VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"Jnana Sangama", Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS
(23CS4PCOPS)

Submitted by

Likhith M (1BM22CS135)

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 **Likhith M (1BM22CS135),** 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 an **OPERATING SYSTEMS - (23CS4PCOPS)** work prescribed for the said degree.

Rajeshwari Madli
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

LABORATORY 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 (pre-emptive & non-preemptive).

Code: FCFS

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

struct process
{
   char name[5]; // Process name
   int AT;
   int BT;
   int CT;
   int TAT;
   int WT;
};

int main()
{
   int n, temp;
   printf("Enter the number of processes: ");
   scanf("%d", &n);
   struct process p[n];
   printf("Enter the arrival time and burst time for all the processes:\n");
   for (int i = 0; i < n; i++)
   {
      printf("\n\nProcess %d:\n", i + 1);
      sprintf(p[i].name, "p%d", i + 1);
     printf("Arrival time: ");
      scanf("%d", &p[i].AT);
      printf("Burst Time: ");
      scanf("%d", &p[i].BT);
   }

   // Sorting processes based on arrival time (FCFS)
   for (int i = 0; i < n - 1; i++)
   {
      for (int j = 0; j < n - i - 1; j++)
      {
         if (p[j].AT > p[j + 1].AT)
         {
            // Swap
            temp = p[j].AT;
            p[j].AT = p[j + 1].AT;
            p[j + 1].AT = temp;

            temp = p[j].BT;
```

```
            p[j].BT = p[j + 1].BT;
            p[j + 1].BT = temp;

            char temp_name[5];
            strcpy(temp_name, p[j].name);
            strcpy(p[j].name, p[j + 1].name);
            strcpy(p[j + 1].name, temp_name);
          }
        }
    }

    // Calculate completion time, waiting time, and turnaround time
    int current_time = 0;
    for (int i = 0; i < n; i++)
    {
        if (current_time < p[i].AT)
            current_time = p[i].AT;

        p[i].CT = current_time + p[i].BT;
        p[i].TAT = p[i].CT - p[i].AT;
        p[i].WT = p[i].TAT - p[i].BT;

        current_time = p[i].CT;
    }


    // Calculate average waiting time and average turnaround time
    int total_wt=0, total_tat=0;

    for(int i=0;i<n;i++)
    {
        total_wt+=p[i].WT;
        total_tat+=p[i].TAT;
    }

    float s=(float)total_wt / (float)n;
    float t=(float)total_tat / (float)n;


    // Print the details of each process
    printf("\n\nProcess\tName\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting
Time\n");
    for (int i = 0; i < n; i++)
    {
        printf("%d\t%s\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i + 1, p[i].name, p[i].AT, p[i].BT,
               p[i].CT, p[i].TAT, p[i].WT);
    }
    printf("\n");
    printf("Average waiting time = %0.3f",s);
    printf("\n");
    printf("Average turn around time = %0.3f ",t);

    return 0;
}
```

Output: FCFS

```
Enter the number of processes: 3
Enter the arrival time and burst time for all the processes:


Process 1:
Arrival Time: 0
Burst Time: 24


Process 2:
Arrival Time: 0
Burst Time: 3


Process 3:
Arrival Time: 0
Burst Time: 3


Process Name    Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time
1       p1      0               24              24              24              0
2       p2      0               3               27              27              24
3       p3      0               3               30              30              27

Average waiting time = 17.000
Average turn around time = 27.000
```

Code: SJF Pre-Emptive

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

struct process {
    char name[5]; // Process name
    int AT;
    int BT;
    int CT;
    int TAT;
    int WT;
    int RT; // Remaining burst time for the process
    int executed; // Flag to mark if the process has been executed
};

// Function to find the process with the shortest burst time among the arrived processes
int findShortestJob(struct process p[], int n, int current_time) {
    int SJ_index = -1;
    int SJ_burst = INT_MAX;

    for (int i = 0; i < n; i++) {
        if (p[i].AT <= current_time && p[i].RT < SJ_burst && p[i].RT > 0) {
            SJ_index = i;
            SJ_burst = p[i].RT;
        }
    }

    return SJ_index;
}

int main() {
    int n, temp;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct process p[n];
    printf("Enter the arrival time and burst time for all the processes:\n");
    for (int i = 0; i < n; i++) {
        printf("\n\nProcess %d:\n", i + 1);
        sprintf(p[i].name, "p%d", i + 1);
        printf("Arrival time: ");
        scanf("%d", &p[i].AT);
        printf("Burst Time: ");
        scanf("%d", &p[i].BT);
        p[i].RT = p[i].BT;
        p[i].executed = 0;
    }

    int current_time = 0;
    int completed_processes = 0;
    while (completed_processes < n) {
        int SJ_index = findShortestJob(p, n, current_time);
        if (SJ_index == -1) {
            // No process available to execute, move to the next arrival time
            current_time++;
            continue;
        }
```

```
        // Execute the shortest job for one unit of time
        p[SJ_index].RT--;
        current_time++;

        // Check if the process has been completed
        if (p[SJ_index].RT == 0) {
            p[SJ_index].CT = current_time;
            p[SJ_index].TAT = p[SJ_index].CT - p[SJ_index].AT;
            p[SJ_index].WT = p[SJ_index].TAT - p[SJ_index].BT;
            p[SJ_index].executed = 1;
            completed_processes++;
        }
    }

    // Calculate average waiting time and average turnaround time
    int total_wt=0, total_tat=0;

    for(int i=0;i<n;i++)
    {
        total_wt+=p[i].WT;
        total_tat+=p[i].TAT;
    }

    float s=(float)total_wt / (float)n;
    float t=(float)total_tat / (float)n;


    // Print the details of each process
    printf("\n\nProcess\tName\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting
Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%s\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i + 1, p[i].name, p[i].AT, p[i].BT,
               p[i].CT, p[i].TAT, p[i].WT);
    }

    printf("\n");
    printf("Average waiting time = %0.3f",s);
    printf("\n");
    printf("Average turn around time = %0.3f ",t);

    return 0;
}
```

Output: SJF Pre-Emptive

```
Enter the number of processes: 3
Enter the arrival time and burst time for all the processes:


Process 1:
Arrival time: 2
Burst Time: 1


Process 2:
Arrival time: 1
Burst Time: 5


Process 3:
Arrival time: 4
Burst Time: 1


Process Name    Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time
1       p1      2               1               3               1               0
2       p2      1               5               8               7               2
3       p3      4               1               5               1               0

Average waiting time = 0.667
Average turn around time = 3.000
```

Code: SJF Non-Pre-Emptive

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

struct process {
    char name[5]; // Process name
    int AT;
    int BT;
    int CT;
    int TAT;
    int WT;
    int executed; // Flag to mark if the process has been executed
};

// Function to find the process with the shortest burst time among the arrived processes
int findShortestJob(struct process p[], int n, int current_time) {
    int SJ_index = -1;
    int SJ_burst = INT_MAX;

    for (int i = 0; i < n; i++) {
        if (p[i].AT <= current_time && p[i].executed == 0 && p[i].BT < SJ_burst) {
            SJ_index = i;
            SJ_burst = p[i].BT;
        }
    }

    return SJ_index;
}

int main() {
    int n, temp;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct process p[n];
    printf("Enter the arrival time and burst time for all the processes:\n");
    for (int i = 0; i < n; i++) {
        printf("\n\nProcess %d:\n", i + 1);
        sprintf(p[i].name, "p%d", i + 1);
        printf("Arrival time: ");
        scanf("%d", &p[i].AT);
        printf("Burst Time: ");
        scanf("%d", &p[i].BT);
        p[i].executed = 0;
    }

    // Sort processes based on arrival time (FCFS)
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (p[j].AT > p[j + 1].AT) {
                // Swap
                temp = p[j].AT;
                p[j].AT = p[j + 1].AT;
                p[j + 1].AT = temp;

                temp = p[j].BT;
                p[j].BT = p[j + 1].BT;
                p[j + 1].BT = temp;
```

```
            char temp_name[5];
            strcpy(temp_name, p[j].name);
            strcpy(p[j].name, p[j + 1].name);
            strcpy(p[j + 1].name, temp_name);
        }
      }
   }

   int current_time = 0;
   for (int i = 0; i < n; i++) {
      int SJ_index = findShortestJob(p, n, current_time);
      if (SJ_index == -1) {
         // No process available to execute, move to the next arrival time
         current_time = p[i].AT;
         SJ_index = findShortestJob(p, n, current_time);
      }

      // Execute the shortest job
      p[SJ_index].CT = current_time + p[SJ_index].BT;
      p[SJ_index].TAT = p[SJ_index].CT - p[SJ_index].AT;
      p[SJ_index].WT = p[SJ_index].TAT - p[SJ_index].BT;
      p[SJ_index].executed = 1;

      current_time = p[SJ_index].CT;
   }

   // Calculate average waiting time and average turnaround time
   int total_wt=0, total_tat=0;

   for(int i=0;i<n;i++)
   {
      total_wt+=p[i].WT;
      total_tat+=p[i].TAT;
   }

   float s=(float)total_wt / (float)n;
   float t=(float)total_tat / (float)n;


   // Print the details of each process
   printf("\n\nProcess\tName\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting
Time\n");
   for (int i = 0; i < n; i++) {
      printf("%d\t%s\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i + 1, p[i].name, p[i].AT, p[i].BT,
            p[i].CT, p[i].TAT, p[i].WT);
   }

   printf("\n");
   printf("Average waiting time = %0.3f",s);
   printf("\n");
   printf("Average turn around time = %0.3f ",t);


   return 0;
}
```

Output: SJF Non-Pre-Emptive

```
Enter the number of processes: 3
Enter the arrival time and burst time for all the processes:


Process 1:
Arrival time: 2
Burst Time: 1


Process 2:
Arrival time: 1
Burst Time: 5


Process 3:
Arrival time: 4
Burst Time: 1


Process Name    Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time
1       p2      1               5               6               5               0
2       p1      2               1               7               5               4
3       p3      4               1               8               4               3

Average waiting time = 2.333
Average turn around time = 4.667
```

LABORATORY PROGRAM – 2

Question

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

Code: Priority Pre-Emptive

```
#include<stdio.h>
void sort(int proc_id[], int p[], int at[], int bt[], int b[], int n, int priority_type)
{
   int temp;
   for (int i = 0; i < n - 1; i++)
   {
      for (int j = i + 1; j < n; j++)
      {
         if ((priority_type == 1 && p[i] > p[j]) || (priority_type == 2 && p[i] < p[j]))
         {
            temp = p[i];
            p[i] = p[j];
            p[j] = temp;

            temp = at[i];
            at[i] = at[j];
            at[j] = temp;

            temp = bt[i];
            bt[i] = bt[j];
            bt[j] = temp;

            temp = b[i];
            b[i] = b[j];
            b[j] = temp;

            temp = proc_id[i];
            proc_id[i] = proc_id[j];
            proc_id[j] = temp;
         }
      }
   }
}

void main()
{
   int n, c = 0, priority_type;
   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;

   printf("Enter priorities (1 for higher number means higher priority, 2 for lower number means higher
priority): ");
```

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

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, priority_type);

int count = 0, x = 0;
c = 0;
while (count < n)
{
   int found = 0;
   for (int i = 0; i < n; i++)
   {
      if (at[i] <= c && b[i] > 0 && m[i] != 1)
      {
         if (!found || (priority_type == 1 && p[i] > p[x]) || (priority_type == 2 && p[i] < p[x]))
         {
            x = i;
            found = 1;
         }
      }
   }
   if (found && 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;
      }
   }
   else
   {
      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("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%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);
}
```

Output: Priority Pre-Emptive

```
Enter number of processes: 4
Enter priorities (1 for higher number means higher priority, 2 for
lower number means higher priority): 1
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
```

Code: Priority Non-Pre-Emptive

```
#include<stdio.h>

void sort(int proc_id[], int p[], int at[], int bt[], int b[], int n, int priority_type)
{
   int temp;
   for (int i = 0; i < n - 1; i++)
   {
      for (int j = i + 1; j < n; j++)
      {
         if ((priority_type == 1 && p[i] > p[j]) || (priority_type == 2 && p[i] < p[j]))
         {
            temp = p[i];
            p[i] = p[j];
            p[j] = temp;

            temp = at[i];
            at[i] = at[j];
            at[j] = temp;

            temp = bt[i];
            bt[i] = bt[j];
            bt[j] = temp;

            temp = b[i];
            b[i] = b[j];
            b[j] = temp;

            temp = proc_id[i];
            proc_id[i] = proc_id[j];
            proc_id[j] = temp;
         }
      }
   }
}

void main()
{
   int n, c = 0, priority_type;
   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;

   printf("Enter priorities (1 for higher number means higher priority, 2 for lower number means higher
priority): ");
   scanf("%d", &priority_type);

   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, priority_type);

int count = 0, x = 0;
c = 0;
while (count < n)
{
   int found = 0;
   for (int i = 0; i < n; i++)
   {
      if (at[i] <= c && b[i] > 0 && m[i] != 1)
      {
         if (!found || (priority_type == 1 && p[i] > p[x]) || (priority_type == 2 && p[i] < p[x]))
         {
            x = i;
            found = 1;
         }
      }
   }
   if (found && b[x] > 0)
   {
      if (rt[x] == -1)
         rt[x] = c - at[x];

      c += b[x]; // Process runs to completion
      b[x] = 0;
      count++;
      ct[x] = c;
      m[x] = 1;
   }
   else
   {
      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("Priority scheduling (Non-Preemptive):\n");
printf("PID\tPrior\tAT\tBT\tCT\tTAT\tWT\tRT\n");
for (int i = 0; i < n; i++)
   printf("P%d\t%d\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);
}
```

Output: Priority Non-Pre-Emptive

```
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
Does a higher number indicate higher priority? (1 for Yes, 0 for No): 1

Priority scheduling:
PID     Prior   AT      BT      CT      TAT     WT      RT
P1      10      0       5       5       5       0       0
P2      20      1       4       12      11      7       7
P3      30      2       2       8       6       4       4
P4      40      4       1       6       2       1       1

Average turnaround time: 6.000000ms

Average waiting time: 3.000000ms
```

Code: Round Robin

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

struct Process {
   char name[5]; // Process name
   int AT, BT, ST[20], WT, FT, TAT, pos, CT, RT;
};

int quant;

int main() {
   int n, temp;
   printf("Enter the number of processes: ");
   scanf("%d", &n);
   struct Process p[n];
   printf("Enter the arrival time and burst time for all the processes:\n");
   for (int i = 0; i < n; i++)
   {
      printf("\n\nProcess %d:\n", i + 1);
      sprintf(p[i].name, "p%d", i + 1);
      printf("Arrival Time: ");
      scanf("%d", &p[i].AT);
      printf("Burst Time: ");
      scanf("%d", &p[i].BT);
      p[i].pos = i + 1;
   }

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

   // Initializing variables
   int c = n, s[n][20];
   float time = 0, mini = INT_MAX, b[n], a[n];

   // Initializing burst and arrival time arrays
   int index = -1;
   for(int i = 0; i < n; i++) {
      b[i] = p[i].BT;
      a[i] = p[i].AT;
      for(int j = 0; j < 20; j++) {
         s[i][j] = -1;
      }
   }

   int tot_wt = 0, tot_tat = 0, tot_rt = 0;
   bool flag = false;

   while(c != 0) {
      mini = INT_MAX;
      flag = false;

      for(int i = 0; i < n; i++) {
         float p = time + 0.1;
         if(a[i] <= p && mini > a[i] && b[i] > 0) {
            index = i;
```

```
                mini = a[i];
                flag = true;
            }
        }

        // If no process is available at this moment
        if(!flag) {
            time++;
            continue;
        }

        // Calculating start time
        int j = 0;
        while(s[index][j] != -1) {
            j++;
        }

        if(s[index][j] == -1) {
            s[index][j] = time;
            p[index].ST[j] = time;
        }

        if(b[index] <= quant) {
            time += b[index];
            b[index] = 0;
        }
        else {
            time += quant;
            b[index] -= quant;
        }

        if(b[index] > 0) {
            a[index] = time + 0.1;
        }

        // Calculating arrival, burst, final times
        if(b[index] == 0) {
            c--;
            p[index].FT = time;
            p[index].WT = p[index].FT - p[index].AT - p[index].BT;
            tot_wt += p[index].WT;
            p[index].TAT = p[index].BT + p[index].WT;
            tot_tat += p[index].TAT;
            p[index].CT = time;
            p[index].RT = p[index].ST[0] - p[index].AT;
            tot_rt += p[index].RT;
        }
    } // end of while loop

    // Printing output
    printf("\n\nProcess\tName\tArrival Time\tBurst Time\tStart time\tFinal time\tCompletion Time\tTurnaround
Time\tWaiting Time\tResponse Time\n");

    for(int i = 0; i < n; i++) {
        printf("%d\t%s\t%d\t\t%d\t\t", p[i].pos, p[i].name, p[i].AT, p[i].BT);
        int j = 0;
        while(s[i][j] != -1) {
            printf("%d ", p[i].ST[j]);
            j++;
        }
```

```
        printf("\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", p[i].FT, p[i].CT, p[i].TAT, p[i].WT, p[i].RT);
    }

    // Calculating average wait time, turnaround time, and response time
    double avg_wt = (double) tot_wt / n;
    double avg_tat = (double) tot_tat / n;
    double avg_rt = (double) tot_rt / n;

    // Printing average wait time, turnaround time, and response time
    printf("\nThe average wait time is : %lf\n", avg_wt);
    printf("The average Turnaround time is : %lf\n", avg_tat);
    printf("The average Response time is : %lf\n", avg_rt);

    return 0;
}
```

Output: Round Robin

```
Enter the number of processes: 3
Enter the arrival time and burst time for all the processes:


Process 1:
Arrival Time: 0
Burst Time: 5


Process 2:
Arrival Time: 1
Burst Time: 4


Process 3:
Arrival Time: 2
Burst Time: 2
Enter the quantum: 2


Process Name    Arrival Time    Burst Time      Start time      Final time      Completion Time Turnaround Time Waiting Time    Response Time
1       p1      0               5               0 6 10          11              11              11              6       0
2       p2      1               4               2 8             10              10              9               5       1
3       p3      2               2               4               6               6               4               2       2
The average wait time is : 4.333333
The average Turnaround time is : 8.000000
The average Response time is : 1.000000
```

LABORATORY PROGRAM – 3

Question

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

Code

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

struct process
{
   char name[5];
   int AT;
   int BT;
   int CT;
   int TAT;
   int WT;
   int isSystem; // 1 for system process, 0 for user process
};

void FCFS(struct process p[], int n, int *current_time)
{
   for (int i = 0; i < n; i++) {
      if (*current_time < p[i].AT)
         *current_time = p[i].AT;

      p[i].CT = *current_time + p[i].BT;
      p[i].TAT = p[i].CT - p[i].AT;
      p[i].WT = p[i].TAT - p[i].BT;

      *current_time = p[i].CT;
   }
}

int main()
{
   int n, sys_count = 0, user_count = 0;
   printf("Enter the number of processes: ");
   scanf("%d", &n);
   struct process p[n], system_p[n], user_p[n];
   printf("Enter the arrival time, burst time and type (1 for system process, 0 for user process) for all the
processes:\n");

   for (int i = 0; i < n; i++)
   {
      printf("\n\nProcess %d:\n", i + 1);
      sprintf(p[i].name, "p%d", i + 1);
      printf("Arrival time: ");
```

```
        scanf("%d", &p[i].AT);
        printf("Burst Time: ");
        scanf("%d", &p[i].BT);
        printf("Type (1 for system, 0 for user): ");
        scanf("%d", &p[i].isSystem);

        if (p[i].isSystem)
        {
            system_p[sys_count++] = p[i];
        } else
        {
            user_p[user_count++] = p[i];
        }
    }

    for (int i = 0; i < sys_count - 1; i++)
    {
        for (int j = 0; j < sys_count - i - 1; j++)
        {
            if (system_p[j].AT > system_p[j + 1].AT)
            {
                struct process temp = system_p[j];
                system_p[j] = system_p[j + 1];
                system_p[j + 1] = temp;
            }
        }
    }

    for (int i = 0; i < user_count - 1; i++) {
        for (int j = 0; j < user_count - i - 1; j++) {
            if (user_p[j].AT > user_p[j + 1].AT) {
                struct process temp = user_p[j];
                user_p[j] = user_p[j + 1];
                user_p[j + 1] = temp;
            }
        }
    }

    int current_time = 0;
    int total_wt = 0, total_tat = 0;

    printf("\n\nProcess\tName\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting
Time\tType\n");

    int i = 0, j = 0;
    while (i < sys_count || j < user_count) {
        if (i < sys_count && (j >= user_count || system_p[i].AT <= current_time)) {
            if (current_time < system_p[i].AT)
                current_time = system_p[i].AT;

            system_p[i].CT = current_time + system_p[i].BT;
            system_p[i].TAT = system_p[i].CT - system_p[i].AT;
            system_p[i].WT = system_p[i].TAT - system_p[i].BT;

            current_time = system_p[i].CT;

            printf("%d\t%s\t%d\t\t%d\t\t%d\t\t%d\t\tSystem\n", i + 1, system_p[i].name, system_p[i].AT,
system_p[i].BT, system_p[i].CT, system_p[i].TAT, system_p[i].WT);
            total_wt += system_p[i].WT;
            total_tat += system_p[i].TAT;
```

```
            i++;
        } else if (j < user_count) {
            if (current_time < user_p[j].AT)
                current_time = user_p[j].AT;

            user_p[j].CT = current_time + user_p[j].BT;
            user_p[j].TAT = user_p[j].CT - user_p[j].AT;
            user_p[j].WT = user_p[j].TAT - user_p[j].BT;

            current_time = user_p[j].CT;

            printf("%d\t%s\t%d\t\t%d\t\t%d\t\t%d\t\tUser\n", i + 1 + j, user_p[j].name, user_p[j].AT,
user_p[j].BT, user_p[j].CT, user_p[j].TAT, user_p[j].WT);
            total_wt += user_p[j].WT;
            total_tat += user_p[j].TAT;

            j++;
        }
    }

    float avg_wt = (float)total_wt / n;
    float avg_tat = (float)total_tat / n;

    printf("\nAverage waiting time = %0.3f", avg_wt);
    printf("\nAverage turn around time = %0.3f\n", avg_tat);

    return 0;
}
```

Output

```
Enter the number of processes: 3
Enter the arrival time, burst time and type (1 for system process, 0 for user process) for all the processes:


Process 1:
Arrival time: 2
Burst Time: 1
Type (1 for system, 0 for user): 1


Process 2:
Arrival time: 1
Burst Time: 5
Type (1 for system, 0 for user): 0


Process 3:
Arrival time: 4
Burst Time: 1
Type (1 for system, 0 for user): 1


Process Name    Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time    Type
1       p2      1               5               6               5               0               User
1       p1      2               1               7               5               4               System
2       p3      4               1               8               4               3               System

Average waiting time = 2.333
Average turn around time = 4.667
```

LABORATORY 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: Rate-Monotonic Scheduling

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

#define MAX_TASKS 10

typedef struct
{
   int Ti;
   int Ci;
   int deadline;
   int RT; //Remaining_Time
   int id;
} Task;

void Input(Task tasks[], int *n_tasks)
{
   printf("Enter number of tasks: ");
   scanf("%d", n_tasks);

   for (int i = 0; i < *n_tasks; i++)
   {
      tasks[i].id = i + 1;
      printf("Enter Ti of task %d: ", i + 1);
      scanf("%d", &tasks[i].Ti);
      printf("Enter execution time of task %d: ", i + 1);
      scanf("%d", &tasks[i].Ci);
      tasks[i].deadline = tasks[i].Ti; // In RM, deadline is equal to Ti
      tasks[i].RT = tasks[i].Ci;
   }
}

int compare_by_period(const void *a, const void *b)
{
   return ((Task*)a)->Ti - ((Task*)b)->Ti;
}

void RMS(Task tasks[], int n_tasks, int time_frame)
{
   qsort(tasks, n_tasks, sizeof(Task), compare_by_period);

   printf("\nRate-Monotonic Scheduling:\n");
   for (int time = 0; time < time_frame; time++)
   {
      int s_task = -1;
      for (int i = 0; i < n_tasks; i++)
```

```
        {
            if (time % tasks[i].Ti == 0)
            {
                tasks[i].RT = tasks[i].Ci;
            }
            if (tasks[i].RT > 0 && (s_task == -1 || tasks[i].Ti < tasks[s_task].Ti))
            {
                s_task = i;
            }
        }

        if (s_task != -1)
        {
            printf("Time %d: Task %d\n", time, tasks[s_task].id);
            tasks[s_task].RT--;
        } else
        {
            printf("Time %d: Idle\n", time);
        }
    }
}

int main()
{
    Task tasks[MAX_TASKS];
    int n_tasks;
    int time_frame;

    Input(tasks, &n_tasks);

    printf("Enter time frame for simulation: ");
    scanf("%d", &time_frame);

    RMS(tasks, n_tasks, time_frame);

    return 0;
}
```

Output: Rate-Monotonic Scheduling

```
Enter number of tasks: 2
Enter Ti of task 1: 5
Enter execution time of task 1: 3
Enter Ti of task 2: 10
Enter execution time of task 2: 2
Enter time frame for simulation: 10

Rate-Monotonic Scheduling:
Time 0: Task 1
Time 1: Task 1
Time 2: Task 1
Time 3: Task 2
Time 4: Task 2
Time 5: Task 1
Time 6: Task 1
Time 7: Task 1
Time 8: Idle
Time 9: Idle
```

Code: Earliest-deadline First Scheduling

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

#define MAX_TASKS 10

typedef struct
{
   int Ti;
   int Ci;
   int deadline;
   int RT; // remaining time
   int n_deadline; // next_deadline
   int id;
} Task;

void Input(Task tasks[], int *n_tasks)
{
   printf("Enter number of tasks: ");
   scanf("%d", n_tasks);

   for (int i = 0; i < *n_tasks; i++)
   {
      tasks[i].id = i + 1;
      printf("Enter Ti of task %d: ", i + 1);
      scanf("%d", &tasks[i].Ti);
      printf("Enter execution time of task %d: ", i + 1);
      scanf("%d", &tasks[i].Ci);
      printf("Enter deadline of task %d: ", i + 1);
      scanf("%d", &tasks[i].deadline);
      tasks[i].RT = tasks[i].Ci;
      tasks[i].n_deadline = tasks[i].deadline; // Initialize the next deadline
   }
}

void EDF(Task tasks[], int n_tasks, int time_frame)
{
   printf("\nEarliest-Deadline First Scheduling:\n");
   for (int time = 0; time < time_frame; time++)
   {
      int s_task = -1;

      for (int i = 0; i < n_tasks; i++)
      {
         if (time % tasks[i].Ti == 0)
         {
            tasks[i].RT = tasks[i].Ci;
            tasks[i].n_deadline = time + tasks[i].deadline;
         }
      }

      for (int i = 0; i < n_tasks; i++)
      {
         if (tasks[i].RT > 0 && (s_task == -1 || tasks[i].n_deadline < tasks[s_task].n_deadline)) {
            s_task = i;
         }
      }

      if (s_task != -1)
```

```
        {
            printf("Time %d: Task %d\n", time, tasks[s_task].id);
            tasks[s_task].RT--;
        } else
        {
            printf("Time %d: Idle\n", time);
        }
    }
}

int main()
{
    Task tasks[MAX_TASKS];
    int n_tasks;
    int time_frame;

    Input(tasks, &n_tasks);

    printf("Enter time frame for simulation: ");
    scanf("%d", &time_frame);

    EDF(tasks, n_tasks, time_frame);

    return 0;
}
```

Output: Earliest-deadline First Scheduling

```
Enter number of tasks: 2
Enter Ti of task 1: 10
Enter execution time of task 1: 3
Enter deadline of task 1: 7
Enter Ti of task 2: 5
Enter execution time of task 2: 2
Enter deadline of task 2: 4
Enter time frame for simulation: 10

Earliest-Deadline First Scheduling:
Time 0: Task 2
Time 1: Task 2
Time 2: Task 1
Time 3: Task 1
Time 4: Task 1
Time 5: Task 2
Time 6: Task 2
Time 7: Idle
Time 8: Idle
Time 9: Idle
```

Code: Proportional Scheduling

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

typedef struct
{
   char name[5];
   int tickets;
} Process;

int main()
{
   int n, total_tickets = 0;
   float total_T =0.0;

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

   Process p[n];

   srand(time(NULL));

   for (int i = 0; i < n; i++)
   {
      printf("\nProcess %d:\n", i + 1);
      sprintf(p[i].name, "P%d", i + 1);
      printf("Tickets: ");
      scanf("%d", &p[i].tickets);
      total_tickets += p[i].tickets;
      total_T +=p[i].tickets;
   }

   printf("\n--- Proportional Share Scheduling ---\n");
   printf("Enter the Time Period for scheduling: ");
   int m;
   scanf("%d",&m);

   for (int i = 0; i < m; i++)
   {
      int winning_ticket = rand() % total_tickets + 1;
      int accumulated_tickets = 0;
      int winner_index;

      for (int j = 0; j < n; j++)
      {
         accumulated_tickets += p[j].tickets;
         if (winning_ticket <= accumulated_tickets)
         {
            winner_index = j;
            break;
         }
      }

      printf("Tickets picked: %d, Winner: %s\n", winning_ticket, p[winner_index].name);
   }

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

```
        printf("\nThe Process: %s gets %0.2f%% of Processor Time.\n", p[i].name, ((p[i].tickets / total_T) * 100));
    }

    return 0;
}
```

Output: Proportional Scheduling

```
Enter the number of Processes: 3

Process 1:
Tickets: 10

Process 2:
Tickets: 20

Process 3:
Tickets: 30

--- Proportional Share Scheduling ---
Enter the Time Period for scheduling: 5
Tickets picked: 25, Winner: P2
Tickets picked: 8, Winner: P1
Tickets picked: 32, Winner: P3
Tickets picked: 16, Winner: P2
Tickets picked: 12, Winner: P2

The Process: P1 gets 16.67% of Processor Time.

The Process: P2 gets 33.33% of Processor Time.

The Process: P3 gets 50.00% of Processor Time.
```

LABORATORY PROGRAM – 5

Question

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

Code

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

#define BufferSize 10

int buffer[BufferSize];
int in = 0, out = 0;

int maxP, maxC;
int empty = BufferSize, full = 0, mutex = 1;

void wait(int *S)
{
   while (*S <= 0);
   (*S)--;
}

void signal(int *S)
 {
   (*S)++;
}

void producer()
{
   int pItems = 0;
   while (pItems < maxP)
   {
      int item = rand();
      wait(&empty);
      wait(&mutex);
      buffer[in] = item;
      printf("Producer produced item %d at %d\n", item, in);
      in = (in + 1) % BufferSize;
      signal(&mutex);
      signal(&full);
      pItems++;
   }
}

void consumer()
{
   int cItems = 0;
   while (cItems < maxC)
   {
      wait(&full);
      wait(&mutex);
      int item = buffer[out];
```

```
        printf("Consumer consumed item %d from %d\n", item, out);
        out = (out + 1) % BufferSize;
        signal(&mutex);
        signal(&empty);
        cItems++;
    }
}

int main()
{
    int numPs, numCs;
    printf("Enter number of producers: ");
    scanf("%d", &numPs);
    printf("Enter number of consumers: ");
    scanf("%d", &numCs);

    printf("Enter maximum items each producer can produce: ");
    scanf("%d", &maxP);

    printf("Enter maximum items each consumer can consume: ");
    scanf("%d", &maxC);
    for(int i=1;i<=numPs;i++)
    {
        producer();
    }
    for(int i=1;i<=numCs;i++)
    {
        consumer();
    }


    return 0;
}
```

Output

```
Enter number of producers: 1
Enter number of consumers: 1
Enter maximum items each producer can produce: 3
Enter maximum items each consumer can consume: 3
Producer produced item 41 at 0
Producer produced item 18467 at 1
Producer produced item 6334 at 2
Consumer consumed item 41 from 0
Consumer consumed item 18467 from 1
Consumer consumed item 6334 from 2
```

LABORATORY PROGRAM – 6

Question

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

Code

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

#define MAX_PHILOSOPHERS 100

int mutex = 1;
int mutex2 = 2;

int philosophers[MAX_PHILOSOPHERS];

void wait(int *sem) {
   while (*sem <= 0);
   (*sem)--;
}

void Signal(int *sem) {
   (*sem)++;
}

void* one_eat_at_a_time(void* arg) {
   int philosopher = *((int*) arg);

   wait(&mutex);
   printf("Philosopher %d is granted to eat\n", philosopher + 1);
   sleep(1);
   printf("Philosopher %d has finished eating\n", philosopher + 1);
   Signal(&mutex);

   return NULL;
}

void* two_eat_at_a_time(void* arg) {
   int philosopher = *((int*) arg);

   wait(&mutex2);
   printf("Philosopher %d is granted to eat\n", philosopher + 1);
   sleep(1);
   printf("Philosopher %d has finished eating\n", philosopher + 1);
   Signal(&mutex2);

   return NULL;
}

int main() {
   int N;
```

```
    printf("Enter the total number of philosophers: ");
    scanf("%d", &N);

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

    int hungry_philosophers[hungry_count];
    for (int i = 0; i < hungry_count; i++) {
        printf("Enter philosopher %d position (1 to %d): ", i + 1, N);
        scanf("%d", &hungry_philosophers[i]);
        hungry_philosophers[i]--;
    }

    pthread_t thread[hungry_count];

    int choice;

    do {
        printf("\n1. One can eat at a time\n2. Two can eat at a time\n3. Exit\nEnter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Allow one philosopher to eat at any time\n");
                for (int i = 0; i < hungry_count; i++) {
                    philosophers[i] = hungry_philosophers[i];
                    pthread_create(&thread[i], NULL, one_eat_at_a_time, &philosophers[i]);
                }
                for (int i = 0; i < hungry_count; i++) {
                    pthread_join(thread[i], NULL);
                }
                break;
            case 2:
                printf("Allow two philosophers to eat at the same time\n");
                for (int i = 0; i < hungry_count; i++) {
                    philosophers[i] = hungry_philosophers[i];
                    pthread_create(&thread[i], NULL, two_eat_at_a_time, &philosophers[i]);
                }
                for (int i = 0; i < hungry_count; i++) {
                    pthread_join(thread[i], NULL);
                }
                break;
            case 3:
                printf("Exit\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 3);

    return 0;
}
```

Output

```
Enter the total number of philosophers: 5
How many are hungry: 3
Enter philosopher 1 position (1 to 5): 2
Enter philosopher 2 position (1 to 5): 4
Enter philosopher 3 position (1 to 5): 5

1. One can eat at a time
2. Two can eat at a time
3. Exit
Enter your choice: 1
Allow one philosopher to eat at any time
Philosopher 4 is granted to eat
Philosopher 4 has finished eating
Philosopher 2 is granted to eat
Philosopher 2 has finished eating
Philosopher 5 is granted to eat
Philosopher 5 has finished eating

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

LABORATORY PROGRAM – 7

Question

Write a C program to simulate Banker's algorithm for the purpose of deadlock avoidance.

Code

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

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

bool isSafe(int n, int m, int avai[m], int alloc[n][m], int need[n][m], int safeSequence[n])
{
   int work[m];
   for (int i = 0; i < m; i++)
      work[i] = avai[i];

   bool finish[n];
   for (int i = 0; i < n; i++)
      finish[i] = false;

   int count = 0;

   while (count < n)
   {
      bool found = false;
      for (int p = 0; p < n; p++)
      {
         if (!finish[p])
         {
            int j;
            for (j = 0; j < m; j++)
            {
               if (need[p][j] > work[j])
                  break;
            }
            if (j == m)
            {
               for (int k = 0; k < m; k++)
                  work[k] += alloc[p][k];
               safeSequence[count++] = p;
               finish[p] = true;
               found = true;
            }
         }
      }
```

```
        if (!found)
            return false;
    }
    return true;
}

void printTable(int n, int m, int alloc[n][m], int max[n][m], int need[n][m], int avai[m], int safeSequence[n],
bool is_safe)
{
    printf("\nProcess\tAllocation\tMaximum\t\tNeed\n");
    for (int i = 0; i < n; i++)
    {
        printf("P%d\t", i);
        for (int j = 0; j < m; j++) printf("%-2d ", alloc[i][j]);
        printf("\t");
        for (int j = 0; j < m; j++) printf("%-2d ", max[i][j]);
        printf("\t");
        for (int j = 0; j < m; j++) printf("%-2d ", need[i][j]);
        printf("\n");
    }

    if (is_safe)
    {
        int work[m];
        for (int i = 0; i < m; i++) work[i] = avai[i];

        for (int count = 0; count < n; count++)
        {
            int p = safeSequence[count];
            for (int j = 0; j < m; j++)
            {
                work[j] += alloc[p][j];
            }
            printf("After P%d execution, available: ", p);
            for (int j = 0; j < m; j++)
            {
                printf("%-2d ", work[j]);
            }
            printf("\n");
        }
    }
}

int main()
{
    int n, m;

    printf("Enter the number of resources: ");
    scanf("%d", &m);

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

    int avai[m];
    int max[n][m];
    int alloc[n][m];
    int need[n][m];
    int safeSequence[n];

    printf("Enter the available m (R1 R2 ... Rn):\n");
```

```
    for (int i = 0; i < m; i++)
    {
        scanf("%d", &avai[i]);
    }

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

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

    Need(n, m, max, alloc, need);

    bool is_safe = isSafe(n, m, avai, alloc, need, safeSequence);

    printTable(n, m, alloc, max, need, avai, safeSequence, is_safe);

    if (is_safe)
    {
        printf("\nThe system is in a safe state.\n");
        printf("Safe sequence is: ");
        for (int i = 0; i < n; i++) {
            printf("P%d ", safeSequence[i]);
        }
        printf("\n");
    }
    else
    {
        printf("The system is not in a safe state.\n");
    }

    return 0;
}
```

Output

```
Enter the number of resources: 3
Enter the number of processes: 2
Enter the available m (R1 R2 ... Rn):
3 3 2
Enter the maximum resource matrix:
Process 0: 7 5 3
Process 1: 3 2 2
Enter the allocation matrix:
Process 0: 0 1 0
Process 1: 2 0 0

Process Allocation      Maximum         Need
P0      0  1  0        7  5  3         7  4  3
P1      2  0  0        3  2  2         1  2  2
The system is not in a safe state.
```

LABORATORY 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 in safe state.\n");
    printf("Safe Sequence is: ");
    for (i = 0; i < n; i++) {
        printf("P%d ", safeSeq[i]);
    }
    printf("\n");
}

return 0;
}
```

Output

```
Enter the number of processes: 3
Enter the number of resources: 2
Enter the allocation matrix:
Process 0: 0 1
Process 1: 2 0
Process 2: 3 0
Enter the request matrix:
Process 0: 0 0
Process 1: 2 0
Process 2: 0 0
Enter the available resources: 0 0
System is in safe state.
Safe Sequence is: P0 P2 P1
```

LABORATORY PROGRAM – 9

Question

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

Code

```
#include <stdio.h>

#define MAX 25

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

   for (i = 1; i <= nf; i++) {
      for (j = 1; j <= nb; j++) {
         if (bf[j] != 1) {
            temp = b[j] - f[i];
            if (temp >= 0) {
               ff[i] = j;
               frag[i] = temp;
               bf[j] = 1;
               break;
            }
         }
      }
   }

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

void bestFit(int nb, int nf, int b[], int f[]) {
   int frag[MAX], bf[MAX] = {0}, ff[MAX] = {0};
   int i, j, temp, lowest = 10000;

   for (i = 1; i <= nf; i++) {
      for (j = 1; j <= nb; j++) {
         if (bf[j] != 1) {
            temp = b[j] - f[i];
            if (temp >= 0 && lowest > temp) {
               ff[i] = j;
               lowest = temp;
            }
```

```
            }
        }
        frag[i] = lowest;
        bf[ff[i]] = 1;
        lowest = 10000;
    }

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

void worstFit(int nb, int nf, int b[], int f[]) {
    int frag[MAX], bf[MAX] = {0}, ff[MAX] = {0};
    int i, j, temp, highest = 0;

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

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

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

    printf("\nEnter the number of blocks:");
    scanf("%d", &nb);
    printf("Enter the number of files:");
    scanf("%d", &nf);
    printf("\nEnter the size of the blocks:-\n");
    for (int i = 1; i <= nb; i++) {
        printf("Block %d:", i);
```

```
        scanf("%d", &b[i]);
    }
    printf("Enter the size of the files :-\n");
    for (int i = 1; i <= nf; i++) {
        printf("File %d:", i);
        scanf("%d", &f[i]);
    }

    int b1[MAX], b2[MAX], b3[MAX];
    for (int i = 1; i <= nb; i++) {
        b1[i] = b[i];
        b2[i] = b[i];
        b3[i] = b[i];
    }

    firstFit(nb, nf, b1, f);
    bestFit(nb, nf, b2, f);
    worstFit(nb, nf, b3, f);

    return 0;
}
```

Output

```
Enter the number of blocks:2
Enter the number of files:2

Enter the size of the blocks:-
Block 1:2
Block 2:3
Enter the size of the files :-
File 1:1
File 2:2

Memory Management Scheme - First Fit
File_no:        File_size :     Block_no:       Block_size:     Fragment
1               1               1               2               1
2               2               2               3               1

Memory Management Scheme - Best Fit
File No File Size       Block No        Block Size      Fragment
1               1               1               2               1
2               2               2               3               1

Memory Management Scheme - Worst Fit
File_no:        File_size :     Block_no:       Block_size:     Fragment
1               1               2               3               2
2               2               Not Allocated
```

LABORATORY PROGRAM – 10

Question

Write a C program to simulate page replacement algorithms a) FIFO, b) LRU, c) Optimal.

Code

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

void print_frames(int F[], int n) {
    for (int i = 0; i < n; i++) {
        if (F[i] == -1)
            printf("- ");
        else
            printf("%d ", F[i]);
    }
    printf("\n");
}

void fifo(int F[], int n, int P[], int m) {
    int index = 0;
    int p_fault = 0;

    for (int i = 0; i < m; i++) {
        bool found = false;
        for (int j = 0; j < n; j++) {
            if (F[j] == P[i]) {
                found = true;
                break;
            }
        }

        if (!found) {
            F[index] = P[i];
            index = (index + 1) % n;
            p_fault++;
            printf("PF No. %d: ", p_fault);
            print_frames(F, n);
        }
    }

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

void lru(int F[], int n, int P[], int m) {
    int LU[n];
    int p_fault = 0;
    int count = 0;

    for (int i = 0; i < n; i++) {
        F[i] = -1;
        LU[i] = -1;
    }
```

```
    for (int i = 0; i < m; i++) {
        int j;
        bool found = false;

        for (j = 0; j < n; j++) {
            if (F[j] == P[i]) {
                found = true;
                LU[j] = count++;
                break;
            }
        }

        if (!found) {
            int lru_index = 0;
            for (j = 1; j < n; j++) {
                if (LU[j] < LU[lru_index]) {
                    lru_index = j;
                }
            }
            F[lru_index] = P[i];
            LU[lru_index] = count++;
            p_fault++;
            printf("PF No. %d: ", p_fault);
            print_frames(F, n);
        }
    }

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

void optimal(int F[], int n, int P[], int m) {
    int p_fault = 0;

    for (int i = 0; i < n; i++) {
        F[i] = -1;
    }

    for (int i = 0; i < m; i++) {
        bool found = false;
        for (int j = 0; j < n; j++) {
            if (F[j] == P[i]) {
                found = true;
                break;
            }
        }

        if (!found) {
            int replace_i = -1;
            int far = i + 1;

            for (int j = 0; j < n; j++) {
                int k;
                for (k = i + 1; k < m; k++) {
                    if (F[j] == P[k]) {
                        if (k > far) {
                            far = k;
                            replace_i = j;
                        }
                        break;
```

```
                }
            }

            if (k == m) {
                replace_i = j;
                break;
            }
        }

        if (replace_i == -1) {
            replace_i = 0;
        }

        F[replace_i] = P[i];
        p_fault++;
        printf("PF No. %d: ", p_fault);
        print_frames(F, n);
    }
  }

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

int main() {
  int n;
  int m;

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

  printf("Enter the length of reference string: ");
  scanf("%d", &m);

  int P[m];
  printf("Enter the reference string: ");
  for (int i = 0; i < m; i++) {
    scanf("%d", &P[i]);
  }

  int F[n];
  for (int i = 0; i < n; i++) {
    F[i] = -1;
  }

  printf("FIFO Page Replacement Process:\n");
  fifo(F, n, P, m);

  for (int i = 0; i < n; i++) {
    F[i] = -1;
  }

  printf("\nLRU Page Replacement Process:\n");
  lru(F, n, P, m);

  for (int i = 0; i < n; i++) {
    F[i] = -1;
  }

  printf("\nOptimal Page Replacement Process:\n");
  optimal(F, n, P, m);
```

```
    return 0;
}
```

Output

```
Enter the number of Frames: 3
Enter the length of reference string: 6
Enter the reference string: 1 4 2 3 2 1
FIFO Page Replacement Process:
PF No. 1: 1 - -
PF No. 2: 1 4 -
PF No. 3: 1 4 2
PF No. 4: 3 4 2
PF No. 5: 3 1 2
FIFO Page Faults: 5

LRU Page Replacement Process:
PF No. 1: 1 - -
PF No. 2: 1 4 -
PF No. 3: 1 4 2
PF No. 4: 3 4 2
PF No. 5: 3 1 2
LRU Page Faults: 5

Optimal Page Replacement Process:
PF No. 1: 1 - -
PF No. 2: 1 4 -
PF No. 3: 1 2 -
PF No. 4: 1 2 3
Optimal Page Faults: 4
```