VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS

Submitted by

SHREYA MITAWA (1BM22CS266)

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 **SHREYA MITAWA (1BM22CS266),** who is bonafide student of **B. M. S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2024. The Lab report has been approved as it satisfies the academic requirements in respect of a **OPERATING SYSTEMS - (23CS4PCOPS)** work prescribed for the said degree.

Swathi Sridharan
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	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.

Program -1

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

→FCFS

→ SJF (pre-emptive & Non-preemptive)

Code:

//fcfs

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

#define MAX_PROCESS 30

int p[MAX_PROCESS], arrTime[MAX_PROCESS], burstTime[MAX_PROCESS];
int compTime[MAX_PROCESS], TAT[MAX_PROCESS], waitTime[MAX_PROCESS];

// Function to sort processes based on arrival time
void sortProcess(int arrTime[], int burstTime[], int n) {
   int temp;
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n - i - 1; j++) {
         if (arrTime[j] > arrTime[j + 1]) {
            // Swap arrival times
            temp = arrTime[j];
            arrTime[j] = arrTime[j + 1];
            arrTime[j + 1] = temp;
```

```
            // Swap burst times accordingly
            temp = burstTime[j];
            burstTime[j] = burstTime[j + 1];
            burstTime[j + 1] = temp;

            // Swap process numbers accordingly
            temp = p[j];
            p[j] = p[j + 1];
            p[j + 1] = temp;
        }
      }
   }
}

// Function to find turnaround time
int findTurnAroundTime(int ct, int at) {
   return ct - at;
}
// Function to find waiting time
int waitingTime(int tat, int bt) {
   return tat - bt;
}

int main() {
   int n;
   printf("Enter total number of processes: ");
```

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

int total_TAT = 0; // Total turnaround time
int total_WT = 0;  // Total waiting time

for (int i = 0; i < n; i++) {
   printf("Process [%d]\n", i + 1);
   printf("Arrival time: ");
   scanf("%d", &arrTime[i]);
   printf("Burst time: ");
   scanf("%d", &burstTime[i]);
   p[i] = i + 1; // Assigning process number
}

// Sort processes based on arrival time
sortProcess(arrTime, burstTime, n);

// Calculate completion time, turnaround time, and waiting time
for (int i = 0; i < n; i++) {
   if (i == 0 || arrTime[i] > compTime[i - 1]) {
      compTime[i] = arrTime[i] + burstTime[i];
   } else {
      compTime[i] = compTime[i - 1] + burstTime[i];
   }
   TAT[i] = findTurnAroundTime(compTime[i], arrTime[i]);
   waitTime[i] = waitingTime(TAT[i], burstTime[i]);
```

```
        // Summing up turnaround time and waiting time
        total_TAT += TAT[i];
        total_WT += waitTime[i];
    }

    // Calculate averages
    float avg_TAT = (float)total_TAT / n;
    float avg_WT = (float)total_WT / n;

    // Displaying results including averages
    printf("\nProcess\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", p[i], arrTime[i], burstTime[i], compTime[i], TAT[i], waitTime[i]);
    }

    printf("\nAverage Turnaround Time: %.2f", avg_TAT);
    printf("\nAverage Waiting Time: %.2f\n", avg_WT);

    return 0;
}
```

//sjf non preem

```
#include <stdio.h>

// Define a structure to represent each process
struct Process {
   int id;          // Process ID
   int at;          // Arrival Time
   int bt;          // Burst Time
   int ct;          // Completion Time
   int tt;          // Turnaround Time
   int wt;          // Waiting Time
};

// Function to sort processes based on arrival time and burst time
void sort(struct Process p[], int n);

// Function to implement Shortest Job First (SJF) scheduling algorithm
void sjf(struct Process p[], int n);

int main() {
   int n; // Number of processes
   int total_tat = 0; // Total Turnaround Time
   int total_wt = 0;  // Total Waiting Time

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

```
    // Array of processes
    struct Process p[n];

    // Input arrival time and burst time for each process
    printf("Enter the arrival time and burst time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d:\n", i + 1);
        p[i].id = i + 1;
        printf("Arrival Time: ");
        scanf("%d", &p[i].at);
        printf("Burst Time: ");
        scanf("%d", &p[i].bt);
    }

    // Sort processes based on arrival time and burst time
    sort(p, n);

    // Implement Shortest Job First (SJF) scheduling algorithm
    sjf(p, n);

    // Display process schedule
    printf("\nProcess Schedule:\n");
    printf("Process ID\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting
Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", p[i].id, p[i].at, p[i].bt, p[i].ct, p[i].tt, p[i].wt);
        total_tat += p[i].tt;
```

```
        total_wt += p[i].wt;
    }

    // Calculate and display average turnaround time and average waiting time
    printf("\nAvg TAT: %.2f", (float)total_tat / n);
    printf("\nAvg WT: %.2f", (float)total_wt / n);

    return 0;
}

// Function to sort processes based on arrival time and burst time
void sort(struct Process p[], int n) {
    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 || (p[j].at == p[j + 1].at && p[j].bt > p[j + 1].bt)) {
                // Swap processes
                struct Process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }
}

// Function to implement Shortest Job First (SJF) scheduling algorithm
void sjf(struct Process p[], int n) {
    int current_time = 0; // Current time
```

```
    for (int i = 0; i < n; i++) {
        int sj_index = i; // Index of the process with the shortest burst time
        for (int j = i + 1; j < n && p[j].at <= current_time; j++) {
            if (p[j].bt < p[sj_index].bt) {
                sj_index = j;
            }
        }
        // Update completion time, turnaround time, and waiting time
        p[sj_index].ct = current_time + p[sj_index].bt;
        p[sj_index].tt = p[sj_index].ct - p[sj_index].at;
        p[sj_index].wt = p[sj_index].tt - p[sj_index].bt;
        // Update current time
        current_time = p[sj_index].ct;
        // Swap processes
        struct Process temp = p[i];
        p[i] = p[sj_index];
        p[sj_index] = temp;
    }
}
```

//sjf preem

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

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

   int pid[n], arrival[n], burst[n], remaining[n], completion[n], waiting[n], turnaround[n];
   float avg_waiting_time = 0, avg_turnaround_time = 0;

   for (int i = 0; i < n; i++) {
      pid[i] = i + 1;
      printf("Enter arrival time and burst time for process %d: ", i + 1);
      scanf("%d %d", &arrival[i], &burst[i]);
      remaining[i] = burst[i];
   }

   int completed = 0, current_time = 0, shortest = 0;
   int min_remaining_time = INT_MAX;
   int finish_time;
   int check = 0;

   while (completed != n) {
      for (int j = 0; j < n; j++) {
         if ((arrival[j] <= current_time) &&
```

```
        (remaining[j] < min_remaining_time) && remaining[j] > 0) {
        min_remaining_time = remaining[j];
        shortest = j;
        check = 1;
    }
}

if (check == 0) {
    current_time++;
    continue;
}

remaining[shortest]--;
min_remaining_time = remaining[shortest];

if (min_remaining_time == 0) {
    min_remaining_time = INT_MAX;
}

if (remaining[shortest] == 0) {
    completed++;
    check = 0;
    finish_time = current_time + 1;
    completion[shortest] = finish_time;
    turnaround[shortest] = finish_time - arrival[shortest];
    waiting[shortest] = turnaround[shortest] - burst[shortest];
```

```
            avg_waiting_time += waiting[shortest];
            avg_turnaround_time += turnaround[shortest];
        }
        current_time++;
    }

    avg_waiting_time /= n;
    avg_turnaround_time /= n;

    printf("\nPID\t\tAT\t\tBT\t\tCT\t\tTAT\t\tWT\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", pid[i], arrival[i], burst[i], completion[i],
turnaround[i], waiting[i]);
    }

    printf("\nAverage Waiting Time: %.2f", avg_waiting_time);
    printf("\nAverage Turnaround Time: %.2f", avg_turnaround_time);

    return 0;
}
```

Result:

FSFS

Enter total number of processes: 3

Process [1]

Arrival time: 0

Burst time: 4

Process [2]

Arrival time: 1

Burst time: 3

Process [3]

Arrival time: 2

Burst time: 1

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

Average Turnaround Time: 5.33

Average Waiting Time: 2.67

SJF (PREEM)

Enter the number of processes: 5

Enter arrival time and burst time for process 1: 2 1

Enter arrival time and burst time for process 2: 1 5

Enter arrival time and burst time for process 3: 4 1

Enter arrival time and burst time for process 4: 0 6

Enter arrival time and burst time for process 5: 2 3

PID	AT	BT	CT	TAT	WT
1	2	1	3	1	0
2	1	5	16	15	10
3	4	1	5	1	0
4	0	6	11	11	5
5	2	3	7	5	2

Average Waiting Time: 3.40

Average Turnaround Time: 6.60

SJF(NON PREEM)

Enter the number of processes: 4

Enter the arrival time and burst time for each process:

Process 1:

Arrival Time: 0

Burst Time: 6

Process 2:

Arrival Time: 2

Burst Time: 8

Process 3:

Arrival Time: 4

Burst Time: 7

Process 4:

Arrival Time: 6

Burst Time: 3

Process Schedule:

Process ID	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
1	0	6	6	6	0
4	6	3	9	3	0
3	4	7	16	12	5
2	2	8	24	22	14

Avg TAT: 10.75

Avg WT: 4.75

Program -2

Question:Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time.

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

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

Code:

//priority preem

```
#include<stdio.h>


void
sort (int proc_id[], int p[], int at[], int bt[], int b[], int n)
{
  int min = p[0], temp = 0;
  for (int i = 0; i < n; i++)
          {
            min = p[i];
            for (int j = i; j < n; j++)
                    {
                      if (p[j] < min)
                              {
                                temp = at[i];
                                at[i] = at[j];
                                at[j] = temp;
                                temp = bt[j];
```

```
                    bt[j] = bt[i];
                    bt[i] = temp;
                    temp = b[j];
                    b[j] = b[i];
                    b[i] = temp;
                    temp = p[j];
                    p[j] = p[i];
                    p[i] = temp;
                    temp = proc_id[i];
                    proc_id[i] = proc_id[j];
                    proc_id[j] = temp;
                  }
                }
        }
}

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

```
            m[i] = 0;
        }
printf ("Enter priorities:\n");
for (int i = 0; i < n; i++)
        scanf ("%d", &p[i]);
printf ("Enter arrival times:\n");
for (int i = 0; i < n; i++)
        scanf ("%d", &at[i]);
printf ("Enter burst times:\n");
for (int i = 0; i < n; i++)
        {
          scanf ("%d", &bt[i]);
          b[i] = bt[i];
          m[i] = -1;
          rt[i] = -1;
        }

sort (proc_id, p, at, bt, b, n);

//completion time
int count = 0, pro = 0, priority = p[0];
int x = 0;
c = 0;
while (count < n)
        {
          for (int i = 0; i < n; i++)
                  {
```

```
            if (at[i] <= c && p[i] >= priority && b[i] > 0 && m[i] != 1)
                {
                  x = i;
                  priority = p[i];
                }
            }
        if (b[x] > 0)
            {
             if (rt[x] == -1)
                rt[x] = c - at[x];
             b[x]--;
             c++;
            }
        if (b[x] == 0)
            {
             count++;
             ct[x] = c;
             m[x] = 1;
             while (x >= 1 && b[x] == 0)
                priority = p[--x];
            }
        if (count == n)
            break;
      }

//turnaround time and RT
for (int i = 0; i < n; i++)
```

```
        tat[i] = ct[i] - at[i];
  //waiting time
  for (int i = 0; i < n; i++)
        wt[i] = tat[i] - bt[i];

  printf ("Priority scheduling(Pre-Emptive):\n");
  printf ("PID\tPrior\tAT\tBT\tCT\tTAT\tWT\tRT\n");
  for (int i = 0; i < n; i++)
        printf ("P%d\t %d\t\t%d\t%d\t%d\t%d\t%d\t%d\n", proc_id[i], p[i], at[i],
                    bt[i], ct[i], tat[i], wt[i], rt[i]);

  for (int i = 0; i < n; i++)
        {
          ttat += tat[i];
          twt += wt[i];
        }
  avg_tat = ttat / (double) n;
  avg_wt = twt / (double) n;
  printf ("\nAverage turnaround time:%lfms\n", avg_tat);
  printf ("\nAverage waiting time:%lfms\n", avg_wt);
}
```

//priority non preem

```
#include<stdio.h>


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

```
                    }
            }
}

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

```
        m[i] = -1;
        rt[i] = -1;
      }

sort (proc_id, p, at, bt, n);

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

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

//turnaround time and RT
for (int i = 0; i < n; i++)
        tat[i] = ct[i] - at[i];
//waiting time
for (int i = 0; i < n; i++)
        wt[i] = tat[i] - bt[i];

printf ("\nPriority scheduling:\n");
printf ("PID\tPrior\tAT\tBT\tCT\tTAT\tWT\tRT\n");
for (int i = 0; i < n; i++)
        printf ("P%d\t %d\t\t%d\t%d\t%d\t%d\t%d\t%d\n", proc_id[i], p[i], at[i],
                        bt[i], ct[i], tat[i], wt[i], rt[i]);

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

```
        {
          ttat += tat[i];
          twt += wt[i];
        }
  avg_tat = ttat / (double) n;
  avg_wt = twt / (double) n;
  printf ("\nAverage turnaround time:%lfms\n", avg_tat);
  printf ("\nAverage waiting time:%lfms\n", avg_wt);
}
```

//round-robin

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

void sort(int pid[], int at[], int bt[], int n) {
   int temp;
   for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
         if (at[j] < at[i]) {
            // Swap arrival times
            temp = at[i];
            at[i] = at[j];
            at[j] = temp;
            // Swap burst times
            temp = bt[i];
            bt[i] = bt[j];
            bt[j] = temp;
            // Swap process IDs
            temp = pid[i];
            pid[i] = pid[j];
            pid[j] = temp;
         }
      }
   }
}

int main() {
```

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

    int pid[n], at[n], bt[n], st[n], ct[n], tat[n], wt[n], rt[n];
    int burst_remaining[n];
    int total_turnaround_time = 0, total_waiting_time = 0, total_response_time = 0,
total_idle_time = 0;
    int q[100], front = 0, rear = 0, current_time = 0, completed = 0;
    int mark[n];

    for (int i = 0; i < n; i++) {
        printf("Enter arrival time of process %d: ", i + 1);
        scanf("%d", &at[i]);
        printf("Enter burst time of process %d: ", i + 1);
        scanf("%d", &bt[i]);
        burst_remaining[i] = bt[i];
        pid[i] = i + 1;
        mark[i] = 0;
        printf("\n");
    }

    sort(pid, at, bt, n);

    q[rear++] = 0; // start with the first process
    mark[0] = 1;
```

```
while (completed != n) {
    int idx = q[front++];

    if (burst_remaining[idx] == bt[idx]) {
        st[idx] = (current_time > at[idx]) ? current_time : at[idx];
        total_idle_time += st[idx] - current_time;
        current_time = st[idx];
    }

    if (burst_remaining[idx] - tq > 0) {
        burst_remaining[idx] -= tq;
        current_time += tq;
    } else {
        current_time += burst_remaining[idx];
        burst_remaining[idx] = 0;
        completed++;
        ct[idx] = current_time;
    }

    for (int i = 1; i < n; i++) {
        if (burst_remaining[i] > 0 && at[i] <= current_time && mark[i] == 0) {
            q[rear++] = i;
            mark[i] = 1;
        }
    }
```

```
    if (burst_remaining[idx] > 0) {
        q[rear++] = idx;
    }

    if (front == rear) {
        for (int i = 1; i < n; i++) {
            if (burst_remaining[i] > 0) {
                q[rear++] = i;
                mark[i] = 1;
                break;
            }
        }
    }
}

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

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

for (int i = 0; i < n; i++) {
    rt[i] = st[i] - at[i];
```

```
        total_response_time += rt[i];
    }

    float avg_turnaround_time = (float) total_turnaround_time / n;
    float avg_waiting_time = (float) total_waiting_time / n;
    float avg_response_time = (float) total_response_time / n;

    printf("\n#P\tAT\tBT\tST\tCT\tTAT\tWT\tRT\n\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", pid[i], at[i], bt[i], st[i], ct[i], tat[i], wt[i],
rt[i]);
    }
    printf("Average Turnaround Time = %.2f\n", avg_turnaround_time);
    printf("Average Waiting Time = %.2f\n", avg_waiting_time);
    printf("Average Response Time = %.2f\n", avg_response_time);

    return 0;
}
```

Result:

PRIORITY NON PREEM

Enter number of processes: 4

Enter priorities:

10

20

30

40

Enter arrival times:

0

1

2

4

Enter burst times:

5

4

2

1

Priority scheduling:

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

PRIORITY PREEM

Enter number of processes: 4

Enter priorities:

10

20

30

40

Enter arrival times:

0

1

2

4

Enter burst times:

5

4

2

1

Priority scheduling(Pre-Emptive):

PID	Prior	AT	BT	CT	TAT	WT	RT	
P1	10		0	5	12	12	7	0
P2	20		1	4	8	7	3	0
P3	30		2	2	4	2	0	0
P4	40		4	1	5	1	0	0

Average turnaround time:5.500000ms

Average waiting time:2.500000ms

ROUND ROBIN

AT - Arrival Time of the process

BT - Burst time of the process

ST - Start time of the process

CT - Completion time of the process

TAT - Turnaround time of the process

WT - Waiting time of the process

RT - Response time of the process

Formulas used:

TAT = CT - AT

WT = TAT - BT

RT = ST - AT

OUTPUT:

Enter the number of processes: 5

Enter time quantum: 2

Enter arrival time of process 1: 0

Enter burst time of process 1: 5

Enter arrival time of process 2: 1

Enter burst time of process 2: 3

Enter arrival time of process 3: 2

Enter burst time of process 3: 1

Enter arrival time of process 4: 3

Enter burst time of process 4: 2

Enter arrival time of process 5: 4

Enter burst time of process 5: 3

#P	AT	BT	ST	CT	TAT	WT	RT
1	0	5	0	13	13	8	0
2	1	3	2	12	11	8	1
3	2	1	4	5	3	2	2
4	3	2	7	9	6	4	4
5	4	3	9	14	10	7	5

Average Turnaround Time = 8.60

Average Waiting Time = 5.80

Average Response Time = 2.40

Program -3

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

Code:

```
#include <stdio.h>


void sort(int proc_id[], int at[], int bt[], int n) {
   int min, temp;
   for(int i=0; i<n-1; i++) {
      for(int j=i+1; j<n; j++) {
         if(at[j] < at[i]) {
            // Swap arrival time
            temp = at[i];
            at[i] = at[j];
            at[j] = temp;
            // Swap burst time
            temp = bt[i];
            bt[i] = bt[j];
            bt[j] = temp;
            // Swap process ID
            temp = proc_id[i];
            proc_id[i] = proc_id[j];
            proc_id[j] = temp;
         }
```

```
        }
    }
}

void simulateFCFS(int proc_id[], int at[], int bt[], int n, int start_time) {
    int c = start_time, ct[n], tat[n], wt[n];
    double ttat = 0.0, twt = 0.0;

    // Completion time
    for(int i=0; i<n; i++) {
        if(c >= at[i])
            c += bt[i];
        else
            c = at[i] + bt[i];
        ct[i] = c;
    }

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

    // Waiting time
    for(int i=0; i<n; i++)
        wt[i] = tat[i] - bt[i];

    printf("PID\tAT\tBT\tCT\tTAT\tWT\n");
    for(int i=0; i<n; i++) {
```

```
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", proc_id[i], at[i], bt[i], ct[i], tat[i], wt[i]);
        ttat += tat[i];
        twt += wt[i];
    }

    printf("Average Turnaround Time: %.2lf ms\n", ttat/n);
    printf("Average Waiting Time: %.2lf ms\n", twt/n);
}

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

    int proc_id[n], at[n], bt[n], type[n];
    int sys_proc_id[n], sys_at[n], sys_bt[n], user_proc_id[n], user_at[n], user_bt[n];
    int sys_count = 0, user_count = 0;

    for(int i=0; i<n; i++) {
        proc_id[i] = i + 1;
        printf("Enter arrival time, burst time and type (0 for system, 1 for user) for process %d: ",
i+1);
        scanf("%d %d %d", &at[i], &bt[i], &type[i]);

        if(type[i] == 0) {
            sys_proc_id[sys_count] = proc_id[i];
            sys_at[sys_count] = at[i];
            sys_bt[sys_count] = bt[i];
```

```
            sys_count++;
        } else {
            user_proc_id[user_count] = proc_id[i];
            user_at[user_count] = at[i];
            user_bt[user_count] = bt[i];
            user_count++;
        }
    }

    // Sort both queues by arrival time
    sort(sys_proc_id, sys_at, sys_bt, sys_count);
    sort(user_proc_id, user_at, user_bt, user_count);

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

    // Find the time when system processes finish
    int system_end_time = 0; // Initialize system_end_time to 0. This variable will hold the time at which the last system process finishes.
    if (sys_count > 0) {    // Check if there are any system processes. If there are no system processes (sys_count == 0), there's no need to calculate system_end_time.
        system_end_time = sys_at[sys_count - 1] + sys_bt[sys_count - 1];   // Set system_end_time to the sum of the arrival time and burst time of the last system process in the sorted list. This is a rough initial estimate, assuming that all previous processes have completed before the last one starts.
        for (int i = 0; i < sys_count - 1; i++) {       // Loop over all system processes
            if (sys_at[i + 1] > system_end_time) {       // Check if the arrival time of the next process is greater than the current system_end_time.
```

```
            system_end_time = sys_at[i + 1];         // If it is, update system_end_time to this process's arrival time. This handles any idle time (gaps) between processes.
        }
        system_end_time += sys_bt[i];   // Add the burst time of the current process to system_end_time. This updates the end time to include the time taken by the current process.
    }
  }

  printf("\nUser Processes Scheduling:\n");
  simulateFCFS(user_proc_id, user_at, user_bt, user_count, system_end_time);
}
```

Result:

Enter number of processes: 4

Enter arrival time, burst time and type (0 for system, 1 for user) for process 1: 0 2 0

Enter arrival time, burst time and type (0 for system, 1 for user) for process 2: 0 1 1

Enter arrival time, burst time and type (0 for system, 1 for user) for process 3: 0 5 0

Enter arrival time, burst time and type (0 for system, 1 for user) for process 4: 0 3 1

System Processes Scheduling:

PID	AT	BT	CT	TAT	WT
1	0	2	2	2	0
3	0	5	7	7	2

Average Turnaround Time: 4.50 ms

Average Waiting Time: 1.00 ms

User Processes Scheduling:

PID	AT	BT	CT	TAT	WT
2	0	1	8	8	7
4	0	3	11	11	8

Average Turnaround Time: 9.50 ms

Average Waiting Time: 7.50 ms

Program -4

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

a) Rate- Monotonic

b) Earliest-deadline First

c) Proportional scheduling

Code:

```
//rate-monotonic
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
void
sort (int proc[], int b[], int pt[], int n)
{
 int temp = 0;
 for (int i = 0; i < n; i++)
         {
           for (int j = i; j < n; j++)
                   {
                     if (pt[j] < pt[i])
                             {
                               temp = pt[i];
                               pt[i] = pt[j];
                               pt[j] = temp;
                               temp = b[j];
```

```
                    b[j] = b[i];
                    b[i] = temp;
                    temp = proc[i];
                    proc[i] = proc[j];
                    proc[j] = temp;
                  }
                }
          }
}

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

int
lcmul (int p[], int n)
{
  int lcm = p[0];
```

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



void
main ()
{
 int n;
 printf ("Enter the number of processes:");
 scanf ("%d", &n);
 int proc[n], b[n], pt[n], rem[n];
 printf ("Enter the CPU burst times:\n");
 for (int i = 0; i < n; i++)
          {
           scanf ("%d", &b[i]);
           rem[i] = b[i];
          }
 printf ("Enter the time periods:\n");
 for (int i = 0; i < n; i++)
          scanf ("%d", &pt[i]);
 for (int i = 0; i < n; i++)
          proc[i] = i + 1;
```

```
sort (proc, b, pt, n);
//LCM
int l = lcmul (pt, n);
printf ("LCM=%d\n", l);

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

//feasibility
double sum = 0.0;
for (int i = 0; i < n; i++)
        {
          sum += (double) b[i] / pt[i];
        }
double rhs = n * (pow (2.0, (1.0 / n)) - 1.0);
printf ("\n%lf <= %lf =>%s\n", sum, rhs, (sum <= rhs) ? "true" : "false");
if (sum > rhs)
        exit (0);

printf ("Scheduling occurs for %d ms\n\n", l);

//RMS
int time = 0, prev = 0, x = 0;
while (time < l)
        {
```

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

```
            time++;
          }
}
```

// Earliest Deadline First C Program

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
void
sort (int proc[], int d[], int b[], int pt[], int n)
{
 int temp = 0;
 for (int i = 0; i < n; i++)
          {
            for (int j = i; j < n; j++)
                    {
                      if (d[j] < d[i])
                              {
                                temp = d[j];
                                d[j] = d[i];
                                d[i] = temp;
                                temp = pt[i];
                                pt[i] = pt[j];
                                pt[j] = temp;
                                temp = b[j];
                                b[j] = b[i];
```

```
                    b[i] = temp;
                    temp = proc[i];
                    proc[i] = proc[j];
                    proc[j] = temp;
                  }
                }
          }
}

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

int
lcmul (int p[], int n)
{
  int lcm = p[0];
  for (int i = 1; i < n; i++)
```

```
        {
          lcm = (lcm * p[i]) / gcd (lcm, p[i]);
        }
  return lcm;
}



void
main ()
{
  int n;
  printf ("Enter the number of processes:");
  scanf ("%d", &n);
  int proc[n], b[n], pt[n], d[n], rem[n];
  printf ("Enter the CPU burst times:\n");
  for (int i = 0; i < n; i++)
        {
          scanf ("%d", &b[i]);
          rem[i] = b[i];
        }
  printf ("Enter the deadlines:\n");
  for (int i = 0; i < n; i++)
        scanf ("%d", &d[i]);
  printf ("Enter the time periods:\n");
  for (int i = 0; i < n; i++)
        scanf ("%d", &pt[i]);
  for (int i = 0; i < n; i++)
```

```
        proc[i] = i + 1;

sort (proc, d, b, pt, n);
//LCM
int l = lcmul (pt, n);

printf ("\nEarliest Deadline Scheduling:\n");
printf ("PID\t Burst\tDeadline\tPeriod\n");
for (int i = 0; i < n; i++)
        printf ("%d\t\t%d\t\t%d\t\t%d\n", proc[i], b[i], d[i], pt[i]);

printf ("Scheduling occurs for %d ms\n\n", l);

//EDF
int time = 0, prev = 0, x = 0;
int nextDeadlines[n];
for (int i = 0; i < n; i++)
        {
          nextDeadlines[i] = d[i];
          rem[i] = b[i];
        }
while (time < l)
        {
          for (int i = 0; i < n; i++)
                  {
                    if (time % pt[i] == 0 && time != 0)
                            {
```

```
                    nextDeadlines[i] = time + d[i];
                    rem[i] = b[i];
                  }
            }
      int minDeadline = l + 1;
      int taskToExecute = -1;
      for (int i = 0; i < n; i++)
            {
             if (rem[i] > 0 && nextDeadlines[i] < minDeadline)
                  {
                   minDeadline = nextDeadlines[i];
                   taskToExecute = i;
                  }
            }
      if (taskToExecute != -1)
            {
             printf ("%dms : Task %d is running.\n", time, proc[taskToExecute]);
             rem[taskToExecute]--;
            }
      else
            {
             printf ("%dms: CPU is idle.\n", time);
            }

      time++;
     }
}
```

//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;
}
```

Result:

a) Rate- Monotonic

Enter the number of processes:3

Enter the CPU burst times:

2

1

3

Enter the time periods:

5

3

7

LCM=105

Rate Monotone Scheduling:

PID	Burst	Period
2	1	3
1	2	5
3	3	7

1.161905 <= 0.779763 =>false

b) Earliest-deadline First

Enter the number of processes:3

Enter the CPU burst times:

2

1

3

Enter the deadlines:

4

2

7

Enter the time periods:

5

3

7

Earliest Deadline Scheduling:

PID	Burst	Deadline	Period
2	1	2	3
1	2	4	5
3	3	7	7

Scheduling occurs for 105 ms

0ms : Task 2 is running.

1ms : Task 1 is running.

2ms : Task 1 is running.

3ms : Task 2 is running.

4ms : Task 3 is running.

5ms : Task 3 is running.

6ms : Task 3 is running.

7ms : Task 2 is running.

8ms : Task 1 is running.

9ms : Task 1 is running.

10ms : Task 2 is running.

…….

c) 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: 10

Tickets picked: 8, Winner: P1

Tickets picked: 19, Winner: P2

Tickets picked: 10, Winner: P1

Tickets picked: 25, Winner: P2

Tickets picked: 51, Winner: P3

Tickets picked: 13, Winner: P2

Tickets picked: 58, Winner: P3

Tickets picked: 25, Winner: P2

Tickets picked: 47, Winner: P3

Tickets picked: 33, Winner: P3

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.

Program -5

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

Code:

```
#include<stdio.h>
#include<stdlib.h>
int mutex=1,full=0,empty=5,x=0;
void wait()
{
   --mutex;
}
void signal()
{
   ++mutex;
}
void producer()
{
   wait();++full;--empty;x++;
   printf("Producer has produced: Item %d\n",x);
   signal();
}
void consumer()
{
   wait();--full;++empty;
   printf("Consumer has consumed: Item %d\n",x);
   x--;signal();
}
```

```
void main()
{
   int ch;
   printf("Enter 1.Producer 2.Consumer 3.Exit\n");
   while(1)
   {
      printf("Enter your choice:\n");
      scanf("%d",&ch);
      switch(ch)
      {
         case 1:
            if(mutex==1 && empty!=0)
            producer();
            else
               printf("Buffer is full!\n");
            break;
         case 2:
            if(mutex==1 && full!=0)
            consumer();
            else
               printf("Buffer is empty!\n");
            break;
         case 3:exit(0);
         default:printf("Invalid choice!\n");
      }
   }
}
```

Result:

Enter 1.Producer 2.Consumer 3.Exit

Enter your choice:

1

Producer has produced: Item 1

Enter your choice:

1

Producer has produced: Item 2

Enter your choice:

1

Producer has produced: Item 3

Enter your choice:

1

Producer has produced: Item 4

Enter your choice:

1

Producer has produced: Item 5

Enter your choice:

1

Buffer is full!

Enter your choice:

2

Consumer has consumed: Item 5

Enter your choice:

2

Consumer has consumed: Item 4

Enter your choice:

2

Consumer has consumed: Item 3

Enter your choice:

2

Consumer has consumed: Item 2

Enter your choice:

2

Consumer has consumed: Item 1

Enter your choice:

2

Buffer is empty!

Enter your choice:

3

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;
}
```

Result:

Enter the total number of philosophers: 5

How many are hungry: 3

Enter philosopher 1 position (1 to 5): 1

Enter philosopher 2 position (1 to 5): 3

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 1 is granted to eat

Philosopher 1 has finished eating

Philosopher 3 is granted to eat

Philosopher 3 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: 2

Allow two philosophers to eat at the same time

Philosopher 1 is granted to eat

Philosopher 3 is granted to eat

Philosopher 1 has finished eating

Philosopher 5 is granted to eat

Philosopher 3 has finished eating

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

Program -7

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

Code:

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

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

bool isSafe(int P, int R, int processes[], int avail[], int max[][R], int allot[][R]) {
   int need[P][R];
   calculateNeed(P, R, need, max, allot);

   bool finish[P];
   for (int i = 0; i < P; i++) {
      finish[i] = 0;
   }

   int safeSeq[P];
   int work[R];
   for (int i = 0; i < R; i++) {
      work[i] = avail[i];
   }
```

```
int count = 0;
while (count < P) {
    bool found = false;
    for (int p = 0; p < P; p++) {
        if (finish[p] == 0) {
            int j;
            for (j = 0; j < R; j++)
                if (need[p][j] > work[j])
                    break;

            if (j == R) {
                printf("P%d is visited (", p);
                for (int k = 0; k < R; k++) {
                    work[k] += allot[p][k];
                    printf("%d ", work[k]);
                }
                printf(")\n");
                safeSeq[count++] = p;
                finish[p] = 1;
                found = true;
            }
        }
    }

    if (found == false) {
        printf("System is not in safe state\n");
```

```
            return false;
        }
    }

    printf("SYSTEM IS IN SAFE STATE\nThe Safe Sequence is -- (");
    for (int i = 0; i < P; i++) {
        printf("P%d ", safeSeq[i]);
    }
    printf(")\n");

    return true;
}

int main() {
    int P, R;
    printf("Enter number of processes: ");
    scanf("%d", &P);
    printf("Enter number of resources: ");
    scanf("%d", &R);

    int processes[P];
    int avail[R];
    int max[P][R];
    int allot[P][R];

    for (int i = 0; i < P; i++) {
        processes[i] = i;
```

```
}

for (int i = 0; i < P; i++) {
   printf("Enter details for P%d\n", i);
   printf("Enter allocation -- ");
   for (int j = 0; j < R; j++) {
      scanf("%d", &allot[i][j]);
   }
   printf("Enter Max -- ");
   for (int j = 0; j < R; j++) {
      scanf("%d", &max[i][j]);
   }
}

printf("Enter Available Resources -- ");
for (int i = 0; i < R; i++) {
   scanf("%d", &avail[i]);
}

isSafe(P, R, processes, avail, max, allot);

printf("\nProcess\tAllocation\tMax\tNeed\n");
for (int i = 0; i < P; i++) {
   printf("P%d\t", i);
   for (int j = 0; j < R; j++) {
      printf("%d ", allot[i][j]);
   }
```

```
        printf("\t");
        for (int j = 0; j < R; j++) {
            printf("%d ", max[i][j]);
        }
        printf("\t");
        for (int j = 0; j < R; j++) {
            printf("%d ", max[i][j] - allot[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Result:

```
Enter number of processes: 5
Enter number of resources: 3
Enter details for P0
Enter allocation -- 0 1 0
Enter Max -- 7 5 3
Enter details for P1
Enter allocation -- 2 0 0
Enter Max -- 3 2 2
Enter details for P2
Enter allocation -- 3 0 2
Enter Max -- 9 0 2
Enter details for P3
```

Enter allocation -- 2 1 1

Enter Max -- 2 2 2

Enter details for P4

Enter allocation -- 0 0 2

Enter Max -- 4 3 3

Enter Available Resources -- 3 3 2

P1 is visited (5 3 2)

P3 is visited (7 4 3)

P4 is visited (7 4 5)

P0 is visited (7 5 5)

P2 is visited (10 5 7)

SYSTEM IS IN SAFE STATE

The Safe Sequence is -- (P1 P3 P4 P0 P2)

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

Program -8

Question:Write a C program to simulate deadlock detection

Code:

```
#include <stdio.h>

int main() {
   int n, m, i, j, k;

   // Take user input for number of processes and resources
   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];

   // Take user input for allocation matrix
   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]);
      }
   }

   // Take user input for request matrix
   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]);
   }
}

// Take user input for available resources
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; // Initially all processes are unfinished
}

// Copy available resources to work array
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;
    }
}

// Check for deadlock
int deadlock = 0;
for (i = 0; i < n; i++) {
    if (finish[i] == 0) {
```

```
            deadlock = 1;
            printf("System is in a deadlock state.\n");
            printf("The deadlocked processes are: ");
            for (j = 0; j < n; j++) {
                if (finish[j] == 0) {
                    printf("P%d ", j);
                }
            }
            printf("\n");
            break;
        }
    }

    if (deadlock == 0) {
        printf("System is not in a deadlock state.\n");
        printf("Safe Sequence is: ");
        for (i = 0; i < n; i++) {
            printf("P%d ", safeSeq[i]);
        }
        printf("\n");
    }
    return 0;
}
```

Result:

Enter the number of processes: 5

Enter the number of resources: 3

Enter the allocation matrix:

Process 0: 0 1 0

Process 1: 2 0 0

Process 2: 3 0 2

Process 3: 2 1 1

Process 4: 0 0 2

Enter the request matrix:

Process 0: 0 0 0

Process 1: 2 0 2

Process 2: 0 0 0

Process 3: 1 0 0

Process 4: 0 0 2

Enter the available resources: 0 0 0

System is not in a deadlock state.

Safe Sequence is: P0 P2 P3 P4 P1

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>
#include <stdlib.h>

#define MAX 25

void firstFit(int nb, int nf, int b[], int f[]) {
   int ff[MAX] = {0};
   int allocated[MAX] = {0}; // To keep track of allocated blocks

   for (int i = 0; i < nf; i++) {
      ff[i] = -1; // Initialize as not allocated
      for (int j = 0; j < nb; j++) {
         if (allocated[j] == 0 && b[j] >= f[i]) {
            ff[i] = j;
            allocated[j] = 1;
            break;
         }
      }
   }

   printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:");
   for (int i = 0; i < nf; i++) {
```

```
        if (ff[i] != -1)
            printf("\n%d\t\t%d\t\t%d\t\t%d", i + 1, f[i], ff[i] + 1, b[ff[i]]);
        else
            printf("\n%d\t\t%d\t\t-\t\t-", i + 1, f[i]);
    }
}

void bestFit(int nb, int nf, int b[], int f[]) {
    int ff[MAX] = {0};
    int allocated[MAX] = {0}; // To keep track of allocated blocks

    for (int i = 0; i < nf; i++) {
        int best = -1;
        ff[i] = -1; // Initialize as not allocated
        for (int j = 0; j < nb; j++) {
            if (allocated[j] == 0 && b[j] >= f[i]) {
                if (best == -1 || b[j] < b[best])
                    best = j;
            }
        }
        if (best != -1) {
            ff[i] = best;
            allocated[best] = 1;
        }
    }

    printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:");
```

```
    for (int i = 0; i < nf; i++) {
        if (ff[i] != -1)
            printf("\n%d\t\t%d\t\t%d\t\t%d", i + 1, f[i], ff[i] + 1, b[ff[i]]);
        else
            printf("\n%d\t\t%d\t\t-\t\t-", i + 1, f[i]);
    }
}

void worstFit(int nb, int nf, int b[], int f[]) {
    int ff[MAX] = {0};
    int allocated[MAX] = {0}; // To keep track of allocated blocks

    for (int i = 0; i < nf; i++) {
        int worst = -1;
        ff[i] = -1; // Initialize as not allocated
        for (int j = 0; j < nb; j++) {
            if (allocated[j] == 0 && b[j] >= f[i]) {
                if (worst == -1 || b[j] > b[worst])
                    worst = j;
            }
        }
        if (worst != -1) {
            ff[i] = worst;
            allocated[worst] = 1;
        }
    }
```

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

int main() {
    int nb, nf, choice;

    printf("Memory Management Scheme");
    printf("\nEnter the number of blocks: ");
    scanf("%d", &nb);
    printf("Enter the number of files: ");
    scanf("%d", &nf);
    int b[nb], f[nf];
    printf("\nEnter the size of the blocks:\n");
    for (int i = 0; i < nb; i++) {
        printf("Block %d: ", i + 1);
        scanf("%d", &b[i]);
    }
    printf("Enter the size of the files:\n");
    for (int i = 0; i < nf; i++) {
        printf("File %d: ", i + 1);
        scanf("%d", &f[i]);
```

```
}

while (1) {
   printf("\n1. First Fit\n2. Best Fit\n3. Worst Fit\n4. Exit\n");
   printf("Enter your choice: ");
   scanf("%d", &choice);
   switch (choice) {
      case 1:
         printf("\n\tMemory Management Scheme - First Fit\n");
         firstFit(nb, nf, b, f);
         break;
      case 2:
         printf("\n\tMemory Management Scheme - Best Fit\n");
         bestFit(nb, nf, b, f);
         break;
      case 3:
         printf("\n\tMemory Management Scheme - Worst Fit\n");
         worstFit(nb, nf, b, f);
         break;
      case 4:
         printf("\nExiting...\n");
         exit(0);
         break;
      default:
         printf("\nInvalid choice.\n");
         break;
   }
```

```
    }

    return 0;
}
```

Result:

Memory Management Scheme

Enter the number of blocks: 5

Enter the number of files: 4

Enter the size of the blocks:

Block 1: 100

Block 2: 500

Block 3: 200

Block 4: 300

Block 5: 600

Enter the size of the files:

File 1: 212

File 2: 417

File 3: 112

File 4: 426

1. First Fit

2. Best Fit

3. Worst Fit

4. Exit

Enter your choice: 1

Memory Management Scheme - First Fit

File_no:	File_size :	Block_no:	Block_size:
1	212	2	500
2	417	5	600
3	112	3	200
4	426	-	-

1. First Fit
2. Best Fit
3. Worst Fit
4. Exit

Enter your choice: 2

Memory Management Scheme - Best Fit

File_no:	File_size :	Block_no:	Block_size:
1	212	4	300
2	417	2	500
3	112	3	200
4	426	5	600

1. First Fit
2. Best Fit
3. Worst Fit
4. Exit

Enter your choice: 3

Memory Management Scheme - Worst Fit

File_no:	File_size :	Block_no:	Block_size:
1	212	5	600
2	417	2	500
3	112	4	300
4	426	-	-

1. First Fit

2. Best Fit

3. Worst Fit

4. Exit

Enter your choice: 4

Exiting...

Program -10

Question:Write a C program to simulate page replacement algorithms a) FIFO b) LRU c) Optimal

Code:

```
#include <stdio.h>

// FIFO (First-In-First-Out)
void fifo(int page_table[], int page_table_size, int reference_string[], int reference_string_size) {
    int page_faults = 0;
    int page_hits = 0;
    int front = 0;

    for (int i = 0; i < reference_string_size; i++) {
        int page = reference_string[i];
        int found = 0;

        for (int j = 0; j < page_table_size; j++) {
            if (page_table[j] == page) {
                found = 1;
                page_hits++;
                break;
            }
        }

        if (!found) {
            page_faults++;
            page_table[front] = page;
```

```
            front = (front + 1) % page_table_size;
        }
    }

    printf("FIFO Page Faults: %d\n", page_faults);
    printf("FIFO Page Hits: %d\n", page_hits);
}
```

// OPTIMAL

```
void optimal(int page_table[], int page_table_size, int reference_string[], int
reference_string_size) {
    int page_faults = 0;
    int page_hits = 0;

    for (int i = 0; i < reference_string_size; i++) {
        int page = reference_string[i];
        int found = 0;
        int max_distance = -1;
        int victim_index = -1;

        for (int j = 0; j < page_table_size; j++) {
            if (page_table[j] == page) {
                found = 1;
                page_hits++;
                break;
            }
        }
```

```
        if (!found) {
            page_faults++;

            for (int j = 0; j < page_table_size; j++) {
                int distance = -1;

                for (int k = i + 1; k < reference_string_size; k++) {
                    if (page_table[j] == reference_string[k]) {
                        distance = k - i;
                        break;
                    }
                }

                if (distance > max_distance) {
                    max_distance = distance;
                    victim_index = j;
                }
            }

            page_table[victim_index] = page;
        }
    }

    printf("OPTIMAL Page Faults: %d\n", page_faults);
    printf("OPTIMAL Page Hits: %d\n", page_hits);
}
```

```
// LRU (Least Recently Used)
void lru(int page_table[], int page_table_size, int reference_string[], int reference_string_size) {
    int page_faults = 0;
    int page_hits = 0;
    int timestamps[page_table_size];

    for (int i = 0; i < page_table_size; i++) {
        timestamps[i] = 0;
    }

    for (int i = 0; i < reference_string_size; i++) {
        int page = reference_string[i];
        int found = 0;
        int min_timestamp = -1;
        int victim_index = -1;

        for (int j = 0; j < page_table_size; j++) {
            if (page_table[j] == page) {
                found = 1;
                page_hits++;
                timestamps[j] = i;
                break;
            }
        }

        if (!found) {
            page_faults++;
```

```
            for (int j = 0; j < page_table_size; j++) {
                if (timestamps[j] < min_timestamp || min_timestamp == -1) {
                    min_timestamp = timestamps[j];
                    victim_index = j;
                }
            }

            page_table[victim_index] = page;
            timestamps[victim_index] = i;
        }
    }

    printf("LRU Page Faults: %d\n", page_faults);
    printf("LRU Page Hits: %d\n", page_hits);
}

int main() {
    int page_table_size;
    printf("Enter page table size: ");
    scanf("%d", &page_table_size);

    int page_table[page_table_size];
    for (int i = 0; i < page_table_size; i++) {
        page_table[i] = -1;
    }
```

```
    int reference_string_size;
    printf("Enter reference string size: ");
    scanf("%d", &reference_string_size);

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

    printf("FIFO:\n");
    fifo(page_table, page_table_size, reference_string, reference_string_size);

    printf("OPTIMAL:\n");
    optimal(page_table, page_table_size, reference_string, reference_string_size);

    printf("LRU:\n");
    lru(page_table, page_table_size, reference_string, reference_string_size);
    return 0;
}
```

Result:

Enter page table size: 3

Enter reference string size: 7

Enter reference string: 1 2 3 4 1 2 5

FIFO:

FIFO Page Faults: 7

FIFO Page Hits: 0

OPTIMAL:

OPTIMAL Page Faults: 4

OPTIMAL Page Hits: 3

LRU:

LRU Page Faults: 5

LRU Page Hits: 2