VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS

Submitted by

SAMRAAT DABOLAY (1BM22CS236)

Under the Guidance of
Sunayana S
Assistant Professor
Department of CSE

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 **SAMRAAT DABOLAY (1BM22CS236),** who is a bonafide student of **B. M. S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 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.

Sunayana S
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 Systems.
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.

a) FCFS
b) SJF

Code:

FCFS and SJF(Non Preemptive)

```
#include <stdio.h>
int n, i, j, pos, temp, choice, total = 0;
int Burst_time[20], Arrival_time[20], Waiting_time[20], Turn_around_time[20], process[20];
float avg_Turn_around_time = 0, avg_Waiting_time = 0;

void FCFS() {
  int total_waiting_time = 0, total_turnaround_time = 0;
  int current_time = 0;


  for (i = 0; i < n - 1; i++) {
    for (j = i + 1; j < n; j++) {
      if (Arrival_time[i] > Arrival_time[j]) {
        temp = Arrival_time[i];
        Arrival_time[i] = Arrival_time[j];
        Arrival_time[j] = temp;

        temp = Burst_time[i];
        Burst_time[i] = Burst_time[j];
        Burst_time[j] = temp;

        temp = process[i];
        process[i] = process[j];
        process[j] = temp;
      }
    }
  }
```

```
    Waiting_time[0] = 0;
    current_time = Arrival_time[0] + Burst_time[0];

    for (i = 1; i < n; i++) {
        if (current_time < Arrival_time[i]) {
            current_time = Arrival_time[i];
        }
        Waiting_time[i] = current_time - Arrival_time[i];
        current_time += Burst_time[i];
        total_waiting_time += Waiting_time[i];
    }

    printf("\nProcess\t\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time");
    for (i = 0; i < n; i++) {
        Turn_around_time[i] = Burst_time[i] + Waiting_time[i];
        total_turnaround_time += Turn_around_time[i];
        printf("\nP[%d]\t\t%d\t\t%d\t\t%d\t\t%d", process[i], Arrival_time[i], Burst_time[i],
Waiting_time[i], Turn_around_time[i]);
    }

    avg_Waiting_time = (float)total_waiting_time / n;
    avg_Turn_around_time = (float)total_turnaround_time / n;
    printf("\nAverage Waiting Time: %.2f", avg_Waiting_time);
    printf("\nAverage Turnaround Time: %.2f\n", avg_Turn_around_time);
}

void SJF() {
    int total_waiting_time = 0, total_turnaround_time = 0;
    int completed = 0, current_time = 0, min_index;
    int is_completed[20] = {0};

    while (completed != n) {
        int min_burst_time = 9999;
        min_index = -1;

        for (i = 0; i < n; i++) {
            if (Arrival_time[i] <= current_time && is_completed[i] == 0) {
                if (Burst_time[i] < min_burst_time) {
                    min_burst_time = Burst_time[i];
                    min_index = i;
                }
                if (Burst_time[i] == min_burst_time) {
                    if (Arrival_time[i] < Arrival_time[min_index]) {
```

```
                min_burst_time = Burst_time[i];
                min_index = i;
            }
        }
    }
}

if (min_index != -1) {
    Waiting_time[min_index] = current_time - Arrival_time[min_index];
    current_time += Burst_time[min_index];
    Turn_around_time[min_index] = current_time - Arrival_time[min_index];
    total_waiting_time += Waiting_time[min_index];
    total_turnaround_time += Turn_around_time[min_index];
    is_completed[min_index] = 1;
    completed++;
} else {
    current_time++;
}
}

printf("\nProcess\t\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time");
for (i = 0; i < n; i++) {
    printf("\nP[%d]\t\t%d\t\t%d\t\t%d\t\t%d", process[i], Arrival_time[i], Burst_time[i],
Waiting_time[i], Turn_around_time[i]);
}

avg_Waiting_time = (float)total_waiting_time / n;
avg_Turn_around_time = (float)total_turnaround_time / n;
printf("\n\nAverage Waiting Time = %.2f", avg_Waiting_time);
printf("\nAverage Turnaround Time = %.2f\n", avg_Turn_around_time);
}

int main() {
    printf("Enter the total number of processes: ");
    scanf("%d", &n);
    printf("\nEnter Arrival Time and Burst Time:\n");
    for (i = 0; i < n; i++) {
        printf("P[%d] Arrival Time: ", i + 1);
        scanf("%d", &Arrival_time[i]);
        printf("P[%d] Burst Time: ", i + 1);
        scanf("%d", &Burst_time[i]);
        process[i] = i + 1;
    }
    while (1) {
```

```
        printf("\n-----MAIN MENU-----\n");
        printf("1. FCFS Scheduling\n2. SJF Scheduling\n");
        printf("\nEnter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1: FCFS();
                    break;
            case 2: SJF();
                    break;
            default: printf("Invalid Input!!!\n");
        }
    }
    return 0;
}
```

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:

a) FCFS

```
Enter the total number of processes: 5

Enter Arrival Time and Burst Time:
P[1] Arrival Time: 0
P[1] Burst Time: 10
P[2] Arrival Time: 0
P[2] Burst Time: 1
P[3] Arrival Time: 3
P[3] Burst Time: 2
P[4] Arrival Time: 5
P[4] Burst Time: 1
P[5] Arrival Time: 10
P[5] Burst Time: 5

-----MAIN MENU-----
1. FCFS Scheduling
2. SJF Scheduling

Enter your choice: 1

Process     Arrival Time    Burst Time  Waiting Time    Turnaround Time
P[1]        0       10      0       10
P[2]        0       1       10      11
P[3]        3       2       8       10
P[4]        5       1       8       9
P[5]        10      5       4       9
Average Waiting Time: 6.00
Average Turnaround Time: 9.80
```

b) SJF (Non Preemptive)

```
Enter the total number of processes: 4

Enter Arrival Time and Burst Time:
P[1] Arrival Time: 0
P[1] Burst Time: 3
P[2] Arrival Time: 1
P[2] Burst Time: 6
P[3] Arrival Time: 4
P[3] Burst Time: 4
P[4] Arrival Time: 6
P[4] Burst Time: 2

-----MAIN MENU-----
1. FCFS Scheduling
2. SJF Scheduling

Enter your choice: 2

Process     Arrival Time    Burst Time  Waiting Time    Turnaround Time
P[1]        0       3       0       3
P[2]        1       6       2       8
P[3]        4       4       7       11
P[4]        6       2       3       5

Average Waiting Time = 3.00
Average Turnaround Time = 6.75
```

c) SJF (Preemptive)

```
Enter the total number of processes: 4
Enter Arrival Time and Burst Time for each process:
Process 1:
Arrival Time: 03
Burst Time: 3
Process 2:
Arrival Time: 0
Burst Time: 4
Process 3:
Arrival Time: 5
Burst Time: 7
Process 4:
Arrival Time: 6
Burst Time: 3

Process Arrival Time    Burst Time  Completion Time Waiting Time    Turnaround Time
1   0       3       3       0       3
2   0       4       7       3       7
3   5       7       17      5       12
4   6       3       10      1       4


=== Code Execution Successful ===
```

Program -2

Question:

Write a C program to simulate the following CPU scheduling algorithm to find

turnaround time and waiting time.

a) Priority (preemptive & Non-pre-emptive)
b) Round Robin (Experiment with different quantum sizes for RR algorithm)

Code:

(a) Priority (Non-pre-emptive)

```
#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;
}
```

Priority (Pre-emptive):

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

struct process {
    int process_id;
    int burst_time;
    int priority;
    int arrival_time;
    int remaining_time;
    int waiting_time;
    int turnaround_time;
    int is_completed;
};

void find_average_time(struct process[], int);
void priority_scheduling(struct process[], int);

int main() {
    int n, i;
    struct process proc[10];

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

```
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        printf("\nEnter the process ID: ");
        scanf("%d", &proc[i].process_id);

        printf("Enter the burst time: ");
        scanf("%d", &proc[i].burst_time);

        printf("Enter the arrival time: ");
        scanf("%d", &proc[i].arrival_time);

        printf("Enter the priority: ");
        scanf("%d", &proc[i].priority);

        proc[i].remaining_time = proc[i].burst_time;
        proc[i].is_completed = 0;
    }

    priority_scheduling(proc, n);
    return 0;
}

void find_waiting_time(struct process proc[], int n) {
    int time = 0, completed = 0, min_priority, shortest = 0;
    while (completed != n) {
        min_priority = 10000;
        for (int i = 0; i < n; i++) {
            if ((proc[i].arrival_time <= time) && (!proc[i].is_completed) && (proc[i].priority <
min_priority)) {
                min_priority = proc[i].priority;
                shortest = i;
            }
        }
        proc[shortest].remaining_time--;
```

```
        time++;
        if (proc[shortest].remaining_time == 0) {
            proc[shortest].waiting_time = time - proc[shortest].arrival_time -
proc[shortest].burst_time;
            proc[shortest].turnaround_time = time - proc[shortest].arrival_time;
            proc[shortest].is_completed = 1;
            completed++;
        }
    }
}

void find_turnaround_time(struct process proc[], int n) {
    // Turnaround time is calculated during the find_waiting_time function
}

void find_average_time(struct process proc[], int n) {
    int total_wt = 0, total_tat = 0;
    find_waiting_time(proc, n);
    find_turnaround_time(proc, n);

    printf("\nProcess ID\tBurst Time\tArrival Time\tPriority\tWaiting Time\tTurnaround Time");

    for (int i = 0; i < n; i++) {
        total_wt += proc[i].waiting_time;
        total_tat += proc[i].turnaround_time;
        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d", proc[i].process_id, proc[i].burst_time,
proc[i].arrival_time, proc[i].priority, proc[i].waiting_time, proc[i].turnaround_time);
    }
    printf("\n\nAverage Waiting Time = %f", (float)total_wt / n);
    printf("\nAverage Turnaround Time = %f\n", (float)total_tat / n);
}

void priority_scheduling(struct process proc[], int n) {
    find_average_time(proc, n);
}
```

(b) Round Robin (Non-pre-emptive)

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

void main()
{
   // initlialize the variable name
   int i, NOP, sum=0,count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];
   float avg_wt, avg_tat;
   printf(" Total number of process in the system: ");
   scanf("%d", &NOP);
   y = NOP; // Assign the number of process to variable y

// Use for loop to enter the details of the process like Arrival time and the Burst Time
for(i=0; i<NOP; i++)
{
printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1);
printf(" Arrival time is: \t");  // Accept arrival time
scanf("%d", &at[i]);
printf(" \nBurst time is: \t"); // Accept the Burst time
scanf("%d", &bt[i]);
temp[i] = bt[i]; // store the burst time in temp array
}
// Accept the Time qunat
printf("Enter the Time Quantum for the process: \t");
scanf("%d", &quant);
// Display the process No, burst time, Turn Around Time and the waiting time
printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");
for(sum=0, i = 0; y!=0; )
{
if(temp[i] <= quant && temp[i] > 0) // define the conditions
{
   sum = sum + temp[i];
   temp[i] = 0;
   count=1;
   }
   else if(temp[i] > 0)
   {
      temp[i] = temp[i] - quant;
      sum = sum + quant;
   }
```

```
    if(temp[i]==0 && count==1)
    {
        y--; //decrement the process no.
        printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);
        wt = wt+sum-at[i]-bt[i];
        tat = tat+sum-at[i];
        count =0;
    }
    if(i==NOP-1)
    {
        i=0;
    }
    else if(at[i+1]<=sum)
    {
        i++;
    }
    else
    {
        i=0;
    }
}
// represents the average waiting time and Turn Around time
avg_wt = wt * 1.0/NOP;
avg_tat = tat * 1.0/NOP;
printf("\n Average Turn Around Time: \t%f", avg_wt);
printf("\n Average Waiting Time: \t%f", avg_tat);
getch();
}
```

Result:

a)

```
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
```

```
Enter the number of processes: 5
Enter the process ID: 5
Enter the burst time: 2
Enter the arrival time: 4
Enter the priority: 5
Enter the process ID: 1
Enter the burst time: 4
Enter the arrival time: 0
Enter the priority: 2
Enter the process ID: 2
Enter the burst time: 3
Enter the arrival time: 1
Enter the priority: 3
Enter the process ID: 3
Enter the burst time: 1
Enter the arrival time: 2
Enter the priority: 4
Enter the process ID: 4
Enter the burst time: 5
Enter the arrival time: 3
Enter the priority: 5

Process ID      Burst Time      Arrival Time    Priority        Waiting Time    Turnaround Time
5               2               4               5               4               6
1               4               0               2               0               4
2               3               1               3               3               6
3               1               2               4               5               6
4               5               3               5               7               12

Average Waiting Time = 3.800000
Average Turnaround Time = 6.800000
```

b)

```
 Total number of process in the system: 4

 Enter the Arrival and Burst time of the Process[1]
 Arrival time is:       0

Burst time is:  5

 Enter the Arrival and Burst time of the Process[2]
 Arrival time is:       1

Burst time is:  4

 Enter the Arrival and Burst time of the Process[3]
 Arrival time is:       2

Burst time is:  2

 Enter the Arrival and Burst time of the Process[4]
 Arrival time is:       4

Burst time is:  1
Enter the Time Quantum for the process:         2

 Process No              Burst Time              TAT             Waiting Time
Process No[3]           2                               4                       2
Process No[4]           1                               3                       2
Process No[2]           4                               10                      6
Process No[1]           5                               12                      7
 Average Turn Around Time:      4.250000
 Average Waiting Time:  7.250000
```

Program -3

Question:

Write a C program to simulate a 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: 4
Enter arrival time, burst time and type (0 for system, 1 for user) for process 1: 0 4 1
Enter arrival time, burst time and type (0 for system, 1 for user) for process 2: 1 3 1
Enter arrival time, burst time and type (0 for system, 1 for user) for process 3: 0 4 1
Enter arrival time, burst time and type (0 for system, 1 for user) for process 4: 3 2 0
System Processes Scheduling:
PID     AT      BT      CT      TAT     WT
4       3       2       5       2       0
Average Turnaround Time: 2.00 ms
Average Waiting Time: 0.00 ms

User Processes Scheduling:
PID     AT      BT      CT      TAT     WT
1       0       4       9       9       5
3       0       4       13      13      9
2       1       3       16      15      12
Average Turnaround Time: 12.33 ms
Average Waiting Time: 8.67 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++;
  }
}
```

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++;
  }
}
```

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:

a) Rate Monotonic

```
Enter the number of processes:2
Enter the CPU burst times:
4
5
Enter the time periods:
3
4
LCM=12

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

2.583333 <= 0.828427 =>false
```

```
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

```
Enter the CPU burst times:
4
5
Enter the deadlines:
2
3
Enter the time periods:
10
20

Earliest Deadline Scheduling:
PID      Burst  Deadline        Period
1               4               2               10
2               5               3               20
Scheduling occurs for 20 ms

0ms : Task 1 is running.
1ms : Task 1 is running.
2ms : Task 1 is running.
3ms : Task 1 is running.
4ms : Task 2 is running.
5ms : Task 2 is running.
6ms : Task 2 is running.
7ms : Task 2 is running.
8ms : Task 2 is running.
9ms: CPU is idle.
10ms : Task 1 is running.
11ms : Task 1 is running.
12ms : Task 1 is running.
13ms : Task 1 is running.
14ms: CPU is idle.
15ms: CPU is idle.
16ms: CPU is idle.
17ms: CPU is idle.
18ms: CPU is idle.
19ms: CPU is idle.
```

c) Proportional Scheduling

```
Enter number of processes:3
Enter tickets of the processes:
5 15 20
The winning number is 9 and winning participant is: 2
The winning number is 9 and winning participant is: 3
The winning number is 1 and winning participant is: 1

Probabilities:
The probability of P1 winning: 12.50
The probability of P2 winning: 37.50
The probability of P3 winning: 50.00
```

Program -5

Question:

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

Code:

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


int mutex=1,full=0,empty=3,x=0;


int main()
{
   int n;
   void producer();
   void consumer();
   int wait(int);
   int signal(int);
   printf("\n1.Producer\n2.Consumer\n3.Exit");
   while(1)
   {
      printf("\nEnter your choice: ");
      scanf("%d",&n);
      switch(n)
      {
         case 1: if((mutex==1)&&(empty!=0))
               producer();
               else
               printf("Buffer is full!!");
               break;
         case 2: if((mutex==1)&&(full!=0))
               consumer();
               else
               printf("Buffer is empty!!");
```

```
                break;
          case 3: exit(0);
                break;
       }
   }
   return 0;
}

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

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

void producer()
{
   mutex=wait(mutex);
   full=signal(full);
   empty=wait(empty);
   x++;
   printf("\nProducer produces the item %d",x);
   mutex=signal(mutex);
}

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

```
  printf("\nConsumer consumes item %d",x);
  x--;
  mutex=signal(mutex);
}
```

Result:

```
1.Producer
2.Consumer
3.Exit
Enter your choice: 1

Producer produces the item 1
Enter your choice: 1

Producer produces the item 2
Enter your choice: 2

Consumer consumes item 2
Enter your choice: 2

Consumer consumes item 1
Enter your choice: 2
Buffer is empty!!
Enter your choice: 1

Producer produces the item 1
Enter your choice: 2

Consumer consumes item 1
Enter your choice: 3
```

Program -6

Question:

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

Code:

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

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

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

sem_t mutex;
sem_t S[N];

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

			sleep(2);

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

			printf("Philosopher %d is Eating\n", i +1);

			sem_post(&S[i]);
	}
}

void take_fork(int i)
```

```
{
        sem_wait(&mutex);
        state[i] = HUNGRY;
        printf("Philosopher %d is Hungry\n",i+1);
        test(i);

        sem_post(&mutex);
        sem_wait(&S[i]);
        sleep(1);
}

void put_fork(int i)
{
        sem_wait(&mutex);
        state[i] = THINKING;

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

        printf("Philosopher %d is thinking\n", i+1);
        test(LEFT);
        test(RIGHT);
        sem_post(&mutex);
}

void* philosopher(void* num)
{

        while (1)
        {
                int* i = num;
                sleep(1);
                take_fork(*i);
                sleep(0);
                put_fork(*i);
        }
}

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);
	}
}
```

Result:

```
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 1 is Hungry
Philosopher 3 is Hungry
Philosopher 5 is Hungry
Philosopher 2 is Hungry
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 4 is Hungry
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 2 is Hungry
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
```

Program -7

Question:

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

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

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

   int available[m];
   printf("Enter the Available Resources:\n");
   for (i = 0; i < m; i++)
   {
```

```
    scanf("%d", &available[i]);
}

int f[n], ans[n], ind = 0;
for (k = 0; k < n; k++)
{
    f[k] = 0;
}

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

int y = 0;
for (k = 0; k < n; k++)
{
    for (i = 0; i < n; i++)
    {
        if (f[i] == 0)
        {
            int flag = 0;
            for (j = 0; j < m; j++)
            {
                if (need[i][j] > available[j])
                {
                    flag = 1;
                    break;
                }
            }

            if (flag == 0)
            {
                ans[ind++] = i;
                for (y = 0; y < m; y++)
                {
                    available[y] += allocation[i][y];
                }
                f[i] = 1;
            }
```

```
        }
      }
    }

    int flag = 1;
    for (i = 0; i < n; i++)
    {
      if (f[i] == 0)
      {
        flag = 0;
        printf("The following system is not safe\n");
        break;
      }
    }

    if (flag == 1)
    {
      printf("Following is the SAFE Sequence\n");
      for (i = 0; i < n - 1; i++)
      {
        printf(" P%d ->", ans[i]);
      }
      printf(" P%d\n", ans[n - 1]);
    }
    return 0;
}
```

Result:

```
Enter the number of processes: 5
Enter the number of resources: 3
Enter the Allocation Matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter the MAX Matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter the Available Resources:
3 3 2
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P2
```

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

int main()
{
   int b[max], f[max], nb, nf;

   printf("Memory Management Schemes\n");

   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("\nEnter the size of the files:\n");
   for (int i = 1; i <= nf; i++)
   {
```

```
        printf("File %d:", i);
        scanf("%d", &f[i]);
    }

    printf("\nMemory Management Scheme - First Fit");
    firstFit(b, nb, f, nf);

    printf("\n\nMemory Management Scheme - Worst Fit");
    worstFit(b, nb, f, nf);

    printf("\n\nMemory Management Scheme - Best Fit");
    bestFit(b, nb, f, nf);

    return 0;
}

void firstFit(int b[], int nb, int f[], int nf)
{
    int bf[max] = {0};
    int ff[max] = {0};
    int frag[max], i, j;

    for (i = 1; i <= nf; i++)
    {
        for (j = 1; j <= nb; j++)
        {
            if (bf[j] != 1 && b[j] >= f[i])
            {
                ff[i] = j;
                bf[j] = 1;
                frag[i] = b[j] - f[i];
                break;
            }
        }
    }

    printf("\nFile_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment");
    for (i = 1; i <= nf; i++)
    {
        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]);
    }
}

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

```
{
   int bf[max] = {0};
   int ff[max] = {0};
   int frag[max], 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("\nFile_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment");
   for (i = 1; i <= nf; i++)
   {
      printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]);
   }
}

void bestFit(int b[], int nb, int f[], int nf)
{
   int bf[max] = {0};
   int ff[max] = {0};
   int frag[max], 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("\nFile_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment");
   for (i = 1; i <= nf && ff[i] != 0; i++)
   {
      printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]);
   }
}
```

Result:

```
Memory Management Schemes

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

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 2:415
File 3:63
File 4:200
File 5:255
```

```
Memory Management Scheme - First Fit
File_no:        File_size:      Block_no:       Block_size:     Fragment
1               212             2               500             288
2               415             5               600             185
3               63              1               100             37
4               200             3               200             0
5               255             4               300             45

Memory Management Scheme - Worst Fit
File_no:        File_size:      Block_no:       Block_size:     Fragment
1               212             5               600             388
2               415             2               500             85
3               63              4               300             237
4               200             0               0               0
5               255             0               0               0

Memory Management Scheme - Best Fit
File_no:        File_size:      Block_no:       Block_size:     Fragment
1               212             4               300             88
2               415             2               500             85
3               63              1               100             37
4               200             3               200             0
5               255             5               600             345
```

Program -10

Question:

Write a C program to simulate page replacement algorithms

a) FIFO
b) LRU
c) Optimal

Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <limits.h>


// Function to check if a page is present in memory

int isPresent(int memory[], int n, int page) {

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

        if (memory[i] == page) {

            return 1;

        }

    }

    return 0;

}


// Function to find the index of the least recently used page

int findLRU(int time[], int n) {

    int minimum = time[0], index = 0;

    for (int i = 1; i < n; i++) {

        if (time[i] < minimum) {
```

```
            minimum = time[i];
            index = i;
        }
    }
    return index;
}

// FIFO Page Replacement Algorithm
int fifo(int pages[], int n, int capacity) {
    int memory[capacity]; // Array to store pages in memory frames
    int pageFaults = 0; // Counter for page faults
    int next = 0; // Pointer to the next frame to replace

    // Initialize memory array to -1 (indicating empty frame)
    for (int i = 0; i < capacity; i++) {
        memory[i] = -1;
    }

    // Traverse through each page in the reference string
    for (int i = 0; i < n; i++) {
        // Check if the page is already present in memory
        if (!isPresent(memory, capacity, pages[i])) {
            // If memory is not full, place the page in the next available frame
            if (next < capacity) {
                memory[next] = pages[i];
                next++;
            }
```

```
            // If memory is full, replace the oldest page (FIFO principle)
            else {
                memory[next % capacity] = pages[i];
                next++;
            }
            pageFaults++; // Increment page fault count
        }
    }

    return pageFaults; // Return total number of page faults
}

// Optimal Page Replacement Algorithm
int optimal(int pages[], int n, int capacity) {
    int memory[capacity];
    int pageFaults = 0;
    int count = 0;

    for (int i = 0; i < n; i++) {
        if (!isPresent(memory, capacity, pages[i])) {
            if (count < capacity) {
                memory[count++] = pages[i];
            } else {
                int farthest = i + 1, index = -1;
                for (int j = 0; j < capacity; j++) {
                    int k;
                    for (k = i + 1; k < n; k++) {
```

```
                    if (memory[j] == pages[k]) {
                        if (k > farthest) {
                            farthest = k;
                            index = j;
                        }
                        break;
                    }
                }
                if (k == n) {
                    index = j;
                    break;
                }
            }
            memory[index] = pages[i];
        }
        pageFaults++;
    }
}

    return pageFaults;
}

// LRU Page Replacement Algorithm
int lru(int pages[], int n, int capacity) {
    int memory[capacity];
    int time[capacity];
    int pageFaults = 0;
```

```
    int count = 0;
    int timer = 0;

    for (int i = 0; i < n; i++) {
        if (!isPresent(memory, capacity, pages[i])) {
            if (count < capacity) {
                memory[count] = pages[i];
                time[count] = timer++;
                count++;
            } else {
                int lruIndex = findLRU(time, capacity);
                memory[lruIndex] = pages[i];
                time[lruIndex] = timer++;
            }
            pageFaults++;
        } else {
            for (int j = 0; j < capacity; j++) {
                if (memory[j] == pages[i]) {
                    time[j] = timer++;
                }
            }
        }
    }

    return pageFaults;
}
```

```c
int main() {
    int pages[] = {7,0,1,2,0,3,0,4,2,3,0,3,2,3 };
    int n = sizeof(pages) / sizeof(pages[0]);
    int capacity = 4;

    int fifo_faults = fifo(pages, n, capacity);
    int optimal_faults = optimal(pages, n, capacity);
    int lru_faults = lru(pages, n, capacity);

    printf("FIFO Page Faults: %d\n", fifo_faults);
    printf("Optimal Page Faults: %d\n", optimal_faults);
    printf("LRU Page Faults: %d\n", lru_faults);
}
```

Result:

```
FIFO Page Faults: 7
Optimal Page Faults: 6
LRU Page Faults: 6


=== Code Execution Successful ===
```