_count] = at[i];
            sys_bt[sys_count] = bt[i];
            sys_count++;
        } else {
            user_proc_id[user_count] = proc_id[i];
            user_at[user_count] = at[i];
            user_bt[user_count] = bt[i];
            user_count++;
        }
    }
    sort(sys_proc_id, sys_at, sys_bt, sys_count);
    sort(user_proc_id, user_at, user_bt, user_count);  //arrival time sort

    printf("System Processes Scheduling:\n");
    simulateFCFS(sys_proc_id, sys_at, sys_bt, sys_count, 0);

    int system_end_time = 0;
    if (sys_count > 0) {
        system_end_time = sys_at[sys_count - 1] + sys_bt[sys_count - 1];
        for (int i = 0; i < sys_count - 1; i++) {
            if (sys_at[i + 1] > system_end_time) {
                system_end_time = sys_at[i + 1];
            }
            system_end_time += sys_bt[i];
        }
    }
    printf("\nUser Processes Scheduling:\n");
    simulateFCFS(user_proc_id, user_at, user_bt, user_count, system_end_time);
}
```

Result:



```
Enter number of processes: 5
Enter arrival time, burst time and type (0 for system, 1 for user) for process 1: 0 4 0
Enter arrival time, burst time and type (0 for system, 1 for user) for process 2: 1 2 1
Enter arrival time, burst time and type (0 for system, 1 for user) for process 3: 2 3 1
Enter arrival time, burst time and type (0 for system, 1 for user) for process 4: 2 2 0
Enter arrival time, burst time and type (0 for system, 1 for user) for process 5: 8 3 0
System Processes Scheduling:
PID     AT      BT      CT      TAT     WT
1       0       4       4       4       0
4       2       2       6       4       2
5       8       3       11      3       0
Average Turnaround Time: 3.67 ms
Average Waiting Time: 0.67 ms

User Processes Scheduling:
PID     AT      BT      CT      TAT     WT
2       1       2       19      18      16
3       2       3       22      20      17
Average Turnaround Time: 19.00 ms
Average Waiting Time: 16.50 ms
```

Program - 4

Question:

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

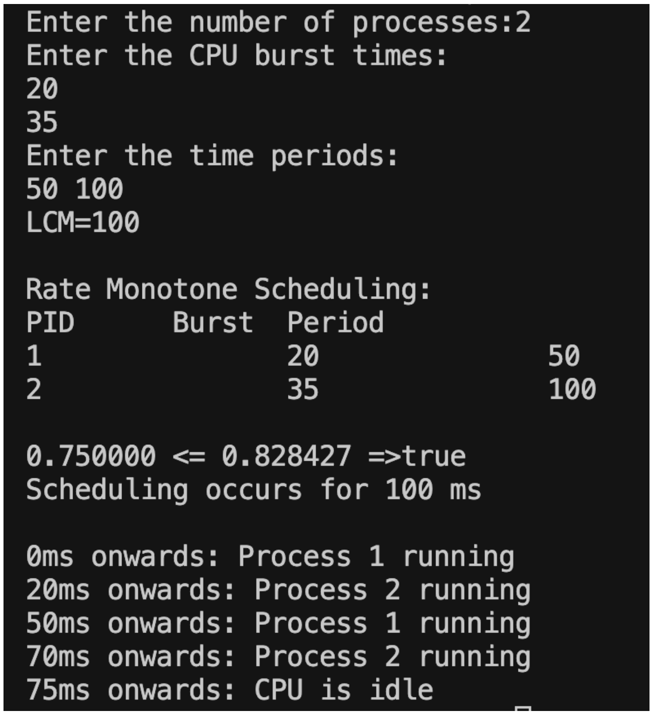
Code:

a) Rate-Monotonic

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

void sort (int proc[], int b[], int pt[], int n){
  int temp = 0;
  for (int i = 0; i < n; i++)
    {
      for (int j = i; j < n; j++)
        {
          if (pt[j] < pt[i])
            {
              temp = pt[i];
              pt[i] = pt[j];
              pt[j] = temp;
              temp = b[j];
              b[j] = b[i];
              b[i] = temp;
              temp = proc[i];
              proc[i] = proc[j];
              proc[j] = temp;
            }
        }
    }
}

int gcd (int a, int b){
  int r;
  while (b > 0)
    {
      r = a % b;
      a = b;
      b = r;
    }
  return a;
}

int lcmul (int p[], int n){
  int lcm = p[0];
  for (int i = 1; i < n; i++){
      lcm = (lcm * p[i]) / gcd (lcm, p[i]);
    }
  return lcm;
}

int main(){
  int n;
  printf ("Enter the number of processes:");
  scanf ("%d", &n);
  int proc[n], b[n], pt[n], rem[n];
  printf ("Enter the CPU burst times:\n");

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

for (int i = 0; i < n; i++)
    scanf ("%d", &pt[i]);

for (int i = 0; i < n; i++)
    proc[i] = i + 1;

sort (proc, b, pt, n);
int l = lcmul (pt, n);
printf ("LCM=%d\n", l);
printf ("\nRate Monotone Scheduling:\n");
printf ("PID\t Burst\tPeriod\n");
for (int i = 0; i < n; i++)
```

```
  printf ("%d\t\t%d\t\t%d\n", proc[i], b[i], pt[i]);
double sum = 0.0;
for (int i = 0; i < n; i++){
      sum += (double) b[i] / pt[i];
    }

double rhs = n * (pow (2.0, (1.0 / n)) - 1.0);
printf ("\n%lf <= %lf =>%s\n", sum, rhs, (sum <= rhs) ? "true" : "false");

if (sum > rhs)
    exit (0);
printf ("Scheduling occurs for %d ms\n\n", l);
int time = 0, prev = 0, x = 0;

while (time < l){
      int f = 0;

      for (int i = 0; i < n; i++)
        {
          if (time % pt[i] == 0)
            rem[i] = b[i];
          if (rem[i] > 0)
            {
              if (prev != proc[i])
                {
                  printf ("%dms onwards: Process %d running\n", time,
                          proc[i]);
                  prev = proc[i];
                }
              rem[i]--;
              f = 1;
              break;
              x = 0;
            }
        }
      if (!f)
        {
          if (x != 1)
            {
              printf ("%dms onwards: CPU is idle\n", time);
              x = 1;
            }
          }
        time++;
      }
    }
```

Result:



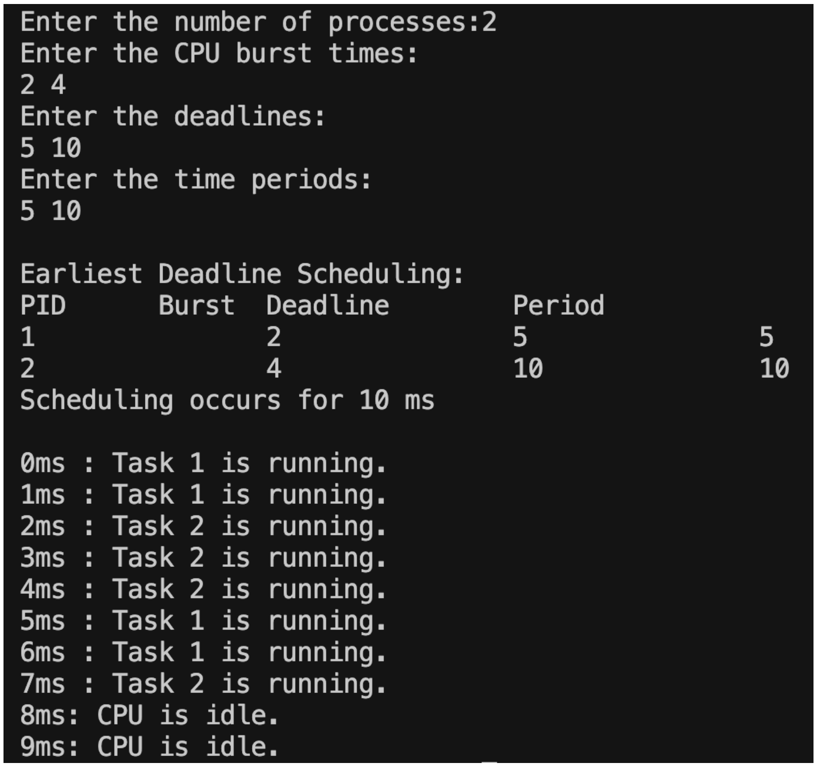
```
Enter the number of processes:2
Enter the CPU burst times:
20
35
Enter the time periods:
50 100
LCM=100

Rate Monotone Scheduling:
PID      Burst  Period
1              20              50
2              35              100

0.750000 <= 0.828427 =>true
Scheduling occurs for 100 ms

0ms onwards: Process 1 running
20ms onwards: Process 2 running
50ms onwards: Process 1 running
70ms onwards: Process 2 running
75ms onwards: CPU is idle
```

b) Earliest Deadline First

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

void sort (int proc[], int d[], int b[], int pt[], int n){
  int temp = 0;
  for (int i = 0; i < n; i++){
      for (int j = i; j < n; j++){
          if (d[j] < d[i]){
              temp = d[j];
              d[j] = d[i];
              d[i] = temp;
              temp = pt[i];
              pt[i] = pt[j];
              pt[j] = temp;
              temp = b[j];
              b[j] = b[i];
              b[i] = temp;
              temp = proc[i];
              proc[i] = proc[j];
              proc[j] = temp;
             }
         }
     }
 }

 int gcd (int a, int b){
   int r;
   while (b > 0)
     {
       r = a % b;
       a = b;
       b = r;
     }
   return a;
 }
 int lcmul (int p[], int n){
   int lcm = p[0];
   for (int i = 1; i < n; i++)
     {
       lcm = (lcm * p[i]) / gcd (lcm, p[i]);
     }
   return lcm;
 }

 int main (){
   int n;
   printf ("Enter the number of processes:");
   scanf ("%d", &n);
   int proc[n], b[n], pt[n], d[n], rem[n];
   printf ("Enter the CPU burst times:\n");
```

```
for (int i = 0; i < n; i++)
  {
    scanf ("%d", &b[i]);
    rem[i] = b[i];
  }
printf ("Enter the deadlines:\n");
for (int i = 0; i < n; i++)
  scanf ("%d", &d[i]);
printf ("Enter the time periods:\n");
for (int i = 0; i < n; i++)
  scanf ("%d", &pt[i]);
for (int i = 0; i < n; i++)
  proc[i] = i + 1;
sort (proc, d, b, pt, n);
int l = lcmul (pt, n);
printf ("\nEarliest Deadline Scheduling:\n");
printf ("PID\t Burst\tDeadline\tPeriod\n");
for (int i = 0; i < n; i++)
  printf ("%d\t\t%d\t\t%d\t\t%d\n", proc[i], b[i], d[i], pt[i]);
printf ("Scheduling occurs for %d ms\n\n", l);
int time = 0, prev = 0, x = 0;
int nextDeadlines[n];
for (int i = 0; i < n; i++)
  {
    nextDeadlines[i] = d[i];
    rem[i] = b[i];
  }
while (time < l)
  {
    for (int i = 0; i < n; i++)
      {
        if (time % pt[i] == 0 && time != 0)
          {
            nextDeadlines[i] = time + d[i];
            rem[i] = b[i];
          }
      }
    int minDeadline = l + 1;
    int taskToExecute = -1;
    for (int i = 0; i < n; i++){
      if (rem[i] > 0 && nextDeadlines[i] < minDeadline){
        minDeadline = nextDeadlines[i];
        taskToExecute = i;
       }
     }
    if (taskToExecute != -1){
      printf ("%dms : Task %d is running.\n", time, proc[taskToExecute]);
      rem[taskToExecute]--;
     }
    else{
      printf ("%dms: CPU is idle.\n", time);
     }
    time++;
  }
}
```

Result:

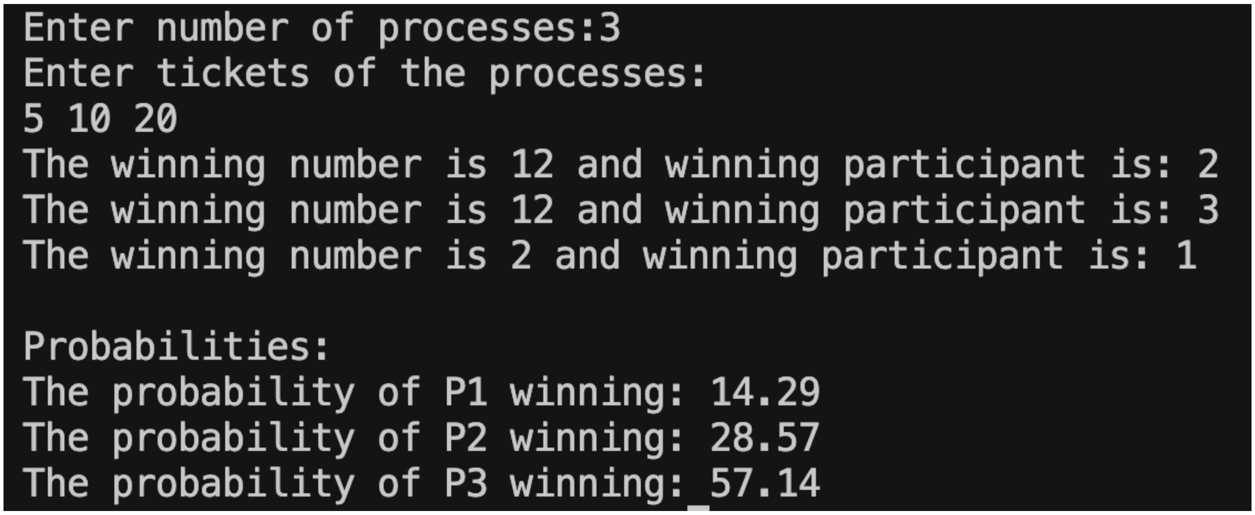


```
Enter the number of processes:2
Enter the CPU burst times:
2 4
Enter the deadlines:
5 10
Enter the time periods:
5 10

Earliest Deadline Scheduling:
PID      Burst  Deadline        Period
1               2               5               5
2               4               10              10
Scheduling occurs for 10 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 1 is running.
6ms : Task 1 is running.
7ms : Task 2 is running.
8ms: CPU is idle.
9ms: CPU is idle.
```

c) Proportional Scheduling

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main() {
   srand(time(NULL));
   int n;
   printf("Enter number of processes:");
   scanf("%d",&n);
   int p[n],t[n],cum[n],m[n];int c=0;int total = 0,count=0;
   printf("Enter tickets of the processes:\n");
   for(int i=0;i<n;i++){
      scanf("%d",&t[i]);
      c+=t[i];
      cum[i]=c;
      p[i]=i+1;
      m[i]=0;
      total+= t[i];
   }
   while(count<n){
      int wt=rand()%total;
      for (int i=0;i<n;i++)
      {
         if (wt<cum[i] && m[i]==0)
         {
            printf("The winning number is %d and winning participant is: %d\n",wt,p[i]);
            m[i]=1;count++;
         }
      }
   }
   printf("\nProbabilities:\n");
   for (int i = 0; i < n; i++)
   {
      printf("The probability of P%d winning: %.2f\n",p[i],((double)t[i]/total*100));
   }
}
```

Result:



```
Enter number of processes:3
Enter tickets of the processes:
5 10 20
The winning number is 12 and winning participant is: 2
The winning number is 12 and winning participant is: 3
The winning number is 2 and winning participant is: 1

Probabilities:
The probability of P1 winning: 14.29
The probability of P2 winning: 28.57
The probability of P3 winning: 57.14
```

Program - 5

Question:

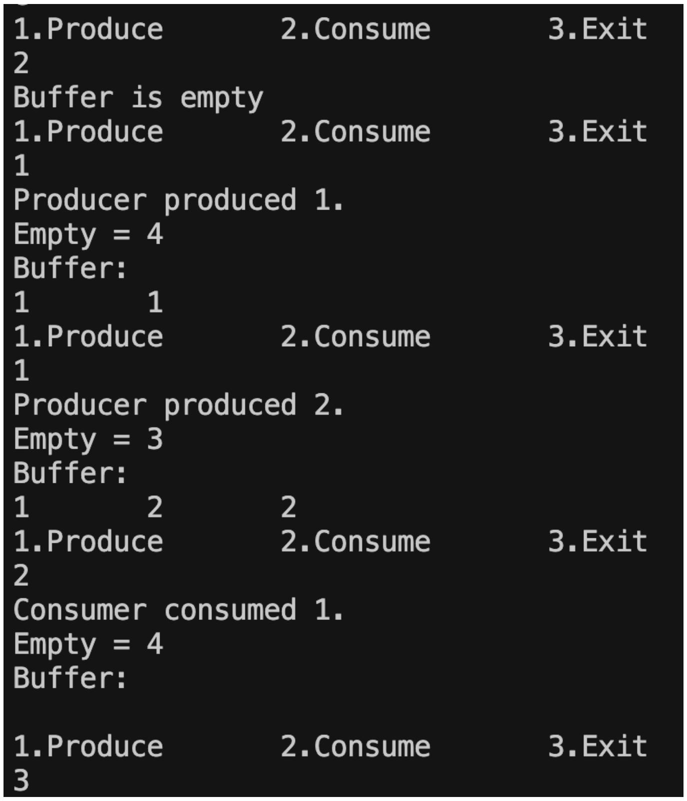
Write a C program to simulate producer-consumer problem using semaphores.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5

int buffer[MAX];
int empty = MAX;
int full = 0;
int mutex = 1;
int x = 0;
void custom_wait(int* s) {
   while (*s <= 0);
   --(*s);
}
void custom_signal(int* s) {
   ++(*s);
}
void producer() {
   custom_wait(&mutex);
   custom_wait(&empty);
   x++;
   buffer[full] = x;
   custom_signal(&full);
   custom_signal(&mutex);
   printf("Producer produced %d.\n", x);
   printf("Empty = %d\n", empty);
   printf("Buffer:\n");
   for (int i = 0; i < full; i++) {
      printf("%d\t", buffer[i]);
   }
   printf("%d\n", buffer[full - 1]); /
}
void consumer() {
   custom_wait(&full);
```

```
    custom_wait(&mutex);
    printf("Consumer consumed %d.\n", buffer[full - 1]);
    full--;
    custom_signal(&empty);
    custom_signal(&mutex);
    printf("Empty = %d\n", empty);
    printf("Buffer:\n");
    for (int i = 0; i < full; i++) {
        printf("%d\t", buffer[i]);
    }
    printf("\n");
}
int main() {
    int ch;
    while (1) {
        printf("1.Produce\t2.Consume\t3.Exit\n");
        scanf("%d", &ch);
        switch (ch) {
            case 1:
                if (mutex == 1 && empty != 0) {
                    producer();
                } else {
                    printf("Buffer is full\n");
                }
                break;
            case 2:
                if (mutex == 1 && full != 0) {
                    consumer();
                } else {
                    printf("Buffer is empty\n");
                }
                break;
            case 3:
                exit(0);
        }
    }
}
```

Result:



```
1.Produce       2.Consume       3.Exit
2
Buffer is empty
1.Produce       2.Consume       3.Exit
1
Producer produced 1.
Empty = 4
Buffer:
1       1
1.Produce       2.Consume       3.Exit
1
Producer produced 2.
Empty = 3
Buffer:
1       2       2
1.Produce       2.Consume       3.Exit
2
Consumer consumed 1.
Empty = 4
Buffer:

1.Produce       2.Consume       3.Exit
3
```

Program - 6

Question:

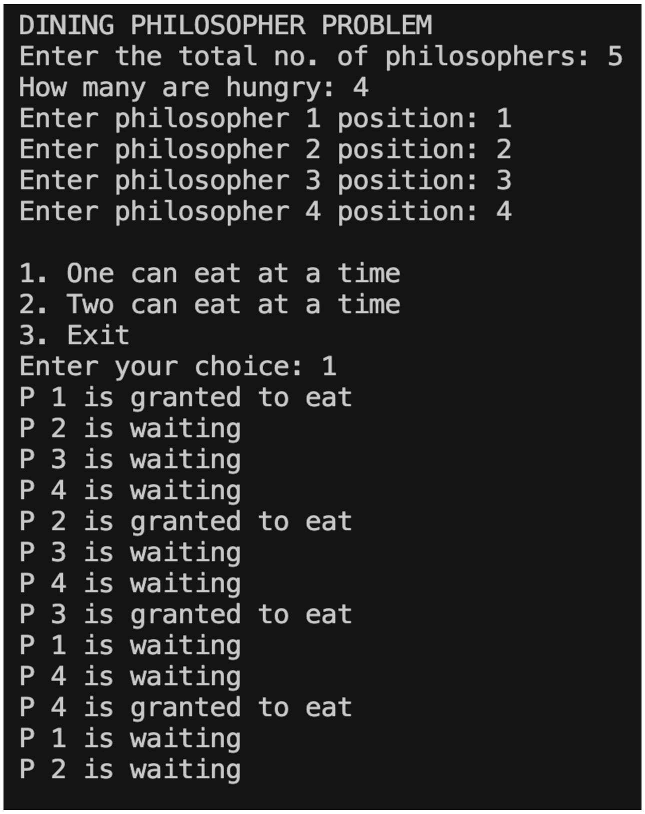
Write a C program to simulate the concept of Dining-Philosophers problem.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_PHILOSOPHERS 5

void allow_one_to_eat(int hungry[], int n) {
   int isWaiting[MAX_PHILOSOPHERS];
   for (int i = 0; i < n; i++) {
      isWaiting[i] = 1;
   }
   for (int i = 0; i < n; i++) {
      printf("P %d is granted to eat\n", hungry[i]);
      isWaiting[hungry[i]] = 0;
      for (int j = 0; j < n; j++) {
         if (isWaiting[hungry[j]]) {
            printf("P %d is waiting\n", hungry[j]);
         }
      }
      for (int k = 0; k < n; k++) {
         isWaiting[k] = 1;
      }
      isWaiting[hungry[i]] = 0;
   }
}

void allow_two_to_eat(int hungry[], int n) {
   if (n < 2 || n > MAX_PHILOSOPHERS) {
      printf("Invalid number of philosophers.\n");
      return;
   }
   for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
         printf("P %d and P %d are granted to eat\n", hungry[i], hungry[j]);
```

```
            for (int k = 0; k < n; k++) {
                if (k != i && k != j) {
                    printf("P %d is waiting\n", hungry[k]);
                }
            }
        }
    }
}

int main() {
    int total_philosophers, hungry_count;
    int hungry_positions[MAX_PHILOSOPHERS];
    printf("DINING PHILOSOPHER PROBLEM\n");
    printf("Enter the total no. of philosophers: ");
    scanf("%d", &total_philosophers);
    if (total_philosophers > MAX_PHILOSOPHERS || total_philosophers < 2) {
        printf("Invalid number of philosophers.\n");
        return 1;
    }
    printf("How many are hungry: ");
    scanf("%d", &hungry_count);
    if (hungry_count < 1 || hungry_count > total_philosophers) {
        printf("Invalid number of hungry philosophers.\n");
        return 1;
    }
    for (int i = 0; i < hungry_count; i++) {
        printf("Enter philosopher %d position: ", i + 1);
        scanf("%d", &hungry_positions[i]);
        if (hungry_positions[i] < 0 || hungry_positions[i] >= total_philosophers) {
            printf("Invalid philosopher position.\n");
            return 1;
        }
    }
    int choice;
    while (1) {
        printf("\n1. One can eat at a time\n");
        printf("2. Two can eat at a time\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                allow_one_to_eat(hungry_positions, hungry_count);
                break;
            case 2:
                allow_two_to_eat(hungry_positions, hungry_count);
                break;
            case 3:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }
    return 0;
}
```

Result:



```
DINING PHILOSOPHER PROBLEM
Enter the total no. of philosophers: 5
How many are hungry: 4
Enter philosopher 1 position: 1
Enter philosopher 2 position: 2
Enter philosopher 3 position: 3
Enter philosopher 4 position: 4

1. One can eat at a time
2. Two can eat at a time
3. Exit
Enter your choice: 1
P 1 is granted to eat
P 2 is waiting
P 3 is waiting
P 4 is waiting
P 2 is granted to eat
P 3 is waiting
P 4 is waiting
P 3 is granted to eat
P 1 is waiting
P 4 is waiting
P 4 is granted to eat
P 1 is waiting
P 2 is waiting
```

Program - 7

Question:

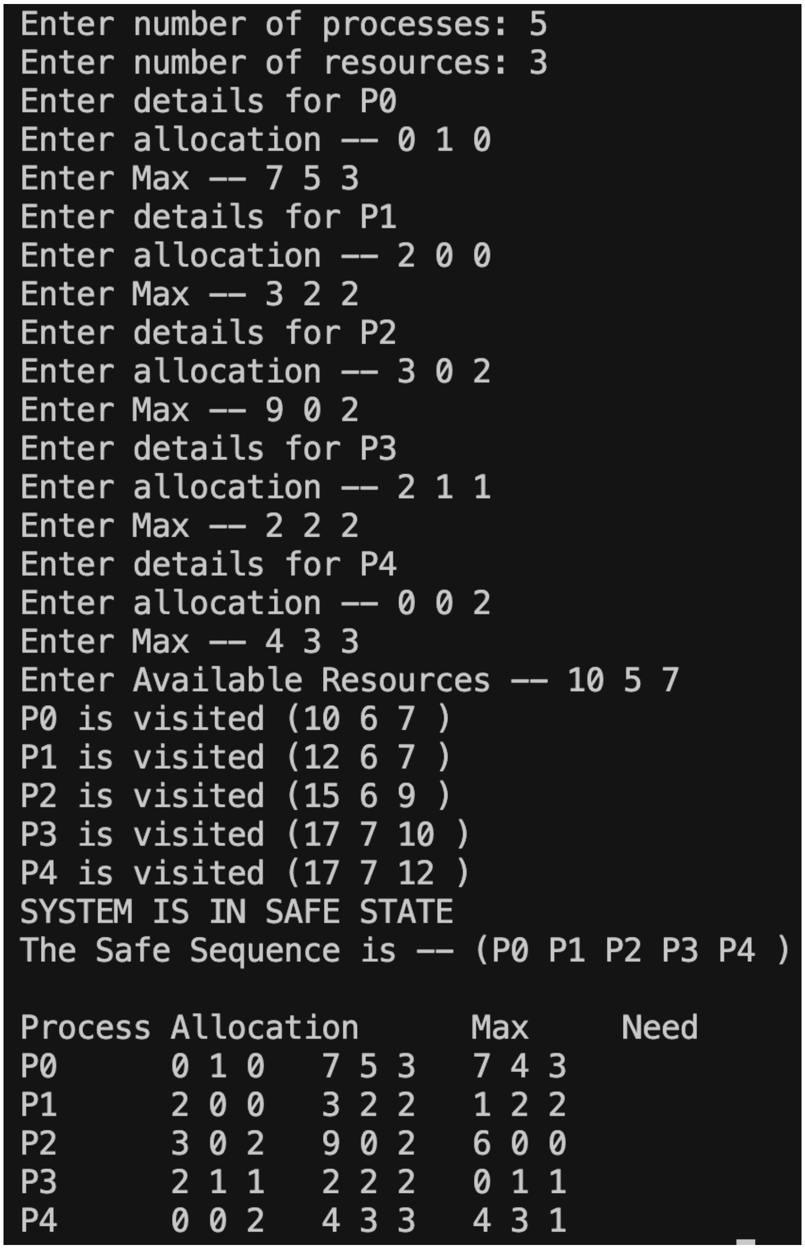
Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

Code:

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

void calculateNeed(int P, int R, int need[P][R], int max[P][R], int allot[P][R]) {
    for (int i = 0; i < P; i++)
        for (int j = 0; j < R; j++)
            need[i][j] = max[i][j] - allot[i][j];
}

bool isSafe(int P, int R, int processes[], int avail[], int max[][R], int allot[][R]) {
    int need[P][R];
    calculateNeed(P, R, need, max, allot);
    bool finish[P];
    for (int i = 0; i < P; i++) {
        finish[i] = 0;
    }
    int safeSeq[P];
    int work[R];
    for (int i = 0; i < R; i++) {
        work[i] = avail[i];
    }
    int count = 0;
    while (count < P) {
        bool found = false;
        for (int p = 0; p < P; p++) {
            if (finish[p] == 0) {
                int j;
                for (j = 0; j < R; j++)
                    if (need[p][j] > work[j])
                        break;
                if (j == R) {
                    printf("P%d is visited (", p);
                    for (int k = 0; k < R; k++) {
                        work[k] += allot[p][k];
                        printf("%d ", work[k]);
                    }
                    printf(")\n");
                    safeSeq[count++] = p;
                    finish[p] = 1;
                    found = true;
                }
            }
        }
        if (found == false) {
            printf("System is not in safe state\n");
            return false;
        }
    }
    printf("SYSTEM IS IN SAFE STATE\nThe Safe Sequence is -- (");
    for (int i = 0; i < P; i++) {
        printf("P%d ", safeSeq[i]);
    }
    printf(")\n");
    return true;
}

int main() {
    int P, R;
    printf("Enter number of processes: ");
    scanf("%d", &P);
    printf("Enter number of resources: ");
    scanf("%d", &R);
    int processes[P];
    int avail[R];
    int max[P][R];
    int allot[P][R];
    for (int i = 0; i < P; i++) {
        processes[i] = i;
    }
    for (int i = 0; i < P; i++) {
        printf("Enter details for P%d\n", i);
```

```
            printf("Enter allocation -- ");
            for (int j = 0; j < R; j++) {
                scanf("%d", &allot[i][j]);
            }
            printf("Enter Max -- ");
            for (int j = 0; j < R; j++) {
                scanf("%d", &max[i][j]);
            }
        }
        printf("Enter Available Resources -- ");
        for (int i = 0; i < R; i++) {
            scanf("%d", &avail[i]);
        }
        isSafe(P, R, processes, avail, max, allot);
        printf("\nProcess\tAllocation\tMax\tNeed\n");
        for (int i = 0; i < P; i++) {
            printf("P%d\t", i);
            for (int j = 0; j < R; j++) {
                printf("%d ", allot[i][j]);
            }
            printf("\t");
            for (int j = 0; j < R; j++) {
                printf("%d ", max[i][j]);
            }
            printf("\t");
            for (int j = 0; j < R; j++) {
                printf("%d ", max[i][j] - allot[i][j]);
            }
            printf("\n");
        }
        return 0;
    }
```

Result:



```
Enter number of processes: 5
Enter number of resources: 3
Enter details for P0
Enter allocation -- 0 1 0
Enter Max -- 7 5 3
Enter details for P1
Enter allocation -- 2 0 0
Enter Max -- 3 2 2
Enter details for P2
Enter allocation -- 3 0 2
Enter Max -- 9 0 2
Enter details for P3
Enter allocation -- 2 1 1
Enter Max -- 2 2 2
Enter details for P4
Enter allocation -- 0 0 2
Enter Max -- 4 3 3
Enter Available Resources -- 10 5 7
P0 is visited (10 6 7 )
P1 is visited (12 6 7 )
P2 is visited (15 6 9 )
P3 is visited (17 7 10 )
P4 is visited (17 7 12 )
SYSTEM IS IN SAFE STATE
The Safe Sequence is -- (P0 P1 P2 P3 P4 )

Process Allocation      Max     Need
P0      0 1 0   7 5 3   7 4 3
P1      2 0 0   3 2 2   1 2 2
P2      3 0 2   9 0 2   6 0 0
P3      2 1 1   2 2 2   0 1 1
P4      0 0 2   4 3 3   4 3 1
```

Program - 8

Question:

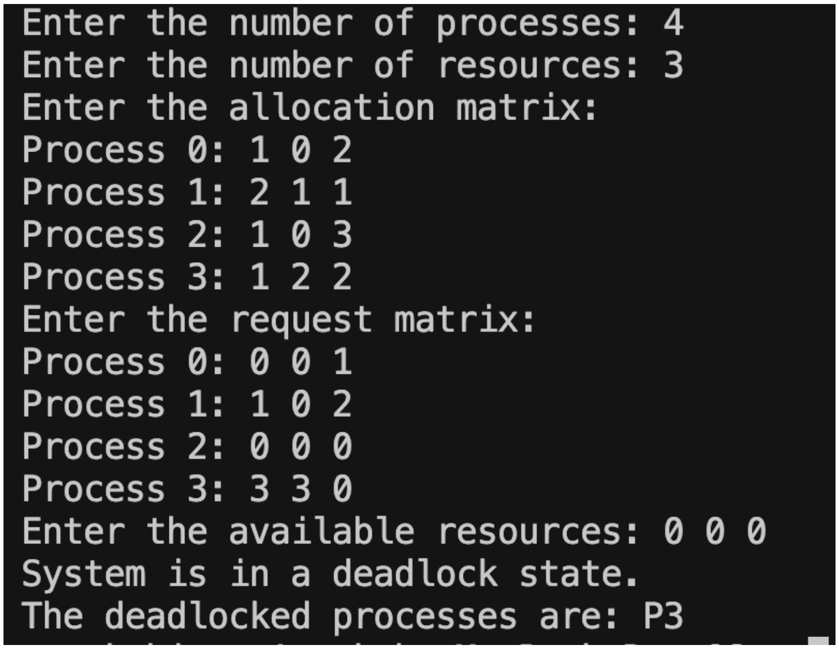
Write a C program to simulate deadlock detection.

Code:

```
#include <stdio.h>

int main() {
    int n, m, i, j, k;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    printf("Enter the number of resources: ");
    scanf("%d", &m);
    int alloc[n][m], request[n][m], avail[m];
    printf("Enter the allocation matrix:\n");
    for (i = 0; i < n; i++) {
        printf("Process %d: ", i);
        for (j = 0; j < m; j++) {
            scanf("%d", &alloc[i][j]);
        }
    }
    printf("Enter the request matrix:\n");
    for (i = 0; i < n; i++) {
        printf("Process %d: ", i);
        for (j = 0; j < m; j++) {
            scanf("%d", &request[i][j]);
        }
    }
    printf("Enter the available resources: ");
    for (j = 0; j < m; j++) {
        scanf("%d", &avail[j]);
    }
    int finish[n], safeSeq[n], work[m], flag;
    for (i = 0; i < n; i++) {
        finish[i] = 0;
    }
    for (j = 0; j < m; j++) {
        work[j] = avail[j];
```

```
}
int count = 0;
while (count < n) {
    flag = 0;
    for (i = 0; i < n; i++) {
        if (finish[i] == 0) {
            int canProceed = 1;
            for (j = 0; j < m; j++) {
                if (request[i][j] > work[j]) {
                    canProceed = 0;
                    break;
                }
            }
            if (canProceed) {
                for (k = 0; k < m; k++) {
                    work[k] += alloc[i][k];
                }
                safeSeq[count++] = i;
                finish[i] = 1;
                flag = 1;
            }
        }
    }
    if (flag == 0) {
        break;
    }
}
int deadlock = 0;
for (i = 0; i < n; i++) {
    if (finish[i] == 0) {
        deadlock = 1;
        printf("System is in a deadlock state.\n");
        printf("The deadlocked processes are: ");
        for (j = 0; j < n; j++) {
            if (finish[j] == 0) {
                printf("P%d ", j);
            }
        }
        printf("\n");
        break;
        }
    }
    if (deadlock == 0) {
        printf("System is not in a deadlock state.\n");
        printf("Safe Sequence is: ");
        for (i = 0; i < n; i++) {
            printf("P%d ", safeSeq[i]);
        }
        printf("\n");
    }
    return 0;
}
```

Result:



```
Enter the number of processes: 4
Enter the number of resources: 3
Enter the allocation matrix:
Process 0: 1 0 2
Process 1: 2 1 1
Process 2: 1 0 3
Process 3: 1 2 2
Enter the request matrix:
Process 0: 0 0 1
Process 1: 1 0 2
Process 2: 0 0 0
Process 3: 3 3 0
Enter the available resources: 0 0 0
System is in a deadlock state.
The deadlocked processes are: P3
```

Program - 9

Question:

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

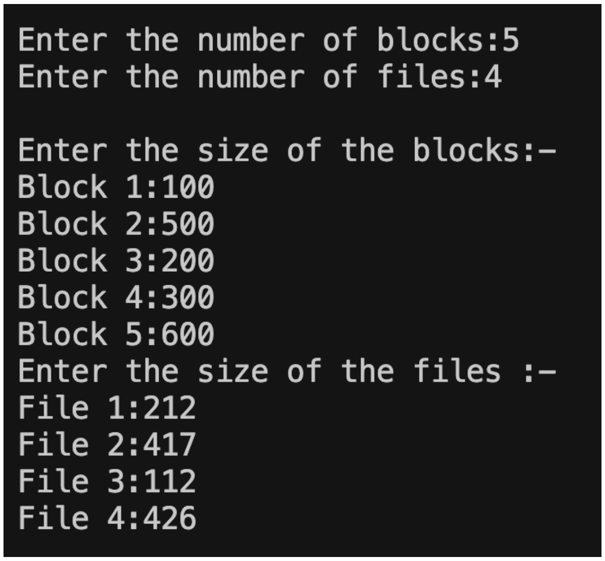
Code:

```
#include <stdio.h>
#define MAX 25
void firstFit(int nb, int nf, int b[], int f[]) {
    int frag[MAX], bf[MAX] = {0}, ff[MAX] = {0};
    int i, j, temp;
    for (i = 1; i <= nf; i++) {
        for (j = 1; j <= nb; j++) {
            if (bf[j] != 1) {
                temp = b[j] - f[i];
                if (temp >= 0) {
                    ff[i] = j;
                    frag[i] = temp;
                    bf[j] = 1;
                    break;
                }
            }
        }
    }

    printf("\nMemory Management Scheme - First Fit\n");
    printf("File_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragment\n");
    for (i = 1; i <= nf; i++) {
        printf("%d\t\t%d\t\t", i, f[i]);
        if (ff[i] != 0) {
            printf("%d\t\t%d\t\t%d\n", ff[i], b[ff[i]], frag[i]);
        } else {
            printf("Not Allocated\n");
        }
    }
```

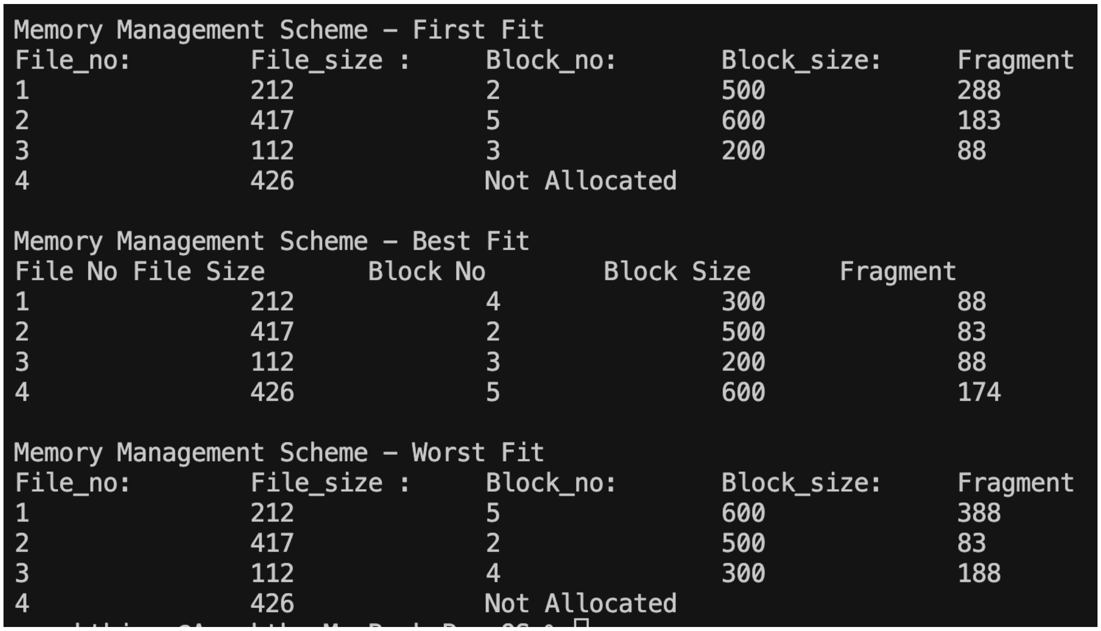
```
}

void bestFit(int nb, int nf, int b[], int f[]) {
    int frag[MAX], bf[MAX] = {0}, ff[MAX] = {0};
    int i, j, temp, lowest = 10000;
    for (i = 1; i <= nf; i++) {
        for (j = 1; j <= nb; j++) {
            if (bf[j] != 1) {
                temp = b[j] - f[i];
                if (temp >= 0 && lowest > temp) {
                    ff[i] = j;
                    lowest = temp;
                }
            }
        }
        frag[i] = lowest;
        bf[ff[i]] = 1;
        lowest = 10000;
    }

    printf("\nMemory Management Scheme - Best Fit\n");
    printf("File No\tFile Size \tBlock No\tBlock Size\tFragment\n");
    for (i = 1; i <= nf; i++) {
        printf("%d\t\t%d\t\t", i, f[i]);
        if (ff[i] != 0) {
            printf("%d\t\t%d\t\t%d\n", ff[i], b[ff[i]], frag[i]);
        } else {
            printf("Not Allocated\n");
        }
    }
}

void worstFit(int nb, int nf, int b[], int f[]) {
    int frag[MAX], bf[MAX] = {0}, ff[MAX] = {0};
    int i, j, temp, highest = 0;
    for (i = 1; i <= nf; i++) {
        for (j = 1; j <= nb; j++) {
            if (bf[j] != 1) {
                temp = b[j] - f[i];
                if (temp >= 0 && highest < temp) {
```

```
                ff[i] = j;
                highest = temp;
            }
        }
    }
    frag[i] = highest;
    bf[ff[i]] = 1;
    highest = 0;
  }

  printf("\nMemory Management Scheme - Worst Fit\n");
  printf("File_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragment\n");
  for (i = 1; i <= nf; i++) {
    printf("%d\t\t%d\t\t", i, f[i]);
    if (ff[i] != 0) {
      printf("%d\t\t%d\t\t%d\n", ff[i], b[ff[i]], frag[i]);
    } else {
      printf("Not Allocated\n");
    }
  }
}

int main() {
  int b[MAX], f[MAX], nb, nf;
  printf("\nEnter the number of blocks:");
  scanf("%d", &nb);
  printf("Enter the number of files:");
  scanf("%d", &nf);
  printf("\nEnter the size of the blocks:-\n");
  for (int i = 1; i <= nb; i++) {
    printf("Block %d:", i);
    scanf("%d", &b[i]);
  }
  printf("Enter the size of the files :-\n");
  for (int i = 1; i <= nf; i++) {
    printf("File %d:", i);
    scanf("%d", &f[i]);
  }
  int b1[MAX], b2[MAX], b3[MAX];
  for (int i = 1; i <= nb; i++) {
        b1[i] = b[i];
        b2[i] = b[i];
        b3[i] = b[i];
    }
    firstFit(nb, nf, b1, f);
    bestFit(nb, nf, b2, f);
    worstFit(nb, nf, b3, f);
    return 0;
}
```

Result:



```
Enter the number of blocks:5
Enter the number of files:4

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
```



```
Memory Management Scheme - First Fit
File_no:        File_size :     Block_no:       Block_size:     Fragment
1               212             2               500             288
2               417             5               600             183
3               112             3               200             88
4               426             Not Allocated

Memory Management Scheme - Best Fit
File No File Size       Block No        Block Size      Fragment
1               212             4               300             88
2               417             2               500             83
3               112             3               200             88
4               426             5               600             174

Memory Management Scheme - Worst Fit
File_no:        File_size :     Block_no:       Block_size:     Fragment
1               212             5               600             388
2               417             2               500             83
3               112             4               300             188
4               426             Not Allocated
```

Program - 10

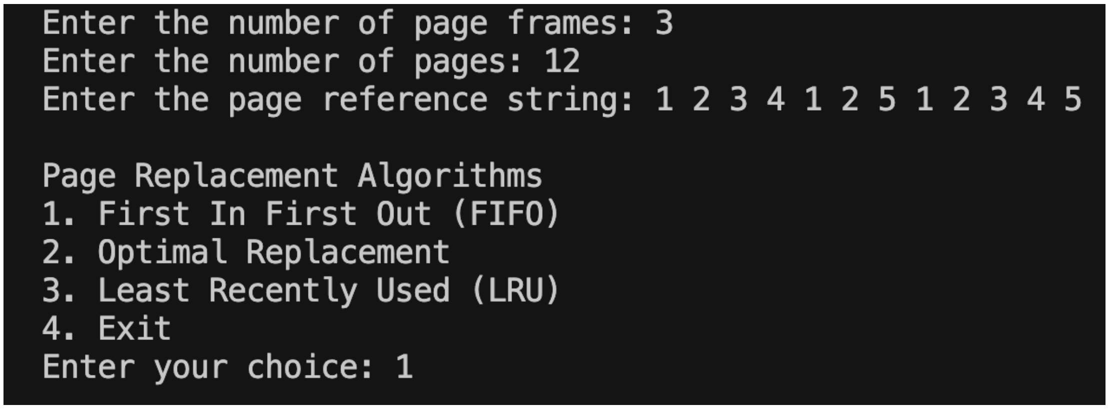
Question :

Write a C program to simulate paging technique of memory management.

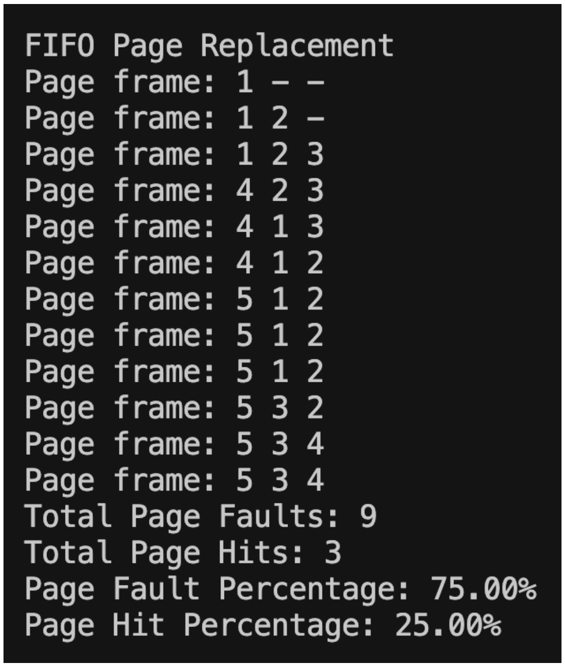
Code:

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

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

int main() {
    int n, f, choice;
    printf("Enter the number of page frames: ");
    scanf("%d", &f);
    printf("Enter the number of pages: ");
    scanf("%d", &n);
    int pages[n];
    printf("Enter the page reference string: ");
    for(int i = 0; i < n; i++) {
        scanf("%d", &pages[i]);
    }
    while(1) {
        printf("\nPage Replacement Algorithms\n");
        printf("1. First In First Out (FIFO)\n");
        printf("2. Optimal Replacement\n");
        printf("3. Least Recently Used (LRU)\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch(choice) {
            case 1:
                fifo(pages, n, f);
                break;
            case 2:
                optimal(pages, n, f);
                break;
            case 3:
                lru(pages, n, f);
                break;
            case 4:
                exit(0);
                break;
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }
    return 0;
}

void fifo(int pages[], int n, int f) {
    int frame[f];
    for(int i = 0; i < f; i++)
        frame[i] = -1;
    int front = 0, pf = 0, ph = 0;
    printf("\nFIFO Page Replacement\n");
    for(int i = 0; i < n; i++) {
        int found = 0;
        for(int j = 0; j < f; j++) {
            if(frame[j] == pages[i]) {
                found = 1;
                ph++;
                break;          }
        }
        if(!found) {
            frame[front] = pages[i];
            front = (front + 1) % f;
            pf++;
        }
        printf("Page frame: ");
        for(int j = 0; j < f; j++) {
            if(frame[j] != -1)
                printf("%d ", frame[j]);
            else
                printf("- ");
        }
        printf("\n");
    }
    printf("Total Page Faults: %d\n", pf);
    printf("Total Page Hits: %d\n", ph);
    printf("Page Fault Percentage: %.2f%%\n", ((float)pf / n) * 100);
    printf("Page Hit Percentage: %.2f%%\n", ((float)ph / n) * 100);
}

void optimal(int pages[], int n, int f) {
    int frame[f];
    for(int i = 0; i < f; i++)
        frame[i] = -1;
    int pf = 0, ph = 0;
    printf("\nOptimal Page Replacement\n");
    for(int i = 0; i < n; i++) {
        int found = 0;
        for(int j = 0; j < f; j++) {
            if(frame[j] == pages[i]) {
                found = 1;
                ph++;
                break;
            }
        }
        if(!found) {
            if(i < f) {
                frame[i] = pages[i];
            } else {
                int farthest = i, replace = 0;
                for(int j = 0; j < f; j++) {
                    int k;
                    for(k = i + 1; k < n; k++) {
                        if(frame[j] == pages[k]) {
                            if(k > farthest) {
                                farthest = k;
                                replace = j;
                            }
                            break;
                        }
                    }
                    if(k == n) {
                        replace = j;
                        break;
                    }
                }
                frame[replace] = pages[i];
            }
            pf++;
        }
        printf("Page frame: ");
        for(int j = 0; j < f; j++) {
            if(frame[j] != -1)
                printf("%d ", frame[j]);
            else
                printf("- ");
        }
        printf("\n");
    }
    printf("Total Page Faults: %d\n", pf);
    printf("Total Page Hits: %d\n", ph);
    printf("Page Fault Percentage: %.2f%%\n", ((float)pf / n) * 100);
    printf("Page Hit Percentage: %.2f%%\n", ((float)ph / n) * 100);
}

void lru(int pages[], int n, int f) {
    int frame[f], cnt[f];
    for(int i = 0; i < f; i++) {
        frame[i] = -1;
        cnt[i] = 0;
    }
    int time = 0, pf = 0, ph = 0;
    printf("\nLRU Page Replacement\n");
    for(int i = 0; i < n; i++) {
        int found = 0, pos = -1, min = time;
        for(int j = 0; j < f; j++) {
            if(frame[j] == pages[i]) {
                found = 1;
                cnt[j] = time;
                ph++;
                break;
            }
        }
        if(!found) {
            for(int j = 0; j < f; j++) {
                if(frame[j] == -1) {
                    pos = j;
                    break;
                }
                if(cnt[j] < min) {
                    min = cnt[j];
                    pos = j;
                }
            }
            frame[pos] = pages[i];
            cnt[pos] = time;
            pf++;
        }
        time++;
        printf("Page frame: ");
        for(int j = 0; j < f; j++) {
            if(frame[j] != -1)
                printf("%d ", frame[j]);
            else
                printf("- ");
        }
        printf("\n");
    }
    printf("Total Page Faults: %d\n", pf);
    printf("Total Page Hits: %d\n", ph);
    printf("Page Fault Percentage: %.2f%%\n", ((float)pf / n) * 100);
    printf("Page Hit Percentage: %.2f%%\n", ((float)ph / n) * 100);
}
```

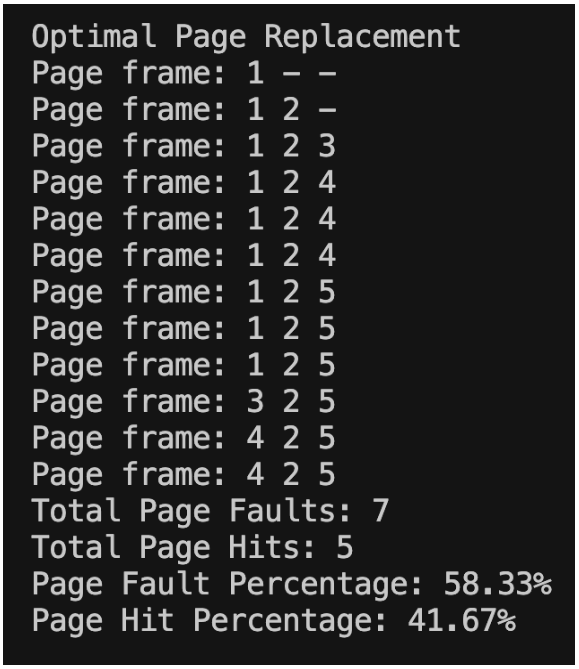
Result:



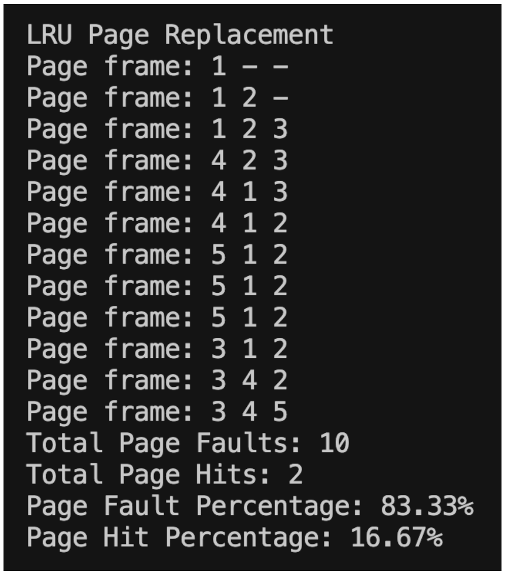
```
Enter the number of page frames: 3
Enter the number of pages: 12
Enter the page reference string: 1 2 3 4 1 2 5 1 2 3 4 5

Page Replacement Algorithms
1. First In First Out (FIFO)
2. Optimal Replacement
3. Least Recently Used (LRU)
4. Exit
Enter your choice: 1
```



```
FIFO Page Replacement
Page frame: 1 - -
Page frame: 1 2 -
Page frame: 1 2 3
Page frame: 4 2 3
Page frame: 4 1 3
Page frame: 4 1 2
Page frame: 5 1 2
Page frame: 5 1 2
Page frame: 5 1 2
Page frame: 5 3 2
Page frame: 5 3 4
Page frame: 5 3 4
Total Page Faults: 9
Total Page Hits: 3
Page Fault Percentage: 75.00%
Page Hit Percentage: 25.00%
```



```
Optimal Page Replacement
Page frame: 1 - -
Page frame: 1 2 -
Page frame: 1 2 3
Page frame: 1 2 4
Page frame: 1 2 4
Page frame: 1 2 4
Page frame: 1 2 5
Page frame: 1 2 5
Page frame: 1 2 5
Page frame: 3 2 5
Page frame: 4 2 5
Page frame: 4 2 5
Total Page Faults: 7
Total Page Hits: 5
Page Fault Percentage: 58.33%
Page Hit Percentage: 41.67%
```



```
LRU Page Replacement
Page frame: 1 - -
Page frame: 1 2 -
Page frame: 1 2 3
Page frame: 4 2 3
Page frame: 4 1 3
Page frame: 4 1 2
Page frame: 5 1 2
Page frame: 5 1 2
Page frame: 5 1 2
Page frame: 3 1 2
Page frame: 3 4 2
Page frame: 3 4 5
Total Page Faults: 10
Total Page Hits: 2
Page Fault Percentage: 83.33%
Page Hit Percentage: 16.67%
```