VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

Operating Systems

(22CS4PCOPS)

Submitted by:

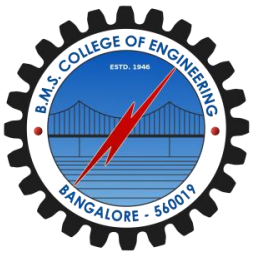
Unnati Bansal(1WA23CS025)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING



B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)

BENGALURU-560019

Feb 2025 - June 2025

B. M. S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "**Operating Systems**" carried out by **Unnati Bansal (1WA23CS025),** 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 2024-25. The Lab report has been approved as it satisfies the academic requirements in respect of **Operating Systems - (22CS4PCOPS)** work prescribed for the said degree.

Dr Seema patil
Associate Professor
Department of CSE
BMSCE, Bengaluru

Dr Kavitha Sooda
Professor and Head
Department of CSE
BMSCE, Bengaluru

Table Of Contents

Course Outcomes

CO1: Apply the different concepts and functionalities of OS

CO2: Analyse 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.

Lab 1:

Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time.

→FCFS

→ SJF (Non-preemptive)

→ SJF (Preemptive)

```
#include <stdio.h>


void calculateTimes(int processes[], int n, int at[], int bt[], int ct[], int tat[], int wt[]) {
    int completion = 0;

    for (int i = 0; i < n; i++) {
        if (completion < at[i]) {
            completion = at[i];
        }
        completion += bt[i];
        ct[i] = completion;
        tat[i] = ct[i] - at[i];
        wt[i]=tat[i]-bt[i];
    }
}


void displayResults(int processes[], int n, int at[], int bt[], int ct[], int tat[], int wt[]) {
    float total_tat = 0, total_wt = 0;

    printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\n");
    for (int i = 0; i < n; i++) {
        total_tat += tat[i];
        total_wt += wt[i];
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", processes[i], at[i], bt[i], ct[i], tat[i], wt[i]);
    }

    printf("\nAverage Turnaround Time = %.2f", total_tat / n);
    printf("\nAverage Waiting Time = %.2f\n", total_wt / n);
}

int main() {
    int n ;
```

```
    printf("Enter no. of processes : ");
    scanf("%d",&n);
    int processes[n];
    for(int i=0;i<n;i++){
        processes[i]=i+1;
          }

    int at[n];
    for(int i=0;i<n;i++){
        printf("Enter arrival time of process %d : ",processes[i]);
        scanf("%d",&at[i]);
          }
    int bt[n];
    for(int i=0;i<n;i++){
        printf("Enter burst time of process %d : ",processes[i]);
        scanf("%d",&bt[i]);
          }
    int ct[n], tat[n], wt[n], rt[n];

    calculateTimes(processes, n, at, bt, ct, tat, wt);
    displayResults(processes, n, at, bt, ct, tat, wt);

    return 0;
}
```

Output:

```
Enter no. of processes : 4
Enter arrival time of process 1 : 0
Enter arrival time of process 2 : 0
Enter arrival time of process 3 : 0
Enter arrival time of process 4 : 0
Enter burst time of process 1 : 7
Enter burst time of process 2 : 3
Enter burst time of process 3 : 4
Enter burst time of process 4 : 6

Process AT      BT      CT      TAT     WT
1       0       7       7       7       0
2       0       3       10      10      7
3       0       4       14      14      10
4       0       6       20      20      14

Average Turnaround Time = 12.75
Average Waiting Time = 7.75
```

1) write a C program to stimulate the following CPU scheduling algorithm to find turnaround time and waiting time

a) FCFS

b) SJF (both preemptive and non-preemptive)

a)
```
#include <stdio.h>
void calculateTimes(int processes[], int n,
int at[], int bt[], int ct[], int tat[],
int wt[]) {
    for (int i=0; i<n; i++) {
        if (completion < at[i]){
            completion = at[i];
        }
        completion += bt[i];
        ct[i] = completion;
        tat[i] = ct[i] - at[i];
        wt[i] = tat[i] - bt[i];
    }
}
void displayResults(int processes[], int n,
int at[], int bt[], int ct[], int tat[],
int wt[]) {
    float total_tat = 0, total_wt = 0;
    printf("\n Process\tAT\tBT\tCT\tTAT\tWT\n");
    for (int i=0; i<n; i++) {
        total_tat += tat[i];
        total_wt += wt[i];
        printf("%d\t%d\t%d\t%d\t%d\t%d\n"
processes[i], at[i], bt[i], ct[i], tat[i], wt[i]);
```

```
    printf("\n Average Turnaround Time = %.2f", total_tat/n);
    printf("\n Average Waiting Time = %.2f\n", total_wt/n);
}
int main() {
    int n;
    printf("Enter no. of processes : ");
    scanf("%d", &n);
    int processes[n];
    for(int i=0; i<n; i++) {
        processes[i] = i+1;
    }
    int at[n];
    for(int i=0; i<n; i++) {
        printf("Enter arrival time of process %d",
                processes[i]);
        scanf("%d", &at[i]);
    }
    int bt[n];
    for(int i=0; i<n; i++) {
        printf("Enter burst time of process %d",
                processes[i]);
        scanf("%d", &bt[i]);
    }
    int ct[n], tat[n], wt[n], rt[n];
    calculateTimes(processes, n, at, bt, ct, tat, wt);
    displayResults(processes, n, at, bt, ct, tat, wt);
    return 0;
}
```

Output

```
Enter no. of processes : 4
Enter arrival time of process 1 : 0
Enter arrival time of process 2 : 0
Enter arrival time of process 3 : 0
Enter arrival time of process 4 : 0
Enter burst time of process 1 : 7
Enter burst time of process 2 : 3
Enter burst time of process 3 : 4
Enter burst time of process 4 : 6
```

Process	AT	BT	CT	TAT	WT
1	0	7	7	7	0
2	0	3	10	10	7
3	0	4	14	14	10
4	0	6	20	20	14

Average Turnaround Time = 12.75
Average Waiting Time = 7.75

6) i) Non preemptive SJF

```
#include<stdio.h>
typedef struct{
   int id, AT, BT, TAT, RT, CT, WT;
}Process;

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

void SJFNonPreemptive(Process p[], int n)
{
   int TotalTAT = 0, TotalWT = 0, time = 0, completed = 0, i;
   int remainingBT[n];
   for(i = 0; i < n; i++)
   {
      remainingBT[i] = p[i].BT;
   }

   while(completed < n)
   {
      int minBTIndex = -1;
      int minBT = 9999;
      for(i = 0; i < n; i++)
      {
         if(p[i].AT <= time && remainingBT[i] > 0 && remainingBT[i] < minBT)
         {
            minBT = remainingBT[i];
```

```
            minBTIndex = i;
        }
    }

    if(minBTIndex != -1)
    {
        time += p[minBTIndex].BT;
        p[minBTIndex].CT = time;
        p[minBTIndex].TAT = p[minBTIndex].CT - p[minBTIndex].AT;
        p[minBTIndex].WT = p[minBTIndex].TAT - p[minBTIndex].BT;
        TotalTAT += p[minBTIndex].TAT;
        TotalWT += p[minBTIndex].WT;
        remainingBT[minBTIndex] = 0;
        completed++;
    }
    else
    {
        time++;
    }
}

float avgTAT = (float)TotalTAT / n;
float avgWT = (float)TotalWT / n;
printf("Average TurnAround Time: %f\n", avgTAT);
printf("Average Waiting Time: %f\n", avgWT);
}

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

for(i = 0; i < n; i++)
{
    p[i].id = i+1;
    printf("Enter arrival time and burst time for process %d: ", i+1);
    scanf("%d %d", &p[i].AT, &p[i].BT);
}
```

```
    sorted(p, n);
    SJFNonPreemptive(p, n);

    return 0;
}
```

Output:

```
Enter the number of processes: 4
Enter arrival time and burst time for process 1: 0 7
Enter arrival time and burst time for process 2: 8 3
Enter arrival time and burst time for process 3: 3 4
Enter arrival time and burst time for process 4: 5 6
Average TurnAround Time: 9.000000
Average Waiting Time: 4.000000

Process returned 0 (0x0)   execution time : 41.377 s
Press any key to continue.
```

b) i) Non preemptive SJF

```
#include <stdio.h>
typedef struct {
    int id, AT, BT, TAT, RT, CT, WT;
} Process;
void sorted (Process p[], int n)
    {
        int i, j;
        for (i=0; i<n-1; i++)
            {
                for (j=0; j<n-i-1; j++)
                    {
                        if (p[j].AT > p[j+1].AT)
```

```
            {
                Process temp = p[j];
                p[j] = p[j+1];
                p[j+1] = temp;
            }
        }
    }
}
void SJFNonPreemptive (Process p[], int n)
    {
    int totalTAT = 0, totalWT = 0, time = 0,
        completed = 0, i;
    int remainingBT[n];
    for (i=0; i<n; i++)
        {
            remainingBT[i] = p[i].BT;
        }
    while (completed < n)
        {
            int minBTIndex = -1;
            int minBT = 9999;
            for (i=0; i<n; i++)
                {
                if (p[i].AT <= time && remainingBT[i]>0
                        && remainingBT[i] < minBT)
                    {
                        minBT = remainingBT[i];
                        minBTIndex = i;
                    }
                }
```

```
if (minBTIndex != -1)
  {
    time += p[minBTIndex].BT;
    p[minBTIndex].CT = time;
    p[minBTIndex].TAT = p[minBTIndex].CT -
                        p[minBTIndex].AT;

    p[minBTIndex].WT = p[minBTIndex].TAT -
                       p[minBTIndex].BT;

    totalTAT += p[minBTIndex].TAT;
    totalWT += p[minBTIndex].WT;
    remainingBT[minBTIndex] = 0;
    completed ++;
  }
  else
    {
      time++;
    }
  }
float avgTAT = (float) totalTAT/n;
float avgWT = (float) totalWT/n;
printf ("Average TurnAround Time : %f\n", avgTAT);
printf ("Average Waiting Time : %f\n", avgWT);
}
int main () {
    int n, i;
    printf ("Enter the number of processes : ");
    scanf ("%d", &n);
    Process p[n];
```

```
for (i=0; i<n; i++)
   {
    p[i].id = i+1;
    printf ("Enter arrival time and burst
      time for process %d: ", i+1);
    scanf ("%d %d", &p[i].AT, &p[i].BT);
   }
  sorted (p,n);
  SJF NonPreemptive (p,n);

  return 0;
}
```

Output

```
Enter the number of processes : 4
Enter arrival time and burst time for
  process 1 : 0 7
Enter arrival time and burst time for
  process 2 : 8 3
Enter arrival time and burst time for
  process 3 : 3 4
Enter arrival time and burst time for
  process 4 : 5 6
Average TurnAround Time : 9.00
Average Waiting Time : 4.00
```

```
#include<stdio.h>
typedef struct{
   int id, AT, BT, TAT, RT, CT, WT, tempBT;
}Process;

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

void SJFPreemptive(Process p[], int n)
{
   int TotalTAT = 0, TotalWT = 0, time = 0, completed = 0, i, minBTIndex;
   int remainingBT[n];
   for(i = 0; i < n; i++)
   {
      remainingBT[i] = p[i].BT;
   }

   while(completed < n)
   {
      minBTIndex = -1;
      int minBT = 9999;
      for(i = 0; i < n; i++)
      {
         if(p[i].AT <= time && remainingBT[i] > 0 && remainingBT[i] < minBT)
         {
            minBT = remainingBT[i];
            minBTIndex = i;
         }
      }

      if(minBTIndex != -1)
      {
         remainingBT[minBTIndex]--;
         time++;
         if(remainingBT[minBTIndex] == 0)
```

```
            {
                completed++;
                p[minBTIndex].CT = time;
                p[minBTIndex].TAT = p[minBTIndex].CT - p[minBTIndex].AT;
                p[minBTIndex].WT = p[minBTIndex].TAT - p[minBTIndex].BT;
                TotalTAT += p[minBTIndex].TAT;
                TotalWT += p[minBTIndex].WT;
            }
        }
        else
        {
            time++;
        }
    }

    float avgTAT = (float)TotalTAT / n;
    float avgWT = (float)TotalWT / n;
    printf("Average TurnAround Time: %f\n", avgTAT);
    printf("Average Waiting Time: %f\n", avgWT);
}

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

    for(i = 0; i < n; i++)
    {
        p[i].id = i+1;
        printf("Enter arrival time and burst time for process %d: ", i+1);
        scanf("%d %d", &p[i].AT, &p[i].BT);
    }

    sorted(p, n);
    SJFPreemptive(p, n);

    return 0;
}
```

Output:

```
Enter the number of processes: 4
Enter arrival time and burst time for process 1: 0 8
Enter arrival time and burst time for process 2: 1 4
Enter arrival time and burst time for process 3: 2 9
Enter arrival time and burst time for process 4: 3 5
Average TurnAround Time: 13.000000
Average Waiting Time: 6.500000
```

ii) SJF Preemptive (SRTF)

```
#include <stdio.h>
typedef struct {
    int id, AT, BT, CT, TAT, RT, WT, tempBT;
} Process;
void sorted (Process p[], int n)
{
    int i, j;
    for (i=0; i<n-1; i++)
    {
        for (j=0; j<n-i-1; j++)
        {
            if (p[j].AT > p[j+1].AT)
            {
                Process temp = p[j];
                p[j] = p[j+1];
                p[j+1] = temp;
            }
        }
    }
}
void SJFPreemptive (Process p[], int n)
{
    int TotalTAT = 0, TotalWT = 0, time = 0,
        completed = 0, i, minBTIndex;
    int remainingBT[n];
    for (i=0; i<n; i++)
    {
        remainingBT[i] = p[i].BT;
    }
```

```
while (completed < n)
  {
    minBTIndex = -1;
    int minBT = 9999;
    for (i=0; i<n; i++)
      {
      if (p[i].AT <= time && remainingBT[i]>0
          && remainingBT[i] < minBT)
      {
        minBT = remainingBT[i];
        minBTIndex = i;
      }
    }
  if (minBTIndex != -1)
    {
      remainingBT[minBTIndex]--;
      time++;
      if (remainingBT[minBTIndex] == 0)
      {
        completed++;
        p[minBTIndex].CT = time;
        p[minBTIndex].TAT = p[minBTIndex].CT -
                            p[minBTIndex].AT
        p[minBTIndex].WT = p[minBTIndex].TAT -
                           p[minBTIndex].BT
      TotalTAT += p[minBTIndex].TAT;
      TotalWT += p[minBTIndex].WT;
    }
  }
```

```
    else {
        time++;
    }
}
float avgTAT = (float) TotalTAT / n;
float avgWT = (float) TotalWT / n;
printf("Average TurnAround Time : %f\n", avgTAT);
printf("Average Waiting Time : %f\n", avgWT);
}
int main()
{
    int n, i;
    printf("Enter the number of processes : ");
    scanf("%d", &n);
    Process p[n];
    for(i=0; i<n; i++)
    {
        p[i].id = i+1;
        printf("Enter arrival time and burst
time for process %d :", i+1);
        scanf("%d %d", &p[i].AT, &p[i].BT);
    }
sorted(p, n);
SJFPreemptive(p, n);
return 0;
}
```

Output

```
Enter the number of processes : 4
Enter arrival time and burst time for
process 1 :  0  8
Enter arrival time and burst time for
process 2 :  1  4
Enter arrival time and burst time for
process 3 :  2  9
Enter arrival time and burst time for
process 4 :  3  5

Average TurnAround Time : 13.00
Average Waiting Time : 6.50
```

Lab 2 :

Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time.

(a) Priority Scheduling (Non Preemptive)

(b) Priority Scheduling (Preemptive)

```
#include <stdio.h>

struct Process {
   int id, arrival_time, burst_time, priority;
   int completion_time, turnaround_time, waiting_time;
   int is_completed;
};

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

   struct Process p[n];
   for (int i = 0; i < n; i++) {
      p[i].id = i + 1;
      printf("\nEnter Arrival Time, Burst Time and Priority for Process %d: ", p[i].id);
      scanf("%d %d %d", &p[i].arrival_time, &p[i].burst_time, &p[i].priority);
      p[i].is_completed = 0;
   }

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

   while (completed < n) {
      int selected = -1;
      int min_priority = 9999;

      for (int i = 0; i < n; i++) {
         if (p[i].arrival_time <= current_time && !p[i].is_completed) {
            if (p[i].priority < min_priority ||
              (p[i].priority == min_priority && p[i].arrival_time < p[selected].arrival_time)) {
               min_priority = p[i].priority;
               selected = i;
```

```
            }
        }
    }

    if (selected == -1) {
        current_time++; // No process is available now, so CPU is idle
    } else {
        p[selected].completion_time = current_time + p[selected].burst_time;
        p[selected].turnaround_time = p[selected].completion_time - p[selected].arrival_time;
        p[selected].waiting_time = p[selected].turnaround_time - p[selected].burst_time;
        p[selected].is_completed = 1;
        current_time = p[selected].completion_time;
        completed++;

        total_tat += p[selected].turnaround_time;
        total_wt += p[selected].waiting_time;
    }
}

// Display Results
printf("\nProcess\tAT\tBT\tPriority\tCT\tTAT\tWT\n");
for (int i = 0; i < n; i++) {
    printf("P%d\t%d\t%d\t%d\t\t%d\t%d\t%d\n",
           p[i].id, p[i].arrival_time, p[i].burst_time,
           p[i].priority, p[i].completion_time,
           p[i].turnaround_time, p[i].waiting_time);
}

printf("\nAverage Turnaround Time: %.2f", total_tat / n);
printf("\nAverage Waiting Time: %.2f\n", total_wt / n);

return 0;
}
```

Output:

```
Enter the number of processes: 4

Enter Arrival Time, Burst Time and Priority for Process 1: 0 5 4

Enter Arrival Time, Burst Time and Priority for Process 2: 2 4 2

Enter Arrival Time, Burst Time and Priority for Process 3: 2 2 6

Enter Arrival Time, Burst Time and Priority for Process 4: 4 4 3

Process AT      BT      Priority        CT      TAT     WT
P1      0       5       4               5       5       0
P2      2       4       2               9       7       3
P3      2       2       6               15      13      11
P4      4       4       3               13      9       5

Average Turnaround Time: 8.50
Average Waiting Time: 4.75
```

Write C programs to demonstrate
a) Non Preemptive Priority Scheduling
b) Preemptive Priority Scheduling.

a)
```
#include <stdio.h>
struct Process {
    int id, arrival_time, burst_time, priority;
    int completion_time, turnaround_time, waiting_time;
    int is_completed;
};
int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process p[n];
    for (int i=0; i<n; i++) {
        p[i].id = i+1;
        printf("\n Enter Arrival Time, Burst Time
          and Priority for Process %d: ", p[i].id);
        scanf("%d %d %d", &p[i].arrival_time,
          &p[i].burst_time, &p[i].priority);
        p[i].is_completed = 0;
    }
    int current_time = 0, completed = 0;
    float total_tat = 0, total_wt = 0;
    while (completed < n) {
        int selected = -1;
        int min_priority = 9999;
```

```
for (int i=0; i<n; i++) {
  if (p[i].arrival_time <= current_time && !p[i].is_completed) {
    if (p[i].priority < min_priority ||
      (p[i].priority == min_priority &&
      p[i].arrival_time < p[selected].arrival_time))
    {
      min_priority = p[i].priority;
      selected = i;
    }
  }
}
if (selected == -1) {
  current_time++;
} else {
p[selected].completion_time = current_time +
                    p[selected].burst_time;
p[selected].turnaround_time = p[selected].
                    completion_time - p[selected].
                    arrival_time;
p[selected].waiting_time = p[selected].turnaround_ti
                    - p[selected].burst_time;
p[selected].is_completed = 1;
current_time = p[selected].completion_time;
completed++;
```

```
        total_tat += p[selected].turnaround_time;
        total_wt += p[selected].waiting_time;

    }
}
printf("\n Process\t AT\t BT\t Priority\t CT\t TAT
        \t WT\n");

for(int i=0; i<n; i++) {
    printf("%d\t %d\t %d\t %d\t %d\t %d\t %d\t %d
        \n", p[i].id, p[i].arrival_time,
    p[i].burst_time, p[i].priority, p[i].completion
    time, p[i].turnaround_time,
    p[i].waiting_time);

    }
printf("\n Average Turnaround Time : %.2f",
                                total_tat/n);

printf("\n Average Waiting Time : %.2f",
                                total_wt/n);

return 0;
}
```

Output

```
Enter the number of processes : 4
Enter Arrival Time, Burst Time and Priority for
 process 1 : 0  5  4
Enter Arrival Time, Burst Time and Priority for
process 2 : 2  4  2
Enter Arrival Time, Burst Time and Priority for
process 3 : 2  2  6
```

Enter Arrival Time, Burst Time and Priority for Process 4 : 4 4 3

Process	AT	BT	Priority	CT	TAT	WT
P1	0	5	4	5	5	0
P2	2	4	2	9	7	3
P3	2	2	6	15	13	11
P4	4	4	3	13	9	5

Average Turnaround Time : 8.50
Average Waiting Time : 4.75

```
#include <stdio.h>

struct Process {
    int id, arrival_time, burst_time, priority;
    int remaining_bt, completion_time, turnaround_time, waiting_time, response_time;
    int is_completed, is_started;
};

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

    struct Process p[n];
    for (int i = 0; i < n; i++) {
        p[i].id = i + 1;
        printf("\nEnter Arrival Time, Burst Time and Priority for Process %d: ", p[i].id);
        scanf("%d %d %d", &p[i].arrival_time, &p[i].burst_time, &p[i].priority);
        p[i].remaining_bt = p[i].burst_time;
        p[i].is_completed = 0;
        p[i].is_started = 0;
    }

    int current_time = 0, completed = 0;
    float total_tat = 0, total_wt = 0, total_rt = 0;

    while (completed < n) {
        int selected = -1;
        int min_priority = 9999;

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

```
        if (p[i].arrival_time <= current_time && p[i].remaining_bt > 0) {
            if (p[i].priority < min_priority ||
               (p[i].priority == min_priority && p[i].arrival_time < p[selected].arrival_time)) {
                min_priority = p[i].priority;
                selected = i;
            }
        }
    }

    if (selected == -1) {
        current_time++;
    } else {
        if (p[selected].is_started == 0) {
            p[selected].response_time = current_time - p[selected].arrival_time;
            p[selected].is_started = 1;
        }

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

        if (p[selected].remaining_bt == 0) {
            p[selected].completion_time = current_time;
            p[selected].turnaround_time = p[selected].completion_time - p[selected].arrival_time;
            p[selected].waiting_time = p[selected].turnaround_time - p[selected].burst_time;
            total_tat += p[selected].turnaround_time;
            total_wt += p[selected].waiting_time;
            total_rt += p[selected].response_time;
            p[selected].is_completed = 1;
            completed++;
        }
    }
}

// Display Results
printf("\nProcess\tAT\tBT\tPri\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",
           p[i].id, p[i].arrival_time, p[i].burst_time, p[i].priority,
           p[i].completion_time, p[i].turnaround_time,
           p[i].waiting_time, p[i].response_time);
}

printf("\nAverage Turnaround Time: %.2f", total_tat / n);
printf("\nAverage Waiting Time: %.2f", total_wt / n);
printf("\nAverage Response Time: %.2f\n", total_rt / n);

return 0;
}
```

Output :

```
Enter the number of processes: 4

Enter Arrival Time, Burst Time and Priority for Process 1: 0 5 4

Enter Arrival Time, Burst Time and Priority for Process 2: 2 4 2

Enter Arrival Time, Burst Time and Priority for Process 3: 2 2 6

Enter Arrival Time, Burst Time and Priority for Process 4: 4 4 3

Process AT      BT      Pri     CT      TAT     WT      RT
P1      0       5       4       13      13      8       0
P2      2       4       2       6       4       0       0
P3      2       2       6       15      13      11      11
P4      4       4       3       10      6       2       2

Average Turnaround Time: 9.00
Average Waiting Time: 5.25
Average Response Time: 3.25
```

b)

```
#include <stdio.h>
struct Process {
    int id, arrival_time, burst_time, priority;
    int remaining_bt, completion_time,
    turnaround_time, waiting_time, response_time;
    int is_completed, is_started;
};
int main() {
    int n;
    printf("Enter the number of processes : ");
    scanf("%d",&n);

struct Process p[n];
for(int i=0; i<n; i++) {
    p[i].id = i+1;
    printf("\n Enter Arrival time, Burst Time
and Priority for Process %d :", p[i].id);
    scanf("%d %d %d", &p[i].arrival_time,
    &p[i].burst_time, &p[i].priority);
```

```
    p[i].remaining_bt = p[i].burst_time;
    p[i].is_completed = 0;
    p[i].is_started = 0;
}
int current_time = 0, completed = 0;
float total_tat = 0, total_wt = 0, total_rt = 0;

while (completed < n) {
    int selected = -1;
    int min_priority = 9999;
    for (int i=0; i<n; i++){
        if (p[i].arrival_time <= current_time &&
            p[i].remaining_bt > 0) {
            if (p[i].priority < min_priority ||
                (p[i].priority == min_priority &&
                 p[i].arrival_time < p[selected].arrival_time))
            {
                min_priority = p[i].priority;
                selected = i;
            }
        }
    }
}
if (selected == -1){
    current_time ++;
} else {
    if (p[selected].is_started == 0) {
        p[selected].response_time = current_time -
                                    p[selected].arrival_time;
        p[selected].is_started = 1;
```

```
        p[selected].remaining_bt--;
        current_time++;
        if (p[selected].remaining_bt == 0) {
            p[selected].completion_time = current_time;
            p[selected].turnaround_time = p[selected].
                completion_time - p[selected].arrival_time;
            p[selected].waiting_time = p[selected].
                turnaround_time - p[selected].burst_time;

            total_tat += p[selected].turnaround_time;
            total_wt += p[selected].waiting_time;
            total_rt += p[selected].response_time;
            p[selected].is_completed = 1;
            completed++;
        }
    }
}
printf("\n Process\t AT\t BT\t Pri\t CT\t TAT\t WT
    \t RT \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",
    p[i].id, p[i].arrival_time, p[i].burst_time,
    p[i].priority, p[i].completion_time,
    p[i].turnaround_time, p[i].waiting_time,
    p[i].response_time); }
```

```
    printf("\n Average Turnaround Time : %.2f",
                                    total_tat/n);
    printf("\n Average Waiting Time : %.2f",
                                    total_wt/n);
    printf("\n Average Waiting Time : %.2f",
                                    total_rt/n);

    return 0;
}
```

Output

Enter the number of processes : 4

Enter Arrival Time, Burst Time and Priority for Process 1 : 0 5 4

Enter Arrival Time, Burst Time and Priority for Process 2 : 2 4 2

Enter Arrival Time, Burst Time and Priority for Process 3 : 2 2 6

Enter Arrival Time, Burst Time and Priority for Process 4 : 4 4 3

Process	AT	BT	Pri	CT	TAT	WT	RT
P1	0	5	4	13	13	8	0
P2	2	4	2	6	4	0	0
P3	2	2	6	15	13	11	11
P4	4	4	3	10	6	2	2

Average Turnaround Time : 9.00
Average Waiting Time : 5.25
Average Response Time : 3.25

Lab 3:

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

```
#include <stdio.h>


#define MAX_PROCESSES 10
#define TIME_QUANTUM 2


typedef struct {
    int id;
    int burst_time;
    int arrival_time;
    int queue; // 1 for system process (RR), 2 for user process (FCFS)
    int remaining_time;
    int waiting_time;
    int turnaround_time;
    int response_time;
} Process;


void round_robin(Process processes[], int n, int tq, int *time) {
    int done, i;
    do {
        done = 1;
        for (i = 0; i < n; i++) {
            if (processes[i].queue == 1 && processes[i].remaining_time > 0) {
                done = 0;
                if (processes[i].remaining_time == processes[i].burst_time)
                    processes[i].response_time = *time - processes[i].arrival_time;
                if (processes[i].remaining_time > tq) {
                    *time += tq;
```

```
                processes[i].remaining_time -= tq;
            } else {
                *time += processes[i].remaining_time;
                processes[i].waiting_time = *time - processes[i].burst_time - processes[i].arrival_time;
                processes[i].turnaround_time = *time - processes[i].arrival_time;
                processes[i].remaining_time = 0;
            }
        }
    }
  } while (!done);
}

void fcfs(Process processes[], int n, int *time) {
  for (int i = 0; i < n; i++) {
    if (processes[i].queue == 2) {
      if (*time < processes[i].arrival_time)
        *time = processes[i].arrival_time;
      processes[i].response_time = *time - processes[i].arrival_time;
      processes[i].waiting_time = *time - processes[i].arrival_time;
      *time += processes[i].burst_time;
      processes[i].turnaround_time = *time - processes[i].arrival_time;
    }
  }
}

void calculate_average(Process processes[], int n) {
  float total_waiting = 0, total_turnaround = 0, total_response = 0;
  for (int i = 0; i < n; i++) {
    total_waiting += processes[i].waiting_time;
    total_turnaround += processes[i].turnaround_time;
    total_response += processes[i].response_time;
  }
```

```
    printf("\nAverage Waiting Time: %.2f", total_waiting / n);
    printf("\nAverage Turn Around Time: %.2f", total_turnaround / n);
    printf("\nAverage Response Time: %.2f", total_response / n);
    printf("\nThroughput: %.2f\n", n / (total_turnaround / n));
}

int main() {
    int n, time = 0;
    Process processes[MAX_PROCESSES];

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

    printf("Queue 1 is system process\nQueue 2 is User Process\n");

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

    round_robin(processes, n, TIME_QUANTUM, &time);
    fcfs(processes, n, &time);

    printf("\nProcess\tWaiting Time\tTurn Around Time\tResponse Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t\t%d\n", processes[i].id, processes[i].waiting_time,
processes[i].turnaround_time, processes[i].response_time);
```

```
    }

    calculate_average(processes, n);

    return 0;
}
```

Output:

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

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

Average Waiting Time: 4.25
Average Turn Around Time: 7.00
Average Response Time: 4.25
Throughput: 0.57
```

Q Write a C program to demonstrate multilevel queue scheduling.

```
#include <stdio.h>
#define MAX_PROCESSES 10
#define TIME_QUANTUM 2

typedef struct {
    int id;
    int burst_time;
    int arrival_time;
    int queue;
    int remaining_time;
    int waiting_time;
    int turnaround_time;
    int response_time;
} Process;

void round_robin(Process processes[], int n,
                 int tq, int *time) {
    int done, i;
    do {
        done = 1;
        for (i=0; i<n; i++) {
            if (processes[i].queue == 1 &&
                processes[i].remaining_time > 0) {
                done = 0;
                if (processes[i].remaining_time == processes[i].
                    burst_time)
                    processes[i].response_time = *time -
                        processes[i].burst_time -
                        processes[i].arrival_time;
```

```
if ( processes[i].remaining_time > tq) {
        *time += tq;
        processes[i].remaining_time -= tq;
    } else {
    *time += processes[i].remaining_time;
    processes[i].waiting_time = *time -
                        processes[i].burst_time
                    - processes[i].arrival_time;
    processes[i].turnaround_time = *time -
                        processes[i].arrival_time;

    processes[i].remaining_time = 0;
    }
  }
 }
} while(!done);
}
void fcfs (Process processes[], int n, int *time) {
    for (int i=0; i<n; i++) {
      if (processes[i].queue == 2) {
        if (*time < processes[i].arrival_time)
          *time = processes[i].arrival_time;
    processes[i].response_time = *time -
                        processes[i].arrival_time;
    processes[i].waiting_time = *time -
                        processes[i].arrival_time;
     *time += processes[i].burst_time;
```

```
            processes[i].turnaround_time = * time -
                                processes[i].arival_time ;
        }
    }
}
void calculate_average (Process processes[], int n){
    float total_waiting = 0 ; total_ turnaround = 0,
        total_ response = 0 ;
    for ( int i=0; i<n ; i++) {
        total_waiting += processes[i]. waiting_time;
        total_turnaround += processes[i]. turnaround_time;
        total_response += processes[i]. response_time ;
    }
    printf ("\n Average Waiting Time : %.2f ",
                                total_waiting / n) ;
    printf ("\n Average Turnaround Time : %.2f",
                                total_turnaround / n) ;
    printf ("\n Average Response Time : %.2f ",
                                total_ response / n) ;
    printf ("\n Throughput : %.2f\n",
                                (total_turnaround / n)) ;
}
int main() {
    int n, time = 0;
    process processes [MAX_PROCESSES] ;
    printf (" Enter number of processes : ");
    scanf ("%d", &n);
```

```
printf ("Queue 1 is system process \n Queue 2
        is user process \n");
for (int i=0 ; i<n; i++) {
printf ("Enter Burst Time, Arrival Time and
        Queue of P%d ", i+1);
scanf ("%d %d %d", &processes[i].burst_time,
        &processes[i].arrival_time,
        &processes[i].queue);
processes[i].id = i+1;
processes[i].remaining_time = processes[i].
                              burst_time;
processes[i].waiting_time = 0;
processes[i].turnaround_time = 0;
processes[i].response_time = 0;
}
round_robin (processes, n, TIME_QUANTUM, &time);
fcfs (processes, n, &time);
printf ("\n Process \t Waiting Time \t Turn Around
    Time \t Response Time \n");
for (int i= 0; i<n; i++) {
  printf ("%d \t %d \t %d \t \t \t %d \n",
  processes[i].pid, processes[i].waiting_time,
  processes[i].turnaround_time, process[i].
  response_time);
}
calculate_average (processes,n);
return 0;
}
```

Output

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

Process	Waiting Time	TurnAround Time	Response Tim
1	0	2	0
2	7	8	7
3	2	7	2
4	8	11	8

```
Average Waiting Time : 4.25
Average TurnAround Time : 7.00
Average Response Time : 4.25
Throughput : 0.57
```

b.) Rate- Monotonic

```
#include <stdio.h>

#define MAX_PROCESSES 10

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

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

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

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

```
}

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

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

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

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

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

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

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

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

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

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

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

Output:

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

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

4.100000 <= 0.779763 => false

System may not be schedulable!

Process returned 0 (0x0)   execution time : 49.003 s
Press any key to continue.
```

Q write C program to demonstrate rate monotic.

```
#include <stdio.h>
#define MAX_PROCESSES 10
typedef struct {
    int id;
    int burst_time;
    int period;
    int remaining_time;
    int next_deadline;
} Process;
void sort_by_period (Process processes [], int n) {
  for (int i=0; i<n-1; i++) {
    for (int j=0; j<n-i-1; j++) {
      if (processes[j].period > processes[j+1].period)
        {
          Process temp = processes[j];
          processes[j] = processes[j+1];
          processes[j+1] = temp;
        }
    }
  }
}
int gcd (int a, int b) {
    return b==0 ? a : gcd (b, a%b);
}
int lcm (int a, int b) {
    return (a*b) / gcd (a,b);
}
```

```
int calculate_lcm (Process processes[], int n);
    int result = processes[0].period;
    for(int i=1; i<n; i++) {
        result = lcm ( result, processes[i]. period
    }
    return result;
}
double utilization_factor (Process processes [],
                           int n) {
    double sum = 0;
    for (int i=0; i<n; i++) {
        sum += (double) processes[i] . burst_time
                processes[i]. period ;
    }
    return sum;
}
double rms_threshold ( int n) {
    return n* (pow (2.0, 1.0/n) -1) ;
}
void rate_monotonic_scheduling (Process processes[],
                                int n) {
    int lcm_period = calculate_lcm (processes, n);
    printf ("LCM = %d\n\n", lcm_period);
    printf (" Rate Monotone Scheduling :\n");
    printf (" PID Burst Period \n");
    for (int i=0; i<n; i++) {
        printf ("%d   %d   %d \n)", processes[i].id,
        processes[i]. burst_time, processes[i]. period);
    }
```

```
double utilization = utilization_factor(processes, n);
double threshold = rms_threshold(n);
printf("\n%.6f <= %.6f => %s\n",
       utilisation, threshold, (utilisation <= threshold)

if (utilisation > threshold) {
    printf("\n System may not be schedulable !\n");
    return;
}
int timeline = 0, executed = 0;
while (timeline < lcm_period) {
        int selected = -1;
        for(int i=0; i<n; i++){
            if (timeline % processes[i].period==0)
            {
            processes[i].remaining_time =
                    processes[i].burst_time;
            }
        if (processes[i].remaining_time > 0) {
            selected = i;
            break;
        }
    }
if (selected != -1) {
    printf("Time %d: Process %d is running\n",
           timeline, processes[selected].id);
    processes[selected].remaining_time--;
    executed++;
} else {
    printf("Time %d: CPU is idle\n", timeline);
}
```

```
        timeline++;
    }
}

int main() {
    int n;
    Process processes[MAX_PROCESSES];
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    printf("Enter the CPU burst times:\n");
    for (int i=0; i<n; i++) {
        processes[i].id = i+1;
        scanf("%d", &processes[i].burst_time);
        processes[i].remaining_time =
                    processes[i].burst_time;
    }
    printf("Enter the time periods:\n");
    for (int i=0; i<n; i++) {
        scanf("%d", &processes[i].period);
    }
    sort_by_period(processes, n);
    rate_monotonic_scheduling(processes, n);

    return 0;
}
```

Output

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

PID	Burst	period
1	3	3
2	6	4
3	8	5

4.100000 ⇐ 0.779763 ⇒ false
System may not be schedulable !

c.) Earliest-deadline First

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

typedef struct {
    int pid;
    int burst;
    int deadline;
    int period;
} Process;

void sortByDeadline(Process proc[], int n) {
    for (int i = 0; i < n - 1; i++) {
```

```
        for (int j = 0; j < n - i - 1; j++) {
            if (proc[j].deadline > proc[j + 1].deadline) {
                Process temp = proc[j];
                proc[j] = proc[j + 1];
                proc[j + 1] = temp;
            }
        }
    }
}

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

    Process proc[n];

    printf("\nEnter the CPU burst times:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &proc[i].burst);
        proc[i].pid = i + 1;
    }

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

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

```
    sortByDeadline(proc, n);

    printf("\nEarliest Deadline Scheduling:\n");
    printf("PID    Burst    Deadline    Period\n");
    for (int i = 0; i < n; i++) {
        printf("%d      %d        %d          %d\n", proc[i].pid, proc[i].burst, proc[i].deadline, proc[i].period);
    }
    printf("\nScheduling occurs for 6 ms\n");
    for (int time = 0; time < 6; time++) {
        printf("%dms : Task %d is running.\n", time, proc[0].pid);
    }
}
```

Output:

```
Enter the number of processes:3

Enter the CPU burst times:
2 3 5

Enter the deadlines:
1 2 3

Enter the time periods:
1 2 3

Earliest Deadline Scheduling:
PID     Burst     Deadline     Period
1       2         1            1
2       3         2            2
3       5         3            3

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

1 Write a C program to demonstrate Earliest deadline first (EDF)

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
    int pid;
    int burst;
    int deadline;
    int period;
} Process;
void sortByDeadline (Process proc[], int n) {
    for (int i=0; i<n-1; i++) {
        for (int j=0; j<n-i-1; j++) {
            if (proc[j].deadline > proc[j+1].
                                    deadline) {
                Process temp = proc[j];
                proc[j] = proc[j+1];
                proc[j+1] = temp;
            }
        }
    }
}
void main () {
    int n;
    printf ("Enter the number of processes : ");
    scanf ("%d", &n);
    Process proc[n];
    printf ("\nEnter the CPU burst times :\n");
    for (int i=0; i<n; i++) {
        scanf ("%d", &proc[i].burst);
        proc[i].pid = i+1;
    }
```

```
printf("Enter the deadlines :\n");
for (int i=0; i<n; i++) {
    scanf("%d", &proc[i].period);
}
sortByDeadline("\nEnter the deadlines :\n");
for (int i=0; i<n; i++) {
    scanf("%d", &proc[i].deadline);
}
printf("\nEnter the time periods :\n");
for (int i=0; i<n; i++) {
    scanf("%d", &proc[i].period);
}
sortByDeadline(proc, n);
printf("\n Earliest Deadline Scheduling :\n");
printf(" PID  Burst  Deadline  Period\n");
for (int i=0; i<n; i++) {
printf("%d  %d  %d  %d\n",
proc[i].pid, proc[i].burst, proc[i].deadline,
proc[i].period);
}
printf("\n scheduling occurs for 6 ms\n");
for (int time = 0; time < 6 ; time++) {
    printf("%dms : Task %d is running.\n",
        time, proc[0].pid);
}
}
```

Output

Enter the number of processes : 3

Enter the CPU burst times :

2 3 5

Enter the deadlines :

1 2 3

Enter the time periods :

1 2 3

Earliest Deadline Scheduling :

PID	Burst	Deadline	Period
1	2	1	1
2	3	2	2
3	5	3	3

Scheduling occurs for 6 ms

0ms : Task 1 is running

1ms : Task 1 is running

2ms : Task 1 is running

3ms : Task 1 is running

4ms : Task 1 is running

5ms : Task 1 is running.

Lab 4:

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

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

int mutex = 1, full = 0, empty = 3, x = 0, buffer = 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 = rand() % 50;
   buffer = x;
   printf("Producer 1 produced %d\n", x);
   printf("Buffer:%d\n", buffer);
   mutex = signal(mutex);
}

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

```
    empty = signal(empty);
    printf("Consumer 2 consumed %d\n", buffer);
    printf("Current buffer len: 0\n");
    x--;
    mutex = signal(mutex);
}

void main() {
    int choice, p, c;

    srand(time(0));
    printf("Enter the number of Producers:");
    scanf("%d", &p);
    printf("Enter the number of Consumers:");
    scanf("%d", &c);
    printf("Enter buffer capacity:");
    scanf("%d", &empty);

    for (int i = 1; i <= p; i++)
        printf("Successfully created producer %d\n", i);
    for (int i = 1; i <= c; i++)
        printf("Successfully created consumer %d\n", i);

    while (1) {
        printf("\n1.Producer\n2.Consumer\n3.Exit\n");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                if ((mutex == 1) && (empty != 0))
                    producer();
                else
                    printf("Buffer is full\n");
```

```
            break;
        case 2:
            if ((mutex == 1) && (full != 0))
                consumer();
            else
                printf("Buffer is empty\n");
            break;
        case 3:
            exit(0);
    }
  }
}
```

Output:

```
Enter the number of Producers:1
Enter the number of Consumers:1
Enter buffer capacity:1
Successfully created producer 1
Successfully created consumer 1

1.Producer
2.Consumer
3.Exit
1
Producer 1 produced 10
Buffer:10

1.Producer
2.Consumer
3.Exit
2
Consumer 2 consumed 10
Current buffer len: 0

1.Producer
2.Consumer
3.Exit
1
Producer 1 produced 12
Buffer:12

1.Producer
2.Consumer
3.Exit
2
Consumer 2 consumed 12
Current buffer len: 0

1.Producer
2.Consumer
3.Exit
```

9 Write a c program to demonstrate producer consumer problem.

```
# include <stdio.h>
# include <stdlib.h>
# include <time.h>
int mutex = 1, full = 0, empty = 3, x = 0,
    buffer = 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 = rand () % 50;
    buffer = x;
    printf ("Producer 1 produced %d\n", x);
    printf ("Buffer %d\n", buffer);
    mutex = signal (mutex);
}
void consumer () {
    int choice, p, c;
    srand (time (0));
    printf ("Enter the number of Producers : ");
    scanf ("%d", &p);
    printf ("Enter the number of consumers : ");
    scanf ("%d", &c);
    printf ("Enter buffer capacity : ");
    scanf ("%d", &empty);
```

```
for (int i=1; i<=p ; i++)
    printf ("Successfully created producer %d\n",i);
for (int i=1; i<=c; i++)
    printf ("successfully created consumer %d\n", i);

while (1) {
  printf ("\n1. Producer \n2. Consumer \n3. Exit\n");
  scanf ("%d", &choice);
  switch (choice) {
      case 1:
          if ((mutex==1) && (empty!=0))
              producer();
          else
              printf ("Buffer is full\n");
          break;
      case 2:
          if ((mutex==1) && (full!=0))
              consumer();
          else
              printf ("Buffer is empty\n");
          break;
      case 3:
          exit(0);
  }
```

Output

```
Enter the number of producers : 1
Enter the number of consumers : 1
Enter buffer capacity : 1
successfully created producer 1
successfully created consumer 1

1. Producer
2. Consumer
3. Exit

1
Producer 1 produced 10
Buffer : 10

1. Producer
2. Consumer
3. Exit
2
Consumer 2 consumed 10
Current buffer len : 10

1. Producer
2. Consumer
3. Exit
1
producer 1 produced 12
Buffer : 12

1. Producer
2. Consumer
3. Exit
2
Consumer 2 consumed 12
Current buffer len : 0

1. Producer
2. Consumer
3. Exit
3.
```

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

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

int n;
sem_t *chopstick;
pthread_t *philosopher;
int *phil_ids;

void* philosopher_fn(void* num) {
    int i = *(int*)num;
    int left = i;
    int right = (i + 1) % n;
    printf("Philosopher %d is hungry\n", i + 1);
    sem_wait(&chopstick[left]);
    sem_wait(&chopstick[right]);
    printf("Philosopher %d took chopstick %d and %d\n", i + 1, left + 1, right + 1);
    printf("Philosopher %d is eating\n", i + 1);
    sleep(1);
    printf("Philosopher %d finished eating and kept the chopstick back on table\n", i + 1);
    sem_post(&chopstick[left]);
    sem_post(&chopstick[right]);
    return NULL;
}

int main() {
    int hungry, i;
    printf("Enter the number of philosophers:");
```

```
    scanf("%d", &n);
    chopstick = malloc(n * sizeof(sem_t));
    philosopher = malloc(n * sizeof(pthread_t));
    phil_ids = malloc(n * sizeof(int));
    for (i = 0; i < n; i++) sem_init(&chopstick[i], 0, 1);
    printf("Enter number of hungry philosophers:");
    scanf("%d", &hungry);
    for (i = 0; i < hungry; i++) {
        phil_ids[i] = i;
        pthread_create(&philosopher[i], NULL, philosopher_fn, &phil_ids[i]);
        sleep(1);
    }
    for (i = 0; i < hungry; i++) {
        pthread_join(philosopher[i], NULL);
    }
    free(chopstick);
    free(philosopher);
    free(phil_ids);
    return 0;
}
```

Output :

```
Enter the number of philosophers:5
Enter number of hungry philosophers:5
Philosopher 1 is hungry
Philosopher 1 took chopstick 1 and 2
Philosopher 1 is eating
Philosopher 1 finished eating and kept the chopstick back on table
Philosopher 2 is hungry
Philosopher 2 took chopstick 2 and 3
Philosopher 2 is eating
Philosopher 2 finished eating and kept the chopstick back on table
Philosopher 3 is hungry
Philosopher 3 took chopstick 3 and 4
Philosopher 3 is eating
Philosopher 3 finished eating and kept the chopstick back on table
Philosopher 4 is hungry
Philosopher 4 took chopstick 4 and 5
Philosopher 4 is eating
Philosopher 5 is hungry
Philosopher 4 finished eating and kept the chopstick back on table
Philosopher 5 took chopstick 5 and 1
Philosopher 5 is eating
Philosopher 5 finished eating and kept the chopstick back on table
```

Q Write a C program to demonstrate dining philosophers problem.

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

int n;
sem_t * chopstick;
pthread_t * philosopher;
int * phil_ids;
void * philosopher_fn (void * num) {
    int i = * (int *) num;
    int left = i;
    int right = (i+1) % n;
    printf ("Philosopher %d is hungry\n", i+1);
    sem_wait (& chopstick [left]);
    sem_wait (& chopstick [right]);
    printf ("Philosopher %d took chopstick %d
        and %d \n", i+1, left+1, right+1);
    printf ("Philosopher %d is eating \n", i+1);
    sleep (1);
    printf ("Philosopher %d finished eating and
    kept the chopsticks back on table \n",
        i+1);
```

```
    sem_post (&chopstick [left]);
    sem_post (&chopstick [right]);
    return NULL;
}
int main () {
    int hungry, i;
    printf ("Enter the number of philosophers:");
    scanf ("%d", &n);
    chopstick = malloc (n * sizeof (sem_t));
    philosopher = malloc (n * sizeof (pthread_t));
    phil_ids = malloc (n * size of (int));
    for (i=0; i<n; i++)
        sem_init (&chopstick [i], 0, 1);
    printf ("Enter number of hungry philosophers:")
    scanf ("%d", & hungry);
    for (i=0; i< hungry; i++) {
        phil_ids[i] = i;
        pthread_create (& philosopher [i],
            NULL, philosopher_fn, &
            phil_ids [i]);

    sleep (1);
}
for (i=0; i < hungry; i++) {
    pthread_join (philosopher [i], NULL);

}
free (chopstick);
free (philosopher);
free (phil_ids);
return 0;

}
```

Output

```
Enter the number of philosophers : 5
Enter the number of hungry philosophers : 5
Philosopher 1 is hungry
Philosopher 1 took chopstick 1 and 2
Philosopher 1 is eating
Philosopher 1 finished eating and kept the
chopstick back on the table.
Philosopher 2 is hungry
Philosopher 2 took chopstick 2 and 3.
Philosopher 2 is eating
Philosopher 2 finished eating and kept the
chopstick back on the table.
Philosopher 3 is hungry
Philosopher 3 took chopstick 3 and 4
Philosopher 3 is eating
Philosopher 3 finished eating and kept the
chopstick back on the table.
Philosopher 4 is hungry
Philosopher 4 took chopstick 4 and 5
Philosopher 4 is eating
Philosopher 4 finished eating and kept the
chopstick back on the table.
Philosopher 5 is hungry
Philosopher 5 took chopstick 5 and 1
Philosopher 5 is eating
Philosopher 5 finished eating and kept the
chopstick back on the table.
```

Lab 5:

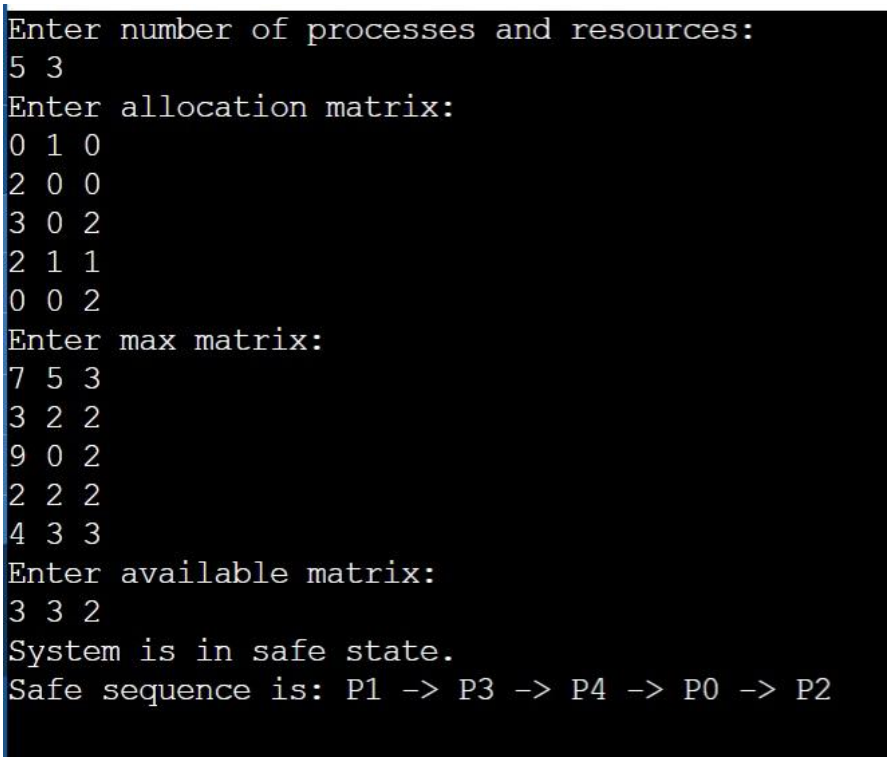
Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

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

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

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

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

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

   int finish[n];
   int safeSeq[n];
   for (int i = 0; i < n; i++) finish[i] = 0;
```

```
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];

int count = 0;
while (count < n) {
    bool found = false;
    for (int i = 0; i < n; i++) {
        if (!finish[i]) {
            int j;
            for (j = 0; j < m; j++)
                if (need[i][j] > avail[j])
                    break;
            if (j == m) {
                for (int k = 0; k < m; k++)
                    avail[k] += alloc[i][k];
                safeSeq[count++] = i;
                finish[i] = 1;
                found = true;
            }
        }
    }
    if (!found) break;
}

if (count == n) {
    printf("System is in safe state.\n");
    printf("Safe sequence is: ");
    for (int i = 0; i < n; i++) {
        printf("P%d", safeSeq[i]);
        if (i != n - 1) printf(" -> ");
```

```
        }
        printf("\n");
    } else {
        printf("System is not in a safe state.\n");
    }

}
```

Output:



1 Write a C program to demonstrate Banker's Algorithm.

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

void main () {
    int n, m;
    printf ("enter number of processes and
            resources : \n");
    scanf ("%d %d", &n, &m);
    int alloc [n][m], max [n][m], avail [m];
    printf ("enter allocation matrix : \n");
    for (int i = 0 ; i < n ; i++)
        for (int j = 0 ; j < m ; j++)
            scanf ("%d", & alloc [i][j]);

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

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

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

    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];
```

```
int count = 0;
while (count < n) {
    bool found = false;
    for (int i=0; i<n; i++) {
        if (!finish[i]) {
            int j;
            for (j=0; j<m; j++)
                if (need[i][j] > avail[j])
                    break;
            if (j==m) {
                for (int k=0; k<m; k++)
                    avail[k] += alloc[i][k];
                safeseq[count++] = i;
                finish[i] = 1;
                found = true;
            }
        }
    }
    if (!found) break;
}
if (count==n) {
    printf("System is in safe state.\n");
    printf("Safe sequence is : ");
    for (int i=0; i<n; i++) {
        printf("P%d", safeseq[i]);
        if (i!=n-1) printf(" -> ");
    }
    printf("\n");
} else {
    printf("System is not in a safe state.\n");
}
}
```

Output

```
Enter number of processes and resources:
5 3
Enter allocation matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter max matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter available matrix:
3 3 2
System is in safe state
Safe sequence is: P1 --> P3 --> P4 --> P0 --> P2
```

b.) Write a C program to simulate deadlock detection

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

int main() {
    int n, m;
    printf("Enter number of processes and resources:\n");
```

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

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

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

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

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

int count = 0;
while (count < n) {
    bool found = false;
    for (int i = 0; i < n; i++) {
        if (!finish[i]) {
            int j;
            for (j = 0; j < m; j++)
                if (request[i][j] > avail[j])
                    break;
            if (j == m) {
                for (int k = 0; k < m; k++)
                    avail[k] += alloc[i][k];
```

```
                finish[i] = 1;
                printf("Process %d can finish.\n", i);
                count++;
                found = true;
            }
        }
    }
    if (!found) break;
}

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

return 0;
}
```

Output:

```
Enter number of processes and resources:
5 3
Enter allocation matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter request matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter available matrix:
3 3 2
Process 1 can finish.
Process 3 can finish.
Process 4 can finish.
System is in a deadlock state.
```

1 write a C program to demonstrate deadlock in operating system.

```
#include <stdio.h>
#include <stdbool.h>
int main () {
    int n, m;
    printf ("Enter number of processes and
            resources :\n");
    scanf ("%d %d", &n, &m);
    int alloc[n][m], request[n][m], avail[m];
    printf ("Enter allocation matrix :\n");
    for (int i=0; i<n; i++)
        for (int j=0; j<m; j++)
            scanf ("%d", &alloc[i][j]);
    printf ("Enter request matrix :\n");
    for (int i=0; i<n; i++)
        for (int j=0; j<m; j++)
            scanf ("%d", &request[i][j]);
    printf ("Enter available matrix:\n");
    for (int i=0; i<m; i++)
        scanf ("%d", &avail[i]);
    int finish[n];
    for (int i=0; i<n; i++)
        finish[i] = 0;
int count = 0;
while (count < n) {
    bool found = false;
    for (int i=0; i<n; i++) {
```

```
if (!finish[i]) {
    int j;
    for (j=0; j<m; j++)
        if (request[i][j] > avail[j])
            break;
    if (j==m) {
        for(int k=0; k<m; k++)
            avail[k] += alloc[i][k];
        finish[i] = 1;
        printf("Process %d can finish\n", i);
        count++;
        found = true;
    }
  }
  if (!found) break;
}
if (count==n)
    printf("System is not in a deadlock state.\n");
else
    printf("System is in deadlock state.\n");

return 0;
}
```

Output

```
Enter number of processes and resources :
5 3
Enter allocation matrix :
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter request matrix :
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter available matrix :
3 3 2
Process 1 can finish
Process 3 can finish
Process 4 can finish
System is in a deadlock state.
```

Lab 6:

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;
};

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

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

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

```
            printf("%d\t%d\t\t%d\t\t%d\t\t%d\n",
                   files[i].file_no,
                   files[i].file_size,
                   blocks[j].block_no,
                   blocks[j].block_size,
                   fragment);

            allocated = 1;
            break;
        }
    }

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

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

  for (int i = 0; i < n_files; i++) {
    int best_fit_block = -1;
    int min_fragment = 10000; // Large initial value

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

```
                best_fit_block = j;
            }
        }
    }

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

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

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

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

```
                max_fragment = fragment;
                worst_fit_block = j;
            }
        }
    }

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

void resetBlocks(struct Block blocks[], struct Block original[], int n_blocks) {
  for (int i = 0; i < n_blocks; i++) {
    blocks[i] = original[i]; // reset to original state
  }
}

int main() {
  int n_blocks, n_files;

  // Input number of blocks
  printf("Enter the number of blocks: ");
  scanf("%d", &n_blocks);
```

```
struct Block blocks[n_blocks], original_blocks[n_blocks];

// Input block sizes
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;
    original_blocks[i] = blocks[i]; // Save initial state
}

// Input number of files
printf("Enter the number of files: ");
scanf("%d", &n_files);

struct File files[n_files];

// Input file sizes
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);
}

int choice;

do {
    printf("\nSelect the memory management scheme:\n");
    printf("1. First Fit\n");
    printf("2. Best Fit\n");
    printf("3. Worst Fit\n");
```

```
        printf("0. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        // Reset block status before each allocation attempt
        resetBlocks(blocks, original_blocks, n_blocks);

        switch (choice) {
            case 1:
                firstFit(blocks, n_blocks, files, n_files);
                break;
            case 2:
                bestFit(blocks, n_blocks, files, n_files);
                break;
            case 3:
                worstFit(blocks, n_blocks, files, n_files);
                break;
            case 0:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice!\n");
        }
    } while (choice != 0);

    return 0;
}
```

Output :

```
Enter the number of blocks: 4
Enter the size of block 1: 10
Enter the size of block 2: 20
Enter the size of block 3: 30
Enter the size of block 4: 40
Enter the number of files: 5
Enter the size of file 1: 5
Enter the size of file 2: 10
Enter the size of file 3: 15
Enter the size of file 4: 20
Enter the size of file 5: 25

Select the memory management scheme:
1. First Fit
2. Best Fit
3. Worst Fit
0. Exit
Enter your choice: 1

Memory Management Scheme - First Fit
File_no File_size       Block_no        Block_size      Fragment
1       5               1               10              5
2       10              2               20              10
3       15              3               30              15
4       20              4               40              20
5       25              Not Allocated
```

```
Select the memory management scheme:
1. First Fit
2. Best Fit
3. Worst Fit
0. Exit
Enter your choice: 2

Memory Management Scheme - Best Fit
File_no File_size       Block_no        Block_size      Fragment
1       5               1               10              5
2       10              2               20              10
3       15              3               30              15
4       20              4               40              20
5       25              Not Allocated
```

```
Select the memory management scheme:
1. First Fit
2. Best Fit
3. Worst Fit
0. Exit
Enter your choice: 3

Memory Management Scheme - Worst Fit
File_no File_size       Block_no        Block_size      Fragment
1       5               4               40              35
2       10              3               30              20
3       15              2               20              5
4       20              Not Allocated
5       25              Not Allocated

Select the memory management scheme:
1. First Fit
2. Best Fit
3. Worst Fit
0. Exit
Enter your choice: 0
Exiting program.
```

1 Write a C program to simulate the following contigous memory allocation technique.

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

```
#include <stdio.h>
struct Block {
    int block_no;
    int block_size;
    int is_free;
};
```

~~void firstfit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {~~

```
struct File {
    int file_no;
    int file_size;
};
void firstfit (struct Block blocks [], int n_blocks,
    struct File files [], int n_files ) {
printf ("\n Memory Management Scheme -
        First Fit \n");

printf ("File_no \tFile_size\t Block_no \t Block_
 size \t Fragment \n");

for (int i=0; i < n_files ; i++) {
    int allocated 0;
for (int j=0 ; j < n_blocks ; j++) {
    if (Blocks [j]. is_free && blocks [j]. block_
     size >= files[i]. file_size)
```

```
{
int fragment = blocks[j].block_size -
                files[i].file_size;
blocks[j].is_free = 0;
printf("%d\t%d\t\t%d\t\t%d\t\t%d\n",
        files[i].file_no,
        files[i].file_size,
        blocks[j].block_no,
        blocks[j].block_size,
        fragment);
allocated = 1;
break;
}
}
if(!allocated) {
printf("%d\t%d\t\tNot allocated\n",
        files[i].file_no, files[i].file_size);
}
}
void bestfit(struct Block blocks[], int n_blocks,
struct File files[], int n_files) {
printf("\nMemory Management Scheme -
                Best fit:\n");
printf("File_no\tFile_size\tBlock_no\t
        Block_size\tFragment\n");
```

```
for (int i=0; i < n_files ; i++) {
    if (blocks[j].is_free && blocks[j].block_size
      >= files[i].file_size ) {

    int fragment = blocks[j].block_size -
                   files[i].file_size ;

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

} else {
printf ("%d\t %d\t \t Not allocated \n",
files[i].file_no , files[i].file_size);

}}
}
```

```
void worstfit (struct Blocks blocks [], int n_blocks,
struct File files [], int n_files) {
  printf ("\n Memory Management scheme - Worst
                                    fit \n");
printf ("File_no \t File_size \t Block_no \t Block_size
  \t Fragment \n");
for (int i=0; i < n_files ; i++) {
    int worst_fit_block = -1;
    int max_fragment = -1;
for (int j=0; j < n_blocks ; j++) {
  if (blocks [j]. is_free && blocks[j] . block_size
>= files [i] . file_size) {
    int fragment = blocks [j]. block_size
- files[i]. file_size ;
if (fragment > max_fragment) {
      max_fragment = fragment;
      worst_fit_block = j;
  }
 }
}
if (worst_fit_block != -1) {
  blocks [worst_fit_block]. is_free = 0;
  printf ("%d\t%d\t\t%d\t\t%d\t\t%d\n",
      files [i]. file_no,
      files [i]. file_size,
      blocks [worst_fit_block]. block_no,
      blocks [worst_fit_block]. block_size,
      max_fragment);
}
```

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

```
int choice;
printf("\n Select the memory management
        scheme : \n");
printf("1. First fit \n");
printf("2. Best fit \n");
printf("3. Worst fit \n");
printf("Enter your choice : ");
scanf("%d", &choice);
switch(choice) {
    case 1:
        firstfit(blocks, n_blocks, files, n_files);
        break;
    case 2:
        bestfit(blocks, n_blocks, files, n_files);
        break;
    case 3:
        worstfit(blocks, n_blocks, files, n_files);
        break;
    default:
        printf("Invalid choice! \n");
}
return 0;
}
```

Output

```
Enter the number of blocks : 4
Enter the size of block 1 : 10
Enter the size of block 2 : 20
Enter the size of block 3 : 30
Enter the size of block 4 : 40
Enter the number of files : 5
Enter the size of file 1 : 5
Enter the size of file 2 : 10
```

Enter the size of file 3 : 15
Enter the size of file 4 : 20
Enter the size of file 5 : 25

Select the memory management scheme :
1. First Fit
2. Best Fit
3. Worst Fit
0. Exit
Enter your choice : 1

Memory Management Scheme - First Fit

File_No	File_size	Block_no	Block_size	Fragment
1	5	1	10	5
2	10	2	20	10
3	15	3	30	15
4	20	4	40	20
5	25	Not Allocated		

Select the memory management scheme :
1. First Fit
2. Best Fit
3. Worst Fit
0. Exit
Enter your choice : 2

Memory Management scheme - Best Fit

File_No	File_size	Block_no	Block_size	Fragment
1	5	1	10	5
2	10	2	20	10
3	15	3	30	15
4	20	4	40	20
5	25	Not Allocated		

Select the memory management scheme:

1. First fit
2. Best fit
3. Worst fit
0. Exit

Enter your choice: 3

Memory Management Scheme - Worst fit

File_No:	File_size	Block_no	Block_size	fragme
1	5	4	40	35
2	10	3	30	20
3	15	2	20	5
4	20	Not Allocated		
5	25	Not Allocated		

select the memory management scheme:

1. First fit
2. Best fit
3. Worst fit
0. Exit

Enter your choice: 0

Exiting program.

Lab 7

Write a C program to simulate page replacement algorithms

a)FIFO

```
#include <stdio.h>

int main() {
   int frames, pages[50], n, frame[10], i, j, k, avail, count = 0;

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

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

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

   for(i = 0; i < frames; i++)
      frame[i] = -1;
   printf("\nPage\tFrames\t\tPage Fault\n");
   j = 0;
   for(i = 0; i < n; i++) {
      avail = 0;

      for(k = 0; k < frames; k++) {
         if(frame[k] == pages[i]) {
            avail = 1;
            break;
         }
      }

      if(avail == 0) {
         frame[j] = pages[i];
         j = (j + 1) % frames;
```

```
            count++;
            printf("%d\t", pages[i]);
            for(k = 0; k < frames; k++) {
               if(frame[k] != -1)
                  printf("%d ", frame[k]);
               else
                  printf("- ");
            }
            printf("\tYes\n");
         } else {
            printf("%d\t", pages[i]);
            for(k = 0; k < frames; k++) {
               if(frame[k] != -1)
                  printf("%d ", frame[k]);
               else
                  printf("- ");
            }
            printf("\tNo\n");
         }
      }
      printf("\nTotal Page Faults = %d\n", count);
      return 0;
   }
```

Output:

```
Enter number of pages: 15
Enter the page reference string:
7 0 1 2 0 3 0 4 2 3 0 3 1 2 0
Enter number of frames: 3

Page    Frames          Page Fault
7       7 - -   Yes
0       7 0 -   Yes
1       7 0 1   Yes
2       2 0 1   Yes
0       2 0 1   No
3       2 3 1   Yes
0       2 3 0   Yes
4       4 3 0   Yes
2       4 2 0   Yes
3       4 2 3   Yes
0       0 2 3   Yes
3       0 2 3   No
1       0 1 3   Yes
2       0 1 2   Yes
0       0 1 2   No

Total Page Faults = 12
```

1 Write C programs to demonstrate page replacement by the following techniques.

i) FIFO
ii) LRU
iii) Optimal

i)
```
#include <stdio.h>
int main () {
    int frames, pages [50], n, frame [10], i,
      j, k, avail, count = 0;
printf ("Enter number of pages : ");
scanf ("%d", &n);
printf ("Enter the page reference string\n");
for (i=0; i<n; i++)
        scanf ("%d", & pages [i]);
printf ("Enter number of frames : ");
scanf ("%d", & frames);
for (i=0 ; i< frames ; i++)
    frame [i] = -1;
printf ("\n Page \t Frames \t\t Page fault \n");
j=0;
for (i=0; i<n; i++) {
    avail = 0;
  for (k=0; k<frames ; k++) {
    if ( frame [k] = = pages [i]) {
        avail = 1;
        break;
    }
  }
}
```

```
if (avail == 0) {
    frame[j] = pages[i];
    j = (j+1) % frames;
    count++;
    printf("%d\t", pages[i]);
    for (k=0; k < frames; k++) {
        if (frame[k] != -1)
            printf("%d", frame[k]);
        else
            printf("-");
    }
    printf("\t Yes\n");
} else {
    printf("%d\t", pages[i]);
    for (k=0; k < frames; k++) {
        if (frame[k] != -1)
            printf("%d", frame[k]);
        else
            printf("-");
    }
    printf("\tNo\n");
}
}
printf("\nTotal Page Faults = %d\n", count);
return 0;
}
```

Output

Enter the number of pages : 15
Enter the page reference string :
7 0 1 2 0 3 0 4 2 3 0 3 1 2 0
Enter the number of frames : 3

Page	Frames	Page Fault
7	7 _ _	Yes
0	7 0 _	Yes
1	7 0 1	Yes
2	2 0 1	Yes
0	2 0 1	No
3	2 3 1	Yes
0	2 3 0	Yes
4	4 3 0	Yes
2	4 2 0	Yes
3	4 2 3	Yes
0	0 2 3	Yes
3	0 2 3	No
1	0 1 3	Yes
2	0 1 2	Yes
0	0 1 2	No

Total Page faults = 12.

b.) LRU

```
#include <stdio.h>

int main() {
   int n, frames, i, j, k, faults = 0;
   printf("Enter number of pages: ");
   scanf("%d", &n);
   int pages[n];
   printf("Enter the reference string: ");
   for(i = 0; i < n; i++)
      scanf("%d", &pages[i]);

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

   int frame_arr[frames];
   int time[frames]; // To track the usage time
   for(i = 0; i < frames; i++) {
      frame_arr[i] = -1;
      time[i] = 0;
   }

   int counter = 0;
   for(i = 0; i < n; i++) {
      int flag = 0;
      for(j = 0; j < frames; j++) {
         if(frame_arr[j] == pages[i]) {
            flag = 1;
            counter++;
            time[j] = counter; // Update the usage time
            break;
         }
      }

      if(flag == 0) { // Page fault
         faults++;
```

```
            int min_time = time[0], min_pos = 0;
            for(k = 1; k < frames; k++) {
                if(time[k] < min_time) {
                    min_time = time[k];
                    min_pos = k;
                }
            }
            frame_arr[min_pos] = pages[i]; // Replace the least recently used
            counter++;
            time[min_pos] = counter; // Update the usage time for the new page
        }

        printf("Frames after accessing %d: ", pages[i]);
        for(j = 0; j < frames; j++) {
            if(frame_arr[j] == -1)
                printf("- ");
            else
                printf("%d ", frame_arr[j]);
        }
        printf("\n");
    }

    printf("Total page faults: %d\n", faults);
    int Hits = n-faults;
    printf("Total page Hits: %d\n",Hits);
    return 0;
}
```

Output:

```
Enter number of pages: 7
Enter the reference string: 1 3 0 3 5 6 3
Enter number of frames: 3
Frames after accessing 1: 1 - -
Frames after accessing 3: 1 3 -
Frames after accessing 0: 1 3 0
Frames after accessing 3: 1 3 0
Frames after accessing 5: 5 3 0
Frames after accessing 6: 5 3 6
Frames after accessing 3: 5 3 6
Total page faults: 5
Total page Hits: 2
```

ii)
```
#include <stdio.h>
int main () {
    int n, frames, i, j, k, faults = 0;
    printf ("Enter number of pages : ");
    scanf ("%d", &n);
    int pages[n];
    printf ("Enter the reference string : ");
    for (i=0; i<n; i++)
        scanf ("%d", &pages[i]);
    printf ("Enter number of frames: ");
    scanf ("%d", &frames);
    int frame_arr[frames];
    int time[frames];
    for (i=0; i<frames; i++) {
        frame_arr[i] = -1;
        time[i] = 0;
    }
int counter = 0;
for (i=0; i<n; i++) {
    int flag = 0;
    for (j=0; j<frames; j++) {
        if (frame_arr[j] == pages[i]) {
            flag = 1;
            counter++;
            time[j] = counter;
            break;
        }
    }
}
```

```
if (flag == 0) {
    faults++;
    int min_time = time[0], min_pos = 0;
    for (k=1; k < frames; k++) {
        if (time[k] < min_time) {
            min_time = time[k];
            min_pos = k;
        }
    }
    frame_arr[min_pos] = pages[i];
    counter++;
    time[min_pos] = counter;
}
printf("Frames after accessing %d: ", pages[i]);
for(j=0; j<frames; j++) {
    if (frame_arr[j] == -1)
        printf("- ");
    else
        printf("%d", frame_arr[j]);
}
printf("\n");
}
printf("Total page faults: %d\n", faults);
int Hits = n - faults;
printf("Total page Hits: %d\n", Hits);
return 0;
}
```

Output

```
Enter number of pages : 7
Enter the reference string : 1 3 0 3 5 6 3
Enter the number of frames : 3
Frames after accessing 1 : 1 - -
Frames after accessing 3 : 1 3 -
Frames after accessing 0 : 1 3 0
Frames after accessing 3 : 1 3 0
Frames after accessing 5 : 5 3 0
Frames after accessing 6 : 5 3 6
Frames after accessing 3 : 5 3 6
Total page faults : 5
Total page Hits : 2.
```

c) Optimal

```
#include <stdio.h>

int main() {
   int n, frames, i, j, k, faults = 0;
   printf("Enter number of pages: ");
   scanf("%d", &n);
   int pages[n];
   printf("Enter the reference string: ");
   for(i = 0; i < n; i++)
      scanf("%d", &pages[i]);

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

   int frame_arr[frames];
   for(i = 0; i < frames; i++)
      frame_arr[i] = -1;

   for(i = 0; i < n; i++) {
      int flag = 0;
      for(j = 0; j < frames; j++) {
         if(frame_arr[j] == pages[i]) {
            flag = 1;
            break;
         }
      }
      if(flag == 0) {
         faults++;
         int pos = -1;
         for(j = 0; j < frames; j++) {
            if(frame_arr[j] == -1) {
               pos = j;
               break;
            }
         }
         if(pos == -1) {
            int farthest = i, replace_index = 0;
            for(j = 0; j < frames; j++) {
               int found = 0;
               for(k = i + 1; k < n; k++) {
                  if(frame_arr[j] == pages[k]) {
                     if(k > farthest) {
                        farthest = k;
                        replace_index = j;
                     }
                     found = 1;
                     break;
                  }
```

```
                }
                if(!found) {
                    replace_index = j;
                    break;
                }
            }
            pos = replace_index;
        }
        frame_arr[pos] = pages[i];
    }
    printf("%d: ", pages[i]);
    for(j = 0; j < frames; j++) {
        if(frame_arr[j] == -1)
            printf("_ ");
        else
            printf("%d ", frame_arr[j]);
    }
    printf("\n");
  }
  printf("Total page faults: %d\n", faults);
  int Hits = n-faults;
  printf("Total page Hits: %d\n",Hits);
  return 0;
}
```

Output:

```
Enter number of pages: 7
Enter the reference string: 1 3 0 3 5 6 3
Enter number of frames: 3
Frames after accessing 1: 1 _ _
Frames after accessing 3: 1 3 _
Frames after accessing 0: 1 3 0
Frames after accessing 3: 1 3 0
Frames after accessing 5: 5 3 0
Frames after accessing 6: 6 3 0
Frames after accessing 3: 6 3 0
Total page faults: 5
Total page Hits: 2
```

iii)
```
#include <stdio.h>
int main() {
    int n, frames, i, j, k, faults = 0;
    printf("Enter number of pages: ");
    scanf("%d", &n);
    int pages[n];
    printf("Enter the reference string: ");
    for(i=0; i<n; i++)
        scanf("%d", &pages[i]);

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

    int frame_arr[frames];
    for(i=0; i<frames; i++)
        frame_arr[i] = -1;
```

```
for (i=0; i<n; i++) {
    int flag = 0;
    for (j=0; j< frames ; j++) {
        if ( frame_arr [j] == pages [i]) {
            flag = 1;
            break;
        }
    }
if (pos == -1) {
int farthest = i , replace_index = 0;
for (j=0; j< frames ; j++) {
    int found = 0;
    for ( k = i+1; k <n;  k++) {
        if (frame_arr [j] == pages (k]){
            if (k > farthest) {
                farthest = k;
                replace_index = j;
            }
            found = 1;
            break;
        }
    }
    if (!found) {
        replace_index = j;
        break;
    }
}
pos = replace_index ;
}
frame_arr [pos] = pages [i];
}
```

```
printf ("%d:", pages[i]);
for (j=0; j < frames ; j++) {
    if (frame_arr[j] == -1)
        printf ("_ ");
    else
        printf ("%d", frame_arr[j]);
}
printf ("\n");
}
printf ("Total page faults : %d\n", faults);
int Hits = n - faults ;
printf ("Total page Hits : %d\n", Hits);
return 0;
}
```

Output

```
Enter number of pages : 7
Enter the reference string : 1 3 0 3 5 6 3
Enter the number of frames : 3
Frames after accessing 1 : 1 _ _
Frames after accessing 3 : 1 3 _
Frames after accessing 0 : 1 3 0
Frames after accessing 3 : 1 3 0
Frames after accessing 5 : 5 3 0
Frames after accessing 6 : 6 3 0
Frames after accessing 3 : 6 3 0
Total page faults : 5
Total page Hits : 2.
```