VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS

(23CS4PCOPS)

Submitted by

Pooja Rajshekhar Arabi(1BM23CS413)

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 **Pooja Rajshekhar Arabi(1BM23CS413),** 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

Lab1)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)

1 a)Code for the FCFS

```
#include <stdio.h>

#define MAX 10

void fcfs(int n, int at[], int bt[]) {
   int ct[MAX];
   int tat[MAX];
   int wt[MAX];
   int total_wt = 0;
   int total_tat = 0;
   int current_time = 0;
   for (int i = 0; i < n; i++) {
      ct[i] = -1;}
   for (int i = 0; i < n; i++) {
      if (current_time < at[i]) {
         current_time = at[i];}
      ct[i] = current_time + bt[i];
      current_time = ct[i];
   }
   for (int i = 0; i < n; i++) {
      tat[i] = ct[i] - at[i];
      total_tat += tat[i];}
   for (int i = 0; i < n; i++) {
      wt[i] = tat[i] - bt[i];
      total_wt += wt[i];
   }
   printf("\nProcess\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting 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", i+1, at[i], bt[i], ct[i], tat[i], wt[i]);
   }
   printf("\nAverage waiting time: %.2f", (float)total_wt / n);
   printf("\nAverage turnaround time: %.2f", (float)total_tat / n);}
int main() {
   int n, i;
   printf("Enter the number of processes: ");
   scanf("%d", &n);

   int at[n], bt[n];
   printf("Enter the arrival time:\n");
   for (i = 0; i < n; i++) {
      scanf("%d", &at[i]);
   }
   printf("Enter the burst time:\n");
   for (i = 0; i < n; i++) {
      scanf("%d", &bt[i]);}
   fcfs(n, at, bt);
   return 0;}
```

output:=

```
Enter the number of processes: 4
Enter the arrival time:
0
1
5
6
Enter the burst time:
2
2
3
4

Process Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time
1       0               2               2               2               0
2       1               2               4               3               1
3       5               3               8               3               0
4       6               4               12              6               2

Average waiting time: 0.75
Average turnaround time: 3.50
Process returned 0 (0x0)   execution time : 18.220 s
Press any key to continue.
```

1 b)code for SJF (pre-emptive)

```
#include <stdio.h>

#define MAX 10

void sjf_non_preemptive(int n, int at[], int bt[]) {
    int ct[MAX];
    int tat[MAX];
    int wt[MAX];
    int rt[MAX]; // Remaining time

    int total_wt = 0;
    int total_tat = 0;
    int completed = 0;
    int current_time = 0;
    int shortest_job = 0;
    int min_bt = 9999; // A very large number initially
    int is_completed[MAX] = {0}; // To keep track of completed processes

    // Initialize remaining times
    for (int i = 0; i < n; i++) {
        rt[i] = bt[i];
    }

    while (completed < n) {
        for (int i = 0; i < n; i++) {
            if (at[i] <= current_time && rt[i] < min_bt && !is_completed[i]) {
```

```
            shortest_job = i;
            min_bt = rt[i];
        }
    }

    rt[shortest_job]--;

    if (rt[shortest_job] == 0) {
        completed++;
        min_bt = 9999;
        is_completed[shortest_job] = 1;

        ct[shortest_job] = current_time + 1;

        tat[shortest_job] = ct[shortest_job] - at[shortest_job];
        total_tat += tat[shortest_job];

        wt[shortest_job] = tat[shortest_job] - bt[shortest_job];
        if (wt[shortest_job] < 0) wt[shortest_job] = 0;
        total_wt += wt[shortest_job];
    }

    current_time++;
  }

  printf("\nProcess\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting 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", i + 1, at[i], bt[i], ct[i], tat[i], wt[i]);
  }

  printf("\nAverage waiting time: %.2f", (float)total_wt / n);
  printf("\nAverage turnaround time: %.2f", (float)total_tat / n);
}

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

  int at[n], bt[n];

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

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

  sjf_non_preemptive(n, at, bt);

  return 0;
}
```

Output:=

```
Enter the number of processes: 4
Enter the arrival time:
0
0
0
0
Enter the burst time:
6
8
7
3

Process Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time
1       0               6               9               9               3
2       0               8               24              24              16
3       0               7               16              16              9
4       0               3               3               3               0

Average waiting time: 7.00
Average turnaround time: 13.00
Process returned 0 (0x0)   execution time : 19.844 s
Press any key to continue.
```

1 c)Code for the SJF (Non-preemptive)

```
#define MAX 10
void sjf_non_preemptive(int n, int at[], int bt[]) {
    int ct[MAX];
    int tat[MAX];
    int wt[MAX];
    int rt[MAX];
    int total_wt = 0;
    int total_tat = 0;
    int completed = 0;
    int current_time = 0;
    int shortest_job = 0;
    int min_bt = 9999;
    int is_completed[MAX] = {0};

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

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

```
            if (at[i] <= current_time && rt[i] < min_bt && !is_completed[i]) {
                shortest_job = i;
                min_bt = rt[i];
            }
        }

        rt[shortest_job]--;

        if (rt[shortest_job] == 0) {
            completed++;
            min_bt = 9999;
            is_completed[shortest_job] = 1;

            ct[shortest_job] = current_time + 1;

            tat[shortest_job] = ct[shortest_job] - at[shortest_job];

            total_tat += tat[shortest_job];
            wt[shortest_job] = tat[shortest_job] - bt[shortest_job];

            if (wt[shortest_job] < 0) wt[shortest_job] = 0;
            total_wt += wt[shortest_job];
        }

        current_time++;
    }

    printf("\nProcess\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting 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", i + 1, at[i], bt[i], ct[i], tat[i], wt[i]);
    }

    printf("\nAverage waiting time: %.2f", (float)total_wt / n);
    printf("\nAverage turnaround time: %.2f", (float)total_tat / n);
}

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

    int at[n], bt[n];

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

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

    sjf_non_preemptive(n, at, bt);

    return 0;
}
```

Output:=

```
Enter the number of processes: 4
Enter the arrival time:
0
0
0
0
Enter the burst time:
6
8
7
3

Process Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time
1       0               6               9               9               3
2       0               8               24              24              16
3       0               7               16              16              9
4       0               3               3               3               0

Average waiting time: 7.00
Average turnaround time: 13.00
Process returned 0 (0x0)   execution time : 19.844 s
Press any key to continue.
```

Lab 2 - Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time.

→ Priority (pre-emptive & Non-pre-emptive)

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

2 a)Code for the Priority (pre-emptive)

```
#include<stdio.h>
void sort (int proc_id[], int p[], int at[], int bt[], int b[], int n){
  int min = p[0], temp = 0;
  for (int i = 0; i < n; i++){
                min = p[i];
                for (int j = i; j < n; j++){
                                if (p[j] < min){
                                                temp = at[i];
                                                at[i] = at[j];
                                                at[j] = temp;
                                                temp = bt[j];
                                                bt[j] = bt[i];
                                                bt[i] = temp;
                                                temp = b[j];
                                                b[j] = b[i];
                                                b[i] = temp;
                                                temp = p[j];
                                                p[j] = p[i];
                                                p[i] = temp;
                                                temp = proc_id[i];
                                                proc_id[i] = proc_id[j];
                                                proc_id[j] = temp;}}}}
void main (){
  int n, c = 0;
  printf ("Enter number of processes: ");
  scanf ("%d", &n);

  int proc_id[n], at[n], bt[n], ct[n], tat[n], wt[n], m[n], b[n], rt[n], p[n];
  double avg_tat = 0.0, ttat = 0.0, avg_wt = 0.0, twt = 0.0;

  for (int i = 0; i < n; i++){
                proc_id[i] = i + 1;
                m[i] = 0;
               }
  printf ("Enter priorities:\n");
  for (int i = 0; i < n; i++)
               scanf ("%d", &p[i]);

  printf ("Enter arrival times:\n");
  for (int i = 0; i < n; i++)
               scanf ("%d", &at[i]);

  printf ("Enter burst times:\n");

  for (int i = 0; i < n; i++){
                scanf ("%d", &bt[i]);
                b[i] = bt[i];
                m[i] = -1;
```

```
            rt[i] = -1;
          }
sort(proc_id, p, at, bt, b, n);
  int count = 0, pro = 0, priority = p[0];
  int x = 0;
  c = 0;
  while (count < n){
          for (int i = 0; i < n; i++){
                  if (at[i] <= c && p[i] >= priority && b[i] > 0 && m[i] != 1){
                          x = i;
                          priority = p[i];
                         }

}
          if (b[x] > 0){
                  if (rt[x] == -1)
                          rt[x] = c - at[x];
                  b[x]--;
                  c++;
}
          if (b[x] == 0){
                  count++;
                  ct[x] = c;
                  m[x] = 1;
                  while (x >= 1 && b[x] == 0)
                          priority = p[--x];
}
          if (count == n)
                 break;
}
  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 ("Priority scheduling(Pre-Emptive):\n");
  printf ("PID\tPrior\tAT\tBT\tCT\tTAT\tWT\tRT\n");

  for (int i = 0; i < n; i++)
          printf ("P%d\t %d\t\t%d\t%d\t%d\t%d\t%d\t%d\n", proc_id[i], p[i], at[i],
                          bt[i], ct[i], tat[i], wt[i], rt[i]);

  for (int i = 0; i < n; i++){
          ttat += tat[i];
          twt += wt[i];
          }

  avg_tat = ttat / (double) n;
  avg_wt = twt / (double) n;

  printf ("\nAverage turnaround time:%lfms\n", avg_tat);
  printf ("\nAverage waiting time:%lfms\n", avg_wt);
}
```

```
Enter number of processes: 4
Enter priorities:
10
20
30
40
Enter arrival times:
0
1
2
4
Enter burst times:
5
4
2
1
Priority scheduling(Pre-Emptive):
PID     Prior   AT      BT      CT      TAT     WT      RT
P1       10             0       5       12      12      7       0
P2       20             1       4       8       7       3       0
P3       30             2       2       4       2       0       0
P4       40             4       1       5       1       0       0

Average turnaround time:5.500000ms

Average waiting time:2.500000ms

Process returned 33 (0x21)   execution time : 16.344 s
Press any key to continue.
```

2 b) code for the Priority (Non-pre-emptive)

```
#include<stdio.h>
void sort (int proc_id[], int p[], int at[], int bt[], int b[], int n){
  int min = p[0], temp = 0;
  for (int i = 0; i < n; i++){
                min = p[i];
                for (int j = i; j < n; j++){
                        if (p[j] < min){
                                temp = at[i];
                                at[i] = at[j];
                                at[j] = temp;
                                temp = bt[j];
                                bt[j] = bt[i];
                                bt[i] = temp;
                                temp = b[j];
                                b[j] = b[i];
                                b[i] = temp;
                                temp = p[j];
                                p[j] = p[i];
                                p[i] = temp;
                                temp = proc_id[i];
                                proc_id[i] = proc_id[j];
                                proc_id[j] = temp;
                               }
                        }
           }
```

```
}
void main (){
  int n, c = 0;
  printf ("Enter number of processes: ");
  scanf ("%d", &n);
  int proc_id[n], at[n], bt[n], ct[n], tat[n], wt[n], m[n], b[n], rt[n], p[n];
  double avg_tat = 0.0, ttat = 0.0, avg_wt = 0.0, twt = 0.0;
  for (int i = 0; i < n; i++){
                proc_id[i] = i + 1;
                m[i] = 0;
               }

  printf ("Enter priorities:\n");
  for (int i = 0; i < n; i++)
               scanf ("%d", &p[i]);

  printf ("Enter arrival times:\n");
  for (int i = 0; i < n; i++)
               scanf ("%d", &at[i]);

  printf ("Enter burst times:\n");

  for (int i = 0; i < n; i++)
               {
                scanf ("%d", &bt[i]);
                b[i] = bt[i];
                m[i] = -1;
                rt[i] = -1;
               }

               sort (proc_id, p, at, bt, b, n);
  int count = 0, pro = 0, priority = p[0];
  int x = 0;
  c = 0;
  while (count < n){
                for (int i = 0; i < n; i++){
                        if (at[i] <= c && p[i] >= priority && b[i] > 0 && m[i] != 1){
                                x = i;
                                priority = p[i];
                               }
                       }

                if (b[x] > 0){
                        if (rt[x] == -1)
                                rt[x] = c - at[x];
                        b[x]--;
                        c++;
                       }

                if (b[x] == 0){
                        count++;
                        ct[x] = c;
                        m[x] = 1;
                        while (x >= 1 && b[x] == 0)
                                priority = p[--x];
                       }
```

```
            if (count == n)
                    break;
            }

  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 ("Priority scheduling(Pre-Emptive):\n");
  printf ("PID\tPrior\tAT\tBT\tCT\tTAT\tWT\tRT\n");

  for (int i = 0; i < n; i++)
            printf ("P%d\t %d\t\t%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], rt[i]);


  for (int i = 0; i < n; i++){
             ttat += tat[i];
             twt += wt[i];
            }

  avg_tat = ttat / (double) n;
  avg_wt = twt / (double) n;

  printf ("\nAverage turnaround time:%lfms\n", avg_tat);
  printf ("\nAverage waiting time:%lfms\n", avg_wt);
}
```

```
Enter number of processes: 4
Enter priorities:
10
20
30
40
Enter arrival times:
0
1
2
4
Enter burst times:
5
4
2
1
Priority scheduling(Pre-Emptive):
PID     Prior   AT      BT      CT      TAT     WT      RT
P1       0              5       12      12      7       0       10
P2       1              4       8       7       3       0       10
P3       2              2       4       2       0       0       10
P4       4              1       5       1       0       0       10

Average turnaround time:5.500000ms

Average waiting time:2.500000ms

Process returned 33 (0x21)   execution time : 20.313 s
Press any key to continue.
```

2 c)Code for the Round Robin (Experiment with different quantum sizes for RR algorithm)

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

void round_robin(int n, int bt[], int quantum) {

   int wt[MAX] = {0};
   int tat[MAX] = {0};
   int remaining_bt[MAX];

   int total_wt = 0, total_tat = 0;
   int time = 0;

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

   while (1) {
      int done = 1;

      for (int i = 0; i < n; i++) {
         if (remaining_bt[i] > 0) {
            done = 0;

            if (remaining_bt[i] > quantum) {
               time += quantum;
               remaining_bt[i] -= quantum;

            }
else {
               time += remaining_bt[i];
               wt[i] = time - bt[i];
               remaining_bt[i] = 0;  }
         }

      }

      if (done == 1) break;
   }

   for (int i = 0; i < n; i++) {
      tat[i] = bt[i] + wt[i];
      total_wt += wt[i];
      total_tat += tat[i];
   }

   printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");

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

   printf("\nAverage waiting time: %.2f", (float)total_wt / n);
   printf("\nAverage turnaround time: %.2f", (float)total_tat / n);
}
```

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

    int bt[MAX];

    printf("Enter Burst Time for each process:\n");

    for (int i = 0; i < n; i++) {
        printf("Process %d: ", i + 1);

        scanf("%d", &bt[i]);

    }
    printf("Enter the size of time slice (quantum): ");
    scanf("%d", &quantum);

    round_robin(n, bt, quantum);
    return 0;
}
```

```
Enter the number of processes: 3
Enter Burst Time for each process:
Process 1: 24
Process 2: 3
Process 3:
3
Enter the size of time slice (quantum): 3

Process Burst Time      Waiting Time    Turnaround Time
1       24              6               30
2       3               3               6
3       3               6               9

Average waiting time: 5.00
Average turnaround time: 15.00
Process returned 0 (0x0)   execution time : 136.048 s
Press any key to continue.
```

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

3a)Code for the multi-level queue

```
#include <stdio.h>
void findWaitingTime(int processes[], int n, int bt[], int at[], int wt[]) {
   wt[0] = 0;
   for (int i = 1; i < n; i++) {
      wt[i] = bt[i-1] + wt[i-1] - at[i-1];
      if (wt[i] < 0)
         wt[i] = 0;
   }
}void findTurnaroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
   for (int i = 0; i < n; i++) {
      tat[i] = bt[i] + wt[i];
   }
}
void roundRobin(int processes[], int n, int bt[], int at[], int quantum) {
   int wt[n], tat[n], ct[n], total_wt = 0, total_tat = 0;
   int remaining_bt[n];
   int completed = 0;
   int time = 0;
   for (int i = 0; i < n; i++) {
      remaining_bt[i] = bt[i];
   }
   while (completed < n) {
      for (int i = 0; i < n; i++) {
         if (remaining_bt[i] > 0 && at[i] <= time) {
            if (remaining_bt[i] <= quantum) {
               time += remaining_bt[i];
               remaining_bt[i] = 0;
               ct[i] = time;
               completed++;
            } else {
               time += quantum;
               remaining_bt[i] -= quantum;
            }
         }
      }
   }
   findWaitingTime(processes, n, bt, at, wt);
   findTurnaroundTime(processes, n, bt, wt, tat);
   printf("Processes  Burst Time  Arrival Time  Waiting Time  Turnaround Time  Completion Time\n");
   for (int i = 0; i < n; i++) {
      printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i], bt[i], at[i], wt[i], tat[i], ct[i]);
      total_wt += wt[i];
      total_tat += tat[i];
   }
```

```
    printf("Average Waiting Time (Round Robin) = %f\n", (float)total_wt / n);
    printf("Average Turnaround Time (Round Robin) = %f\n", (float)total_tat / n);
}
void fcfs(int processes[], int n, int bt[], int at[]) {
    int wt[n], tat[n], ct[n], total_wt = 0, total_tat = 0;
    findWaitingTime(processes, n, bt, at, wt);
    findTurnaroundTime(processes, n, bt, wt, tat);
    printf("Processes  Burst Time  Arrival Time  Waiting Time  Turnaround Time  Completion Time\n");
    for (int i = 0; i < n; i++) {
        ct[i] = at[i] + bt[i];
        printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i], bt[i], at[i], wt[i], tat[i], ct[i]);
        total_wt += wt[i];
        total_tat += tat[i];
    }
    printf("Average Waiting Time (FCFS) = %f\n", (float)total_wt / n);
    printf("Average Turnaround Time (FCFS) = %f\n", (float)total_tat / n);
}
int main() {
    int processes[] = {1, 2, 3, 4, 5};
    int n = sizeof(processes) / sizeof(processes[0]);
    int bt[] = {10, 5, 8, 12, 15};
    int at[] = {0, 1, 2, 3, 4};
    int quantum = 2;
    roundRobin(processes, n, bt, at, quantum);
    fcfs(processes, n, bt, at);
    return 0;
}
```

```
Processes  Burst Time  Arrival Time  Waiting Time  Turnaround Time  Completion Time
P1              10              0              0               10              39
P2              5               1              10              15              23
P3              8               2              14              22              33
P4              12              3              20              32              45
P5              15              4              29              44              50
Average Waiting Time (Round Robin) = 14.600000
Average Turnaround Time (Round Robin) = 24.600000
Processes  Burst Time  Arrival Time  Waiting Time  Turnaround Time  Completion Time
P1              10              0              0               10              10
P2              5               1              10              15              6
P3              8               2              14              22              10
P4              12              3              20              32              15
P5              15              4              29              44              19
Average Waiting Time (FCFS) = 14.600000
Average Turnaround Time (FCFS) = 24.600000

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

Lab4-Write a C program to simulate Real-Time CPU Scheduling algorithms:

a) Rate- Monotonic

b) Earliest-deadline First

c) Proportional scheduling

4 a)code for the 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;}
Void 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);
//LCM
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]);

//feasibility
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);

//RMS
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++}}
```

OUTPUT:=

```
Enter the number of processes:3
Enter the CPU burst times:
3
2
2
Enter the time periods:
20
5
10
LCM=20

Rate Monotone Scheduling:
PID      Burst  Period
2               2               5
3               2               10
1               3               20

0.750000 <= 0.779763 =>true
Scheduling occurs for 20 ms

0ms onwards: Process 2 running
2ms onwards: Process 3 running
4ms onwards: Process 1 running
5ms onwards: Process 2 running
7ms onwards: Process 1 running
8ms onwards: CPU is idle
10ms onwards: Process 2 running

Process returned 20 (0x14)   execution time : 16.484 s
Press any key to continue.
```

4 b)code for the 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;}
void 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++;}}
```

Output:=

Output:=

```
Enter the number of processes:3
Enter the CPU burst times:
0
1
2
Enter the deadlines:
8
5
4
Enter the time periods:
3
4
6

Earliest Deadline Scheduling:
PID      Burst  Deadline        Period
3               2               4               6
2               1               5               4
1               0               8               3
Scheduling occurs for 12 ms

0ms : Task 3 is running.
1ms : Task 3 is running.
2ms : Task 2 is running.
3ms: CPU is idle.
4ms : Task 2 is running.
5ms: CPU is idle.
6ms : Task 3 is running.
7ms : Task 3 is running.
8ms: CPU is idle.
9ms: CPU is idle.
10ms: CPU is idle.
11ms: CPU is idle.

Process returned 12 (0xC)   execution time : 31.625 s
Press any key to continue.
```

4 c)code for the Proportional scheduling

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define MAX_TASKS 10

#define MAX_TICKETS 100

#define TIME_UNIT_DURATION_MS 100


struct Task {
   int tid;
   int tickets;
};

void schedule(struct Task tasks[], int num_tasks, int *time_span_ms) {
   int total_tickets = 0;

   for (int i = 0; i < num_tasks; i++) {
      total_tickets += tasks[i].tickets;
   }
   srand(time(NULL));
   int current_time = 0;

   int completed_tasks = 0;

     printf("Process Scheduling:\n");

   while (completed_tasks < num_tasks) {
      int winning_ticket = rand() % total_tickets;
      int cumulative_tickets = 0;

      for (int i = 0; i < num_tasks; i++) {
         cumulative_tickets += tasks[i].tickets;

         if (winning_ticket < cumulative_tickets) {
            printf("Time %d-%d: Task %d is running\n", current_time, current_time + 1, tasks[i].tid);
            current_time++;
            break;} }
      completed_tasks++;}

   *time_span_ms = current_time * TIME_UNIT_DURATION_MS;
}

int main() {

   struct Task tasks[MAX_TASKS];
   int num_tasks;

   int time_span_ms;

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

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

    if (num_tasks <= 0 || num_tasks > MAX_TASKS) {
        printf("Invalid number of tasks. Please enter a number between 1 and %d.\n", MAX_TASKS);
        return 1;
}

    printf("Enter number of tickets for each task:\n");

    for (int i = 0; i < num_tasks; i++) {
        tasks[i].tid = i + 1;

        printf("Task %d tickets: ", tasks[i].tid);
        scanf("%d", &tasks[i].tickets);
}

    printf("\nRunning tasks:\n");

    schedule(tasks, num_tasks, &time_span_ms);

    printf("\nTime span of the Gantt chart: %d milliseconds\n", time_span_ms);
    return 0;
}
```

Output:=

```
Enter the number of tasks: 3
Enter number of tickets for each task:
Task 1 tickets: 10
Task 2 tickets: 20
Task 3 tickets: 30

Running tasks:
Process Scheduling:
Time 0-1: Task 1 is running
Time 1-2: Task 3 is running
Time 2-3: Task 1 is running

Time span of the Gantt chart: 300 milliseconds

Process returned 0 (0x0)   execution time : 19.017 s
Press any key to continue.
```

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

```
#include<stdio.h>
#include<stdlib.h>
int mutex = 1, full = 0, empty = 5, x = 0;
int main()
{
int n;
void producer();
void consumer();
int wait(int);
int signal(in t);
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);
 }
```

Output:=

```
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:1

Producer produces the item 3
Enter your choice:1

Producer produces the item 4
Enter your choice:1

Producer produces the item 5
Enter your choice:1
Buffer is full!!
Enter your choice:2

Consumer consumes item 5
Enter your choice:2

Consumer consumes item 4
Enter your choice:2

Consumer consumes item 3
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:3

Process returned 0 (0x0)   execution time : 125.468 s
Press any key to continue.
```

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

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

#define MAX_PHILOSOPHERS 10

typedef enum { THINKING, HUNGRY, EATING } state_t;

state_t states[MAX_PHILOSOPHERS];
int num_philosophers;
int num_hungry;
int hungry_philosophers[MAX_PHILOSOPHERS];
int forks[MAX_PHILOSOPHERS];

void print_state() {
   printf("\n");
   for (int i = 0; i < num_philosophers; ++i) {
      if (states[i] == THINKING) printf("P %d is thinking\n", i + 1);
      else if (states[i] == HUNGRY) printf("P %d is waiting\n", i + 1);
      else if (states[i] == EATING) printf("P %d is eating\n", i + 1);
   }
}

int can_eat(int philosopher_id) {
   int left_fork = philosopher_id;
   int right_fork = (philosopher_id + 1) % num_philosophers;

   if (forks[left_fork] == 0 && forks[right_fork] == 0) {
      forks[left_fork] = forks[right_fork] = 1;
      return 1; // Philosopher can eat
   }
   return 0; // Philosopher cannot eat
}

void simulate(int allow_two) {
   int eating_count = 0;
   for (int i = 0; i < num_hungry; ++i) {
      int philosopher_id = hungry_philosophers[i];

      if (states[philosopher_id] == HUNGRY) {
         if (can_eat(philosopher_id)) {
            states[philosopher_id] = EATING;
            eating_count++;
            printf("P %d is granted to eat\n", philosopher_id + 1);
            if (!allow_two && eating_count == 1) break;
            if (allow_two && eating_count == 2) break;
         }
      }
   }


   for (int i = 0; i < num_hungry; ++i) {
      int philosopher_id = hungry_philosophers[i];
      if (states[philosopher_id] == EATING) {
         int left_fork = philosopher_id;
         int right_fork = (philosopher_id + 1) % num_philosophers;
```

```
            forks[left_fork] = forks[right_fork] = 0;
            states[philosopher_id] = THINKING;
        }
    }
}

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

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

    printf("How many are hungry: ");
    scanf("%d", &num_hungry);

    for (int i = 0; i < num_hungry; ++i) {
        printf("Enter philosopher %d position: ", i + 1);
        int position;
        scanf("%d", &position);
        hungry_philosophers[i] = position - 1;
        states[hungry_philosophers[i]] = HUNGRY;
    }
    for (int i = 0; i < num_philosophers; ++i) {
        forks[i] = 0;
    }

    int choice;
    do {
        print_state();
        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:
                simulate(0);
                break;
            case 2:
                simulate(1);
                break;
            case 3:
                printf("Exiting.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
                break;
        }

    } while (choice != 3);

    return 0;
}
```

OUTPUT:=

```
P 1 is granted to eat

P 1 is thinking
P 2 is thinking
P 3 is waiting
P 4 is thinking
P 5 is waiting

1. One can eat at a time
2. Two can eat at a time
3. Exit
Enter your choice: 2
P 3 is granted to eat
P 5 is granted to eat

P 1 is thinking
P 2 is thinking
P 3 is thinking
P 4 is thinking
P 5 is thinking

1. One can eat at a time
2. Two can eat at a time
3. Exit
Enter your choice: 3
Exiting.

Process returned 0 (0x0)   execution time : 24.047 s
Press any key to continue.
```

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

```
#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;
}
```

OUTPUT:=

```
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 -- 3 3 2
P1 is visited (5 3 2 )
P3 is visited (7 4 3 )
P4 is visited (7 4 5 )
P2 is visited (10 4 7 )
System is not in safe state

Process Allocation      Max     Need
P0      1 0 1   7 5 3   6 5 2
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

Process returned 0 (0x0)   execution time : 65.848 s
Press any key to continue.
```

Lab 8- Write a C program to simulate deadlock detection

```
#include<stdio.h>

void main()
{

   int n,m,i,j;

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

   int request[n][m],all[n][m],ava[m],flag=1,finish[n],dead[n],c=0;

   printf("Enter the allocated number of each type of resource needed by each process:\n");

   for(i=0;i<n;i++)
   {
      for(j=0;j<m;j++)
      {
         scanf("%d",&all[i][j]);
      }
   }

   printf("Enter the available number of each type of resource:\n");
   for(j=0;j<m;j++)
   {
      scanf("%d",&ava[j]);
   }

   printf("Enter the request number of each type of resource needed by each process:\n");

   for(i=0;i<n;i++)
   {
      for(j=0;j<m;j++)
      {
         scanf("%d",&request[i][j]);
      }
   }

   for(i=0;i<n;i++)
   {
      finish[i]=0;
   }

   while(flag)
   {
      flag=0;

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

         for(j=0;j<m;j++)
         {
            if(finish[i]==0 && request[i][j]<=ava[j])
            {
```

```
            c++;

            if(c==m)
            {
               for(j=0;j<m;j++)
               {
                  ava[j]-=request[i][j];
                  ava[j]+=all[i][j];
                  finish[i]=1;
                  flag=1;
               }
               if(finish[i]==1)
               {
                  i=n; }}}} }}
   j=0;
   flag=0;
   for(i=0;i<n;i++)
   {
      if(finish[i]==0)
      {
         dead[j]=i;
         j++;
         flag=1;
      }
   }
   if(flag==1)
   {
      printf("Deadlock has occured:\n");
      printf("The deadlock processes are:\n");

      for(i=0;i<j;i++){
         printf("P%d ",dead[i]); }}
   else
   printf("No deadlock has occured!\n");}
```

OUTPUT:=

```
Enter the number of processes and number of types of resources:
4 3
Enter the allocated number of each type of resource needed by each process:
1 0 2
2 1 1
1 0 3
1 2 2
Enter the available number of each type of resource:
0 0 0
Enter the request number of each type of resource needed by each process:
0 0 1
1 0 2
0 0 0
3 3 0
Deadlock has occured:
The deadlock processes are:
P3
Process returned 1 (0x1)   execution time : 40.250 s
Press any key to continue.
```

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

a) Worst-fit

b) Best-fit

c) First-fit

```
#include <stdio.h>
struct Block {
   int block_no;
   int block_size;
   int is_free;
};
struct File {
   int file_no;
   int file_size;
};
void firstFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
   printf("Memory Management Scheme - First Fit\n");
   printf("File_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment\n");
   for (int i = 0; i < n_files; i++) {
      for (int j = 0; j < n_blocks; j++) {
         if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
            blocks[j].is_free = 0;
            printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", files[i].file_no, files[i].file_size, blocks[j].block_no,
blocks[j].block_size, blocks[j].block_size - files[i].file_size);
            break;          }       }    }}
void worstFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
   printf("Memory Management Scheme - Worst Fit\n");
   printf("File_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment\n");

   for (int i = 0; i < n_files; i++) {
      int worst_fit_block = -1;
      int max_fragment = -1;
      for (int j = 0; j < n_blocks; j++) {
         if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
            int fragment = blocks[j].block_size - files[i].file_size;
            if (fragment > max_fragment) {
               max_fragment = fragment;
               worst_fit_block = j;             }          }       }
      if (worst_fit_block!= -1) {
         blocks[worst_fit_block].is_free = 0;
         printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", files[i].file_no, files[i].file_size,
blocks[worst_fit_block].block_no, blocks[worst_fit_block].block_size, max_fragment);       }    }}
void bestFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
   printf("Memory Management Scheme - Best Fit\n");
   printf("File_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment\n");

   for (int i = 0; i < n_files; i++) {
      int best_fit_block = -1;
      int min_fragment = 10000;
      for (int j = 0; j < n_blocks; j++) {
         if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
            int fragment = blocks[j].block_size - files[i].file_size;
            if (fragment < min_fragment) {
               min_fragment = fragment;
               best_fit_block = j;} }}
      if (best_fit_block!= -1) {
```

```
            blocks[best_fit_block].is_free = 0;
            printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", files[i].file_no, files[i].file_size,
blocks[best_fit_block].block_no, blocks[best_fit_block].block_size, min_fragment); }}}
int main() {
    int n_blocks, n_files;
    printf("Enter the number of blocks: ");
    scanf("%d", &n_blocks);
    printf("Enter the number of files: ");
    scanf("%d", &n_files);
    struct Block blocks[n_blocks];
    for (int i = 0; i < n_blocks; i++) {
        blocks[i].block_no = i + 1;
        printf("Enter the size of block %d: ", i + 1);
        scanf("%d", &blocks[i].block_size);
        blocks[i].is_free = 1;    }
    struct File files[n_files];
    for (int i = 0; i < n_files; i++) {
        files[i].file_no = i + 1;
        printf("Enter the size of file %d: ", i + 1);
        scanf("%d", &files[i].file_size);    }
    firstFit(blocks, n_blocks, files, n_files);
    printf("\n");
    for (int i = 0; i < n_blocks; i++) {
        blocks[i].is_free = 1;    }
    worstFit(blocks, n_blocks, files, n_files);
    printf("\n");
    for (int i = 0; i < n_blocks; i++) {
        blocks[i].is_free = 1;    }
    bestFit(blocks, n_blocks, files, n_files);
    return 0;}
```

OUTPUT:=

```
Enter the number of blocks: 3
Enter the number of files: 2
Enter the size of block 1: 5
Enter the size of block 2: 2
Enter the size of block 3: 7
Enter the size of file 1: 1
Enter the size of file 2: 4
Memory Management Scheme - First Fit
File_no:        File_size:      Block_no:       Block_size:     Fragment
1               1               1               5               4
2               4               3               7               3

Memory Management Scheme - Worst Fit
File_no:        File_size:      Block_no:       Block_size:     Fragment
1               1               3               7               6
2               4               1               5               1

Memory Management Scheme - Best Fit
File_no:        File_size:      Block_no:       Block_size:     Fragment
1               1               2               2               1
2               4               1               5               1

Process returned 0 (0x0)   execution time : 45.141 s
Press any key to continue.
```

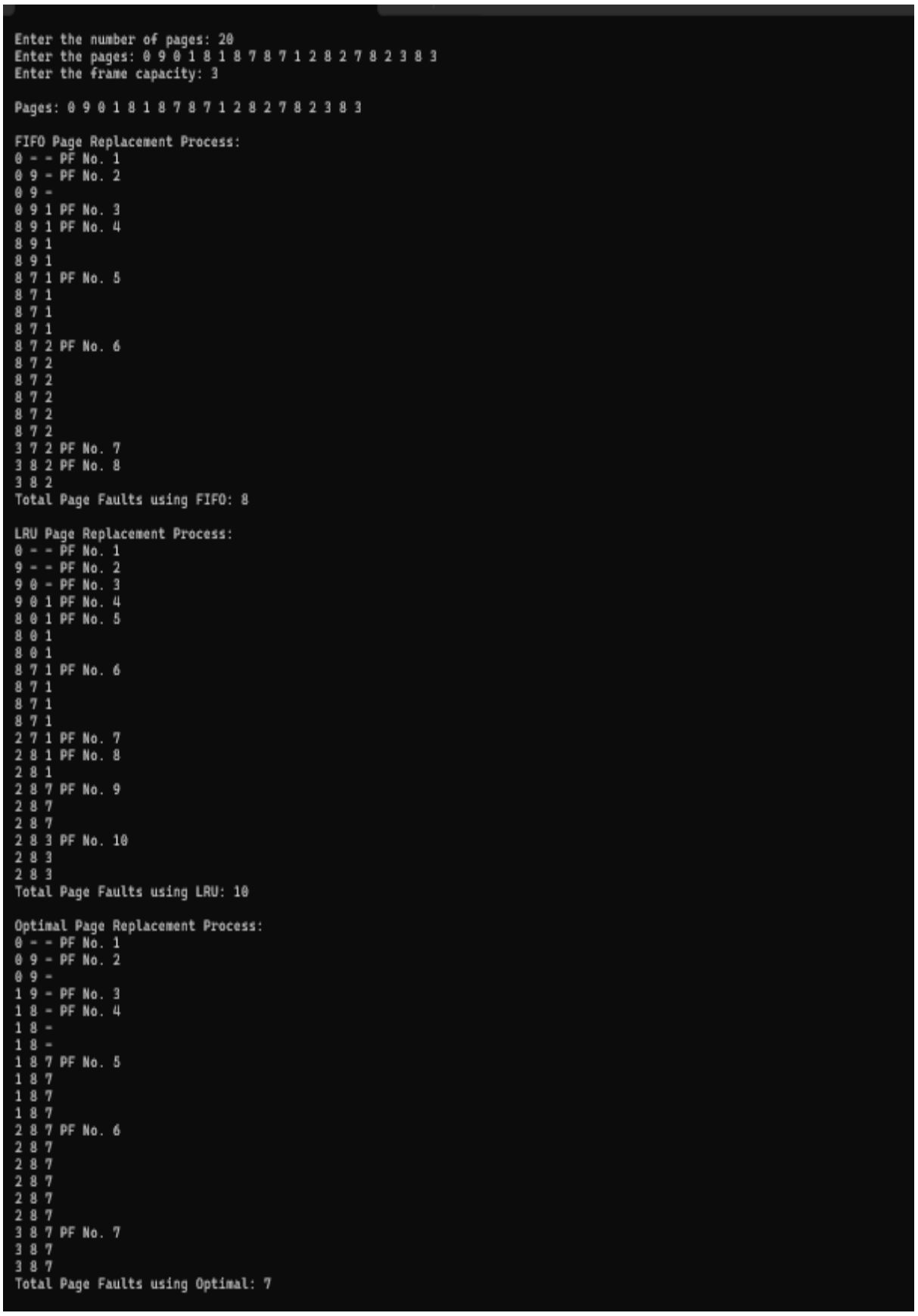
Lab 10) Write a C program to simulate page replacement algorithms
a) FIFO
b) LRU
c) Optimal

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

void print_frames(int frame[], int capacity, int page_faults) {
   for (int i = 0; i < capacity; i++) {
      if (frame[i] == -1)
         printf("- ");
      else
         printf("%d ", frame[i]);
   }
   if (page_faults > 0)
      printf("PF No. %d", page_faults);
   printf("\n");
}
void fifo(int pages[], int n, int capacity) {
   int frame[capacity], index = 0, page_faults = 0;
   for (int i = 0; i < capacity; i++)
      frame[i] = -1;

   printf("FIFO Page Replacement Process:\n");
   for (int i = 0; i < n; i++) {
      int found = 0;
      for (int j = 0; j < capacity; j++) {
         if (frame[j] == pages[i]) {
            found = 1;
            break;
         }
      }
      if (!found) {
         frame[index] = pages[i];
         index = (index + 1) % capacity;
         page_faults++;
      }
      print_frames(frame, capacity, found ? 0 : page_faults);
   }
   printf("Total Page Faults using FIFO: %d\n\n", page_faults);
}
void lru(int pages[], int n, int capacity) {
   int frame[capacity], counter[capacity], time = 0, page_faults = 0;
   for (int i = 0; i < capacity; i++) {
      frame[i] = -1;
      counter[i] = 0; }
   printf("LRU Page Replacement Process:\n");
   for (int i = 0; i < n; i++) {
      int found = 0;
```

```
        for (int j = 0; j < capacity; j++) {
            if (frame[j] == pages[i]) {
                found = 1;
                counter[j] = time++;
                break; } }
        if (!found) {
            int min = INT_MAX, min_index = -1;
            for (int j = 0; j < capacity; j++) {
                if (counter[j] < min) {
                    min = counter[j];
                    min_index = j; } }
            frame[min_index] = pages[i];
            counter[min_index] = time++;
            page_faults++; }
        print_frames(frame, capacity, found ? 0 : page_faults); }
    printf("Total Page Faults using LRU: %d\n\n", page_faults); }
void optimal(int pages[], int n, int capacity) {
    int frame[capacity], page_faults = 0;
    for (int i = 0; i < capacity; i++)
        frame[i] = -1;
    printf("Optimal Page Replacement Process:\n");
    for (int i = 0; i < n; i++) {
        int found = 0;
        for (int j = 0; j < capacity; j++) {
            if (frame[j] == pages[i]) {
                found = 1;
                break; } }
        if (!found) {
            int farthest = i + 1, index = -1;
            for (int j = 0; j < capacity; j++) {
                int k;
                for (k = i + 1; k < n; k++) {
                    if (frame[j] == pages[k])
                        break; }
                if (k > farthest) {
                    farthest = k;
                    index = j; } }
            if (index == -1) {
                for (int j = 0; j < capacity; j++) {
                    if (frame[j] == -1) {
                        index = j;
                        break; } } }
            frame[index] = pages[i];
            page_faults++; }
        print_frames(frame, capacity, found ? 0 : page_faults); }
    printf("Total Page Faults using Optimal: %d\n\n", page_faults); }
int main() {
    int n, capacity;
    printf("Enter the number of pages: ");
    scanf("%d", &n);
    int *pages = (int*)malloc(n * sizeof(int));
```

```
printf("Enter the pages: ");
for (int i = 0; i < n; i++)
   scanf("%d", &pages[i]);
printf("Enter the frame capacity: ");
scanf("%d", &capacity);
printf("\nPages: ");
for (int i = 0; i < n; i++)
   printf("%d ", pages[i]);
printf("\n\n");
fifo(pages, n, capacity);
lru(pages, n, capacity);
optimal(pages, n, capacity);
free(pages);
return 0; }}
```

OUTPUT



```
Enter the number of pages: 20
Enter the pages: 0 9 0 1 8 1 8 7 8 7 1 2 8 2 7 8 2 3 8 3
Enter the frame capacity: 3

Pages: 0 9 0 1 8 1 8 7 8 7 1 2 8 2 7 8 2 3 8 3

FIFO Page Replacement Process:
0 - - PF No. 1
0 9 - PF No. 2
0 9 -
0 9 1 PF No. 3
8 9 1 PF No. 4
8 9 1
8 9 1
8 7 1 PF No. 5
8 7 1
8 7 1
8 7 1
8 7 2 PF No. 6
8 7 2
8 7 2
8 7 2
8 7 2
8 7 2
3 7 2 PF No. 7
3 8 2 PF No. 8
3 8 2
Total Page Faults using FIFO: 8

LRU Page Replacement Process:
0 - - PF No. 1
9 - - PF No. 2
9 0 - PF No. 3
9 0 1 PF No. 4
8 0 1 PF No. 5
8 0 1
8 0 1
8 7 1 PF No. 6
8 7 1
8 7 1
8 7 1
2 7 1 PF No. 7
2 8 1 PF No. 8
2 8 1
2 8 7 PF No. 9
2 8 7
2 8 7
2 8 3 PF No. 10
2 8 3
2 8 3
Total Page Faults using LRU: 10

Optimal Page Replacement Process:
0 - - PF No. 1
0 9 - PF No. 2
0 9 -
1 9 - PF No. 3
1 8 - PF No. 4
1 8 -
1 8 -
1 8 7 PF No. 5
1 8 7
1 8 7
1 8 7
2 8 7 PF No. 6
2 8 7
2 8 7
2 8 7
2 8 7
2 8 7
3 8 7 PF No. 7
3 8 7
3 8 7
Total Page Faults using Optimal: 7
```