VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS

(23CS4PCOPS)

Submitted by

Manav Kumar (1BM22CS348)

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 **Manav Kumar(1BM22CS348),** 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 **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	Analyze various Operating system strategies and techniques
CO3	Demonstrate the different functionalities of Operating System
CO4	Conduct practical experiments to implement the functionalities of Operating system

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:
1.FCFS

```
#include<stdio.h>
void main()
{
   int n;
   printf("Enter number of processes:\n");
   scanf("%d",&n);
   int pr[n], at[n], bt[n], ct[n], tat[n], wt[n];
   printf("Enter Process number:\n");
   for (int i=0; i<n; i++)
   {
      scanf("%d", &pr[i]);
   }
   printf("Enter Arrival Time:\n");
   for (int i=0; i<n; i++)
   {
      scanf("%d", &at[i]);
   }
   printf("Enter Burst Time:\n");
   for (int i=0; i<n; i++)
   {
      scanf("%d", &bt[i]);
   }
   int temp1, temp2, temp3;
   for (int i=0; i<n; i++)
   {
      for (int j=i+1; j<n; j++)
      {
         if (at[j]<at[i])
         {
            temp1 = at[j];
            at[j] = at[i];
            at[i] = temp1;

            temp2 = bt[j];
            bt[j] = bt[i];
            bt[i] = temp2;

            temp3 = pr[j];
            pr[j] = pr[i];
            pr[i] = temp3;
         }
      }
   }
   int x=at[0];
```

```
    for (int i=0; i<n; i++)
    {
        if (x<at[i])
        {
            x = at[i];
        }
        ct[i] = bt[i] + x;
        x = ct[i];
    }
    for (int i=0; i<n; i++)
    {
        tat[i] = ct[i] - at[i];
    }
    for (int i=0; i<n; i++)
    {
        wt[i] = tat[i] - bt[i];
    }
    for (int i=0; i<n; i++)
    {
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", pr[i], at[i], bt[i], ct[i], tat[i], wt[i]);
    }
    float avg_tat = 0, avg_wt = 0;
    for (int i=0; i<n; i++)
    {
        avg_tat = avg_tat + tat[i];
        avg_wt = avg_wt + wt[i];
    }
    avg_tat = avg_tat/n;
    avg_wt = avg_wt/n;
    printf("The average Turnaround time is: %f", avg_tat);
    printf("\nThe average Waiting time is: %f`", avg_wt);
}
```

Result:

```
4
Enter Arrival Time:
0 1 5 6
Enter Burst Time:
2 2 3 4
1       0       2       2       2       0
2       1       2       4       3       1
3       5       3       8       3       0
4       6       4       12      6       2
The average Turnaround time is: 3.500000
The average Waiting time is: 0.750000`
```

2.SJF(Pre-emptive)

```
#include <stdio.h>
void findCompletionTime(int processes[], int n, int bt[], int at[], int wt[], int tat[], int ct[])
{
   int remaining[n];
   int currentTime = 0;
   int completed = 0;
   for (int i = 0; i < n; i++)
      remaining[i] = bt[i];
   while (completed < n)
   {
      int shortest = -1;
      for (int i = 0; i < n; i++)
      {
         if (at[i] <= currentTime && remaining[i] > 0)
         {
            if (shortest == -1 || remaining[i] <= remaining[shortest])
               shortest = i;
         }
      }
       if (shortest == -1)
      {
         currentTime++;
         continue;
      }
      remaining[shortest]--;
      if (remaining[shortest] == 0)
      {
         completed++;
         ct[shortest] = currentTime + 1;
         wt[shortest] = ct[shortest] - bt[shortest] - at[shortest];
         tat[shortest] = ct[shortest] - at[shortest];
      }
      currentTime++;
   }
   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", processes[i], at[i], bt[i], ct[i], tat[i], wt[i]);
   }
   float avg_tat = 0, avg_wt = 0;
   for (int i = 0; i < n; i++)
   {
      avg_tat += tat[i];
      avg_wt += wt[i];
   }
   avg_tat /= n;
   avg_wt /= n;
   printf("The average Turnaround time is %f\n", avg_tat);
   printf("The average Waiting time is %f\n", avg_wt);
}
```

```
void main()
{
   int n;
   printf("Enter the number of processes: ");
   scanf("%d", &n);
   int processes[n];
   int burst_time[n];
   int arrival_time[n];
   printf("Enter Process Number:\n");
   for (int i = 0; i < n; i++)
   {
      scanf("%d", &processes[i]);
   }
   printf("Enter Arrival Time:\n");
   for (int i = 0; i < n; i++)
   {
      scanf("%d", &arrival_time[i]);
   }
   printf("Enter Burst Time:\n");
   for (int i = 0; i < n; i++)
   {
      scanf("%d", &burst_time[i]);
   }
   int wt[n], tat[n], ct[n];
   printf("\nSJF (Preemptive) Scheduling:\n");
   findCompletionTime(processes, n, burst_time, arrival_time, wt, tat, ct);
}
```

Result:

```
Enter the number of processes: 5
Enter Process Number:
1 2 3 4 5
Enter Arrival Time:
2 1 4 0 2
Enter Burst Time:
1 5 1 6 3

SJF (Preemptive) Scheduling:
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
The average Turnaround time is 6.600000
The average Waiting time is 3.400000
```

3.SJF(NON-pre-emptive)

```
#include<stdio.h>
void findCompletionTime(int processes[], int n, int bt[], int at[], int wt[], int tat[], int rt[], int ct[])
{
   int completion[n]; // Array to store completion times of processes
   int remaining[n]; // Array to store remaining burst time of processes

   // Initialize remaining array with burst times
   for (int i = 0; i < n; i++)
      remaining[i] = bt[i];

   int currentTime = 0; // Current time

   // Find process with shortest burst time
   for (int i = 0; i < n; i++)
   {
      int shortest = -1;
      for (int j = 0; j < n; j++)
      {
         if (at[j] <= currentTime && remaining[j] > 0)
         {
            if (shortest == -1 || remaining[j] < remaining[shortest])
               shortest = j;
         }
      }

      if (shortest == -1)
      {
         currentTime++;
         continue;
      }
      completion[shortest] = currentTime + remaining[shortest];
      currentTime = completion[shortest];
      wt[shortest] = currentTime - bt[shortest] - at[shortest];
      tat[shortest] = currentTime - at[shortest];
      rt[shortest] = wt[shortest]; // Response time for non-preemptive SJF is the same as waiting time
      remaining[shortest] = 0;
   }
   for (int i = 0; i < n; i++)
   {
      ct[i] = completion[i];
      printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i], at[i], bt[i], ct[i], wt[i], tat[i], rt[i]);
   }
   float avg_tat = 0, avg_wt = 0;
   for (int i = 0; i < n; i++)
   {
      avg_tat += tat[i];
      avg_wt += wt[i];
   }
   avg_tat /= n;
```

```
    avg_wt /= n;
    printf("The average Turnaround time is %f\n", avg_tat);
    printf("The average Waiting time is %f\n", avg_wt);
}

void main()
{
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int processes[n];
    int burst_time[n];
    int arrival_time[n];
    printf("Enter Process Number:\n");
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &processes[i]);
    }
    printf("Enter Arrival Time:\n");
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &arrival_time[i]);
    }
    printf("Enter Burst Time:\n");
    for (int i = 0; i < n; i++)
    {
        scanf("%d", &burst_time[i]);
    }
    int wt[n], tat[n], rt[n], ct[n];
    for (int i = 0; i < n; i++)
        rt[i] = -1;
    printf("\nSJF (Non-preemptive) Scheduling:\n");
    findCompletionTime(processes, n, burst_time, arrival_time, wt, tat, rt, ct);
}
```

Result:

```
Enter the number of processes: 5
Enter Process Number:
1 2 3 4 5
Enter Arrival Time:
2 1 4 0 2
Enter Burst Time:
1 5 1 6 3

SJF (Non-preemptive) Scheduling:
1               2               1               7               4               5               4
2               1               5               16              10              15              10
3               4               1               8               3               4               3
4               0               6               6               0               6               0
5               2               3               11              6               9               6
The average Turnaround time is 7.800000
The average Waiting time is 4.600000
```

Program -2

Question: Write a C program to simulate the following CPU scheduling algorithm to findturnaround time and waiting time. → Priority (pre-emptive & Non-pre-emptive)→Round Robin (Experiment with different quantum sizes for RR algorithm)

Code:
1.Priority(Pre-emptive)

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

// Function to find the waiting time, turnaround time, and completion time for all processes using Priority Scheduling (Preemptive)
void findCompletionTime(int processes[], int n, int bt[], int at[], int wt[], int tat[], int ct[], int rt[], int priority[], bool isLowerPriorityHigher)
{
   int remaining[n]; // Array to store remaining burst time of processes
   int currentTime = 0; // Current time
   int completed = 0; // Counter for completed processes
   bool isFinished[n]; // Array to indicate if the process is finished

   // Initialize remaining array with burst times and set response times
   for (int i = 0; i < n; i++) {
      remaining[i] = bt[i];
      isFinished[i] = false;
      rt[i] = -1; // Response time is initially unset
   }

   while (completed < n) {
      int highestPriorityIndex = -1;
      int highestPriority = isLowerPriorityHigher ? 1000000 : -1; // Adjust initial value based on priority type

      // Find the process with the highest priority that has arrived and is not finished
      for (int i = 0; i < n; i++) {
         if (at[i] <= currentTime && !isFinished[i] &&
            ((isLowerPriorityHigher && priority[i] < highestPriority) ||
             (!isLowerPriorityHigher && priority[i] > highestPriority))) {
            highestPriority = priority[i];
            highestPriorityIndex = i;
         }
      }

      // If no process is found, move to the next time
      if (highestPriorityIndex == -1) {
         currentTime++;
         continue;
      }
```

```
        int currentProcess = highestPriorityIndex;

        // Set response time if it's the first time the process is executed
        if (rt[currentProcess] == -1) {
            rt[currentProcess] = currentTime - at[currentProcess];
        }

        // Execute the process for 1 unit of time
        remaining[currentProcess]--;
        currentTime++;

        // If the process is completed
        if (remaining[currentProcess] == 0) {
            isFinished[currentProcess] = true;
            completed++;
            ct[currentProcess] = currentTime; // Set completion time for the process
            tat[currentProcess] = ct[currentProcess] - at[currentProcess]; // Calculate turnaround time
            wt[currentProcess] = tat[currentProcess] - bt[currentProcess]; // Calculate waiting time
        }
    }

    // Print the table
    printf("Process\tArrival Time\tBurst Time\tPriority\tCompletion Time\tTurnaround Time\tWaiting
Time\tResponse 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\t\t%d\t\t%d\n",
               processes[i], at[i], bt[i], priority[i], ct[i], tat[i], wt[i], rt[i]);
    }
}

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

    // Process id's
    int processes[n];
    // Burst time of all processes
    int burst_time[n];
    // Arrival time of all processes
    int arrival_time[n];
    // Priority of all processes
    int priority[n];
    // Priority type (true for lower number = higher priority, false for higher number = higher priority)
    int priorityType;
    bool isLowerPriorityHigher;
```

```
    printf("Enter 1 if lower number indicates higher priority, 0 if higher number indicates higher priority: ");
    scanf("%d", &priorityType);
    isLowerPriorityHigher = (priorityType == 1);

    printf("Enter Process Number:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &processes[i]);
    }
    printf("Enter Priority:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &priority[i]);
    }
    printf("Enter Arrival Time:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arrival_time[i]);
    }
    printf("Enter Burst Time:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &burst_time[i]);
    }

    // Arrays to store waiting time, turnaround time, completion time, and response time
    int wt[n], tat[n], ct[n], rt[n];

    printf("\nPriority Scheduling (Preemptive):\n");
    findCompletionTime(processes, n, burst_time, arrival_time, wt, tat, ct, rt, priority, isLowerPriorityHigher);
}
```

Result:

```
Enter the number of processes: 4
Enter 1 if lower number indicates higher priority, 0 if higher number indicates higher priority: 0
Enter Process Number:
1 2 3 4
Enter Priority:
10 20 30 40
Enter Arrival Time:
0 1 2 4
Enter Burst Time:
5 4 2 1

Priority Scheduling (Preemptive):
Process Arrival Time    Burst Time      Priority        Completion Time Turnaround Time  Waiting Time    Response Time
1               0               5               10              12              12              7               0
2               1               4               20              8               7       3               0
3               2               2               30              4               2       0               0
4               4               1               40              5               1       0               0
```

2.Priority(Non-pre-emptive)

```
#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);
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);
}
```

Result:

```
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
```

3.Round Robin

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

void findCompletionTime(int processes[], int n, int bt[], int at[], int wt[], int tat[], int ct[], int rt[], int quantum)
{
   int remaining[n]; // Array to store remaining burst time of processes
   bool firstResponse[n]; // Array to track if response time has been set
   int currentTime = 0; // Current time
   int completed = 0; // Counter for completed processes

   // Initialize remaining array with burst times and first response array
   for (int i = 0; i < n; i++) {
      remaining[i] = bt[i];
      firstResponse[i] = true;
   }

   // Queue to hold the indices of the processes
   int queue[n];
   int front = -1, rear = -1;

   // Function to add process to the queue
   void enqueue(int process) {
      if (rear == n - 1) rear = -1;
      queue[++rear] = process;
      if (front == -1)
         front = 0;
   }

   // Function to remove process from the queue
   int dequeue() {
      int process = queue[front];
      if (front == rear)
         front = rear = -1;
      else {
         front++;
         if (front == n)
            front = 0;
      }
      return process;
   }

   // To track which processes have been added to the queue
   bool inQueue[n];
   for (int i = 0; i < n; i++)
      inQueue[i] = false;

   while (completed < n) {
```

```
// Add all processes to the queue that have arrived by the current time
for (int i = 0; i < n; i++) {
   if (at[i] <= currentTime && !inQueue[i]) {
      enqueue(i);
      inQueue[i] = true;
   }
}

// If no process is ready, increment the current time
if (front == -1) {
   currentTime++;
   continue;
}

int currentProcess = dequeue();

// Set response time if it's the first time the process is executed
if (firstResponse[currentProcess]) {
   rt[currentProcess] = currentTime - at[currentProcess];
   firstResponse[currentProcess] = false;
}

// Execute the process for the time quantum or until completion
if (remaining[currentProcess] > quantum) {
   remaining[currentProcess] -= quantum;
   currentTime += quantum;
} else {
   currentTime += remaining[currentProcess];
   remaining[currentProcess] = 0;
   completed++;
   // Set completion time for the process
   ct[currentProcess] = currentTime;
   // Calculate waiting time and turnaround time for the process
   tat[currentProcess] = ct[currentProcess] - at[currentProcess];
   wt[currentProcess] = tat[currentProcess] - bt[currentProcess];
}

// Add all processes to the queue that have arrived by the current time
for (int i = 0; i < n; i++) {
   if (at[i] <= currentTime && !inQueue[i]) {
      enqueue(i);
      inQueue[i] = true;
   }
}

// Re-enqueue the current process if it is not finished
if (remaining[currentProcess] > 0) {
   enqueue(currentProcess);
}
```

```
    }

    // Print the table
    printf("Process\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting Time\tResponse
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\t\t%d\n",
                processes[i], at[i], bt[i], ct[i], tat[i], wt[i], rt[i]);
    }
}

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

    // Process id's
    int processes[n];
    // Burst time of all processes
    int burst_time[n];
    // Arrival time of all processes
    int arrival_time[n];

    printf("Enter Process Number:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &processes[i]);
    }
    printf("Enter Arrival Time:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arrival_time[i]);
    }
    printf("Enter Burst Time:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &burst_time[i]);
    }

    // Time quantum for Round Robin
    int quantum;
    printf("Enter the time quantum: ");
    scanf("%d", &quantum);

    // Arrays to store waiting time, turnaround time, completion time, and response time
    int wt[n], tat[n], ct[n], rt[n];

    printf("\nRound Robin Scheduling:\n");
    findCompletionTime(processes, n, burst_time, arrival_time, wt, tat, ct, rt, quantum);
}
```

Result:

```
Enter the number of processes: 5
Enter Process Number:
1 2 3 4 5
Enter Arrival Time:
0 1 2 3 4
Enter Burst Time:
5 3 1 2 3
Enter the time quantum: 2

Round Robin Scheduling:
Process Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time    Response Time
1               0               5               13              13              8               0
2               1               3               12              11              8               1
3               2               1               5               3               2               2
4               3               2               9               6               4               4
5               4               3               14              10              7               5
```

```
Enter the number of processes: 5
Enter Process Number:
1 2 3 4 5
Enter Arrival Time:
0 1 2 3 4
Enter Burst Time:
5 3 1 2 3
Enter the time quantum: 3

Round Robin Scheduling:
Process Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time    Response Time
1               0               5               11              11              6               0
2               1               3               6               5               2               2
3               2               1               7               5               4               4
4               3               2               9               6               4               4
5               4               3               14              10              7               7
```

Program -3

Question: Write a C program to simulate multi-level queue scheduling algorithm considering thefollowing 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.

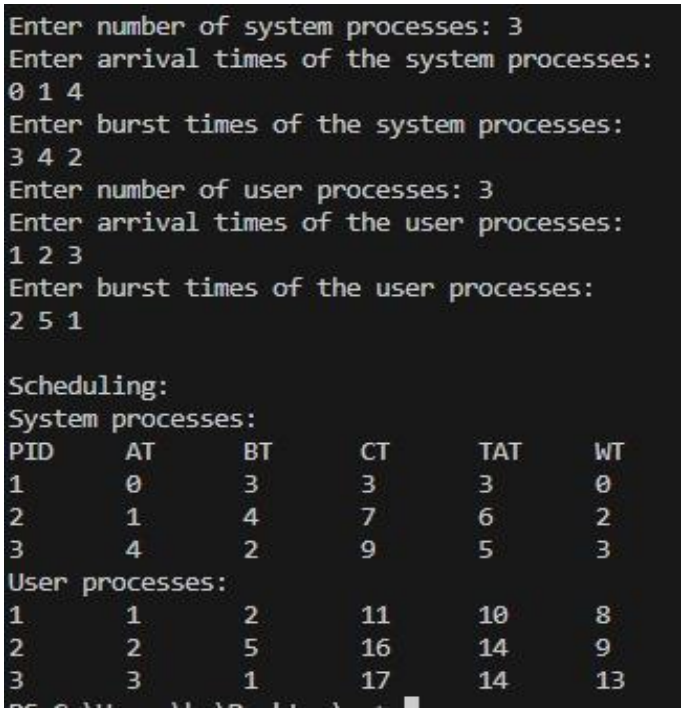
Code:

```
#include<stdio.h>
void sort(int proc_id[],int at[],int bt[],int n)
{
 int temp=0;
 for(int i=0;i<n;i++)
 {
 for(int j=i;j<n;j++)
 {
 if(at[j]<at[i])
 {
 temp=at[i];at[i]=at[j];at[j]=temp;
 temp=bt[j];bt[j]=bt[i];bt[i]=temp;
 temp=proc_id[i];proc_id[i]=proc_id[j];proc_id[j]=temp;
 }
 }
 }
}
void fcfs(int at[],int bt[],int ct[],int tat[],int wt[],int n,int *c)
{
 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]-ct[i-1]+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];
}
void main()
{
 int sn,un,c=0;int n=0;
 printf("Enter number of system processes: ");
 scanf("%d",&sn);n=sn;
 int sproc_id[n],sat[n],sbt[n],sct[n],stat[n],swt[n];
 for(int i=0;i<sn;i++)
 sproc_id[i]=i+1;
```

```
 printf("Enter arrival times of the system processes:\n");
 for(int i=0;i<sn;i++)
 scanf("%d",&sat[i]);
 printf("Enter burst times of the system processes:\n");
 for(int i=0;i<sn;i++)
 scanf("%d",&sbt[i]);

 printf("Enter number of user processes: ");
 scanf("%d",&un);n=un;
 int uproc_id[n],uat[n],ubt[n],uct[n],utat[n],uwt[n];
 for(int i=0;i<un;i++)
 uproc_id[i]=i+1;
 printf("Enter arrival times of the user processes:\n");
 for(int i=0;i<un;i++)
 scanf("%d",&uat[i]);
 printf("Enter burst times of the user processes:\n");
 for(int i=0;i<un;i++)
 scanf("%d",&ubt[i]);
 sort(sproc_id,sat,sbt,sn);
 sort(uproc_id,uat,ubt,un);
 fcfs(sat,sbt,sct,stat,swt,sn,&c);
 fcfs(uat,ubt,uct,utat,uwt,un,&c);
 printf("\nScheduling:\n");
 printf("System processes:\n");
 printf("PID\tAT\tBT\tCT\tTAT\tWT\n");
 for(int i=0;i<sn;i++)
 printf("%d\t%d\t%d\t%d\t%d\t%d\n",sproc_id[i],sat[i],sbt[i],sct[i],stat[i],swt[i]);
 printf("User processes:\n");
 for(int i=0;i<un;i++)
 printf("%d\t%d\t%d\t%d\t%d\t%d\n",uproc_id[i],uat[i],ubt[i],uct[i],utat[i],uwt[i]);
}
```

Result:



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:

1.Rate-Monotonic

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
void sort(int proc[], int b[], int pt[], int n) {
   for (int i = 0; i < n; i++) {
      for (int j = i; j < n; j++) {
         if (pt[j] < pt[i]) {
            int temp = proc[i];
            proc[i] = proc[j];
            proc[j] = temp;
            temp = b[i];
            b[i] = b[j];
            b[j] = temp;
            temp = pt[i];
            pt[i] = pt[j];
            pt[j] = temp;
         }}}}
int lcmul(int p[], int n) {
   int lcm = p[0];
   for (int i = 1; i < n; i++) {
      int a = lcm, b = p[i];
      while (b != 0) {
         int r = a % b;
         a = b;
```

```
        b = r;}
      lcm = (lcm * p[i]) / a;}
   return lcm;}
int main() {
   int n;
   printf("Enter the number of processes: ");
   scanf("%d", &n);
   int proc[n], b[n], pt[n], rem[n];
   printf("Enter the CPU burst times:\n");
   for (int i = 0; i < n; i++) scanf("%d", &b[i]);
   printf("Enter the time periods:\n");
   for (int i = 0; i < n; i++) scanf("%d", &pt[i]);
   for (int i = 0; i < n; i++) proc[i] = i + 1;
   sort(proc, b, pt, n);
   int l = lcmul(pt, n);
   printf("\nRate Monotonic Scheduling:\n");
   printf("PID\t Burst\tPeriod\n");
   for (int i = 0; i < n; i++) printf("%d\t\t%d\t\t%d\n", proc[i], b[i], pt[i]);
   double sum = 0.0;
   for (int i = 0; i < n; i++) {
      sum += (double)b[i] / pt[i];
   }
   double rhs = n * (pow(2.0, (1.0 / n)) - 1.0);
   printf("\n%lf <= %lf => %s\n", sum, rhs, (sum <= rhs) ? "true" : "false");
   if (sum > rhs) {
      printf("The given set of processes is not schedulable.\n");
      exit(0);
   }
   printf("Scheduling occurs for %d ms\n\n", l);
   int time = 0, prev = -1;
   for (int i = 0; i < n; i++) rem[i] = b[i];
   int nextRelease[n];
   for (int i = 0; i < n; i++) nextRelease[i] = 0;
   while (time < l) {
      int taskToExecute = -1;
      for (int i = 0; i < n; i++) {
         if (time == nextRelease[i]) {
            rem[i] = b[i];  // Reset remaining time at the start of the period
            nextRelease[i] += pt[i];  // Schedule next release
         }
         if (rem[i] > 0 && (taskToExecute == -1 || pt[i] < pt[taskToExecute])) {
            taskToExecute = i;}}
     if (taskToExecute != -1) {
         if (prev != taskToExecute) {
            printf("%dms: Task %d is running.\n", time, proc[taskToExecute]);
            prev = taskToExecute;          }
         rem[taskToExecute]--;
      } else if (prev != -1) {
         printf("%dms: CPU is idle.\n", time);
         prev = -1;       }
      time++;    }    return 0;}
```

Result:

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

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

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

0ms: Task 2 is running.
2ms: Task 3 is running.
4ms: Task 1 is running.
5ms: Task 2 is running.
7ms: Task 1 is running.
9ms: CPU is idle.
10ms: Task 2 is running.
12ms: Task 3 is running.
14ms: CPU is idle.
15ms: Task 2 is running.
17ms: CPU is idle.

Process returned 0 (0x0)   execution time : 42.111 s
Press any key to continue.
```

2.Earliest Deadline First

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

```
int
gcd (int a, int b)
{
  int r;
  while (b > 0)
            {
              r = a % b;
              a = b;
              b = r;
            }
  return a;
}
int
lcmul (int p[], int n)
{
  int lcm = p[0];
  for (int i = 1; i < n; i++)
            {
              lcm = (lcm * p[i]) / gcd (lcm, p[i]);
            }
  return lcm;
}
Void main ()
{
  int n;
  printf ("Enter the number of processes:");
  scanf ("%d", &n);
  int proc[n], b[n], pt[n], d[n], rem[n];
  printf ("Enter the CPU burst times:\n");
  for (int i = 0; i < n; i++)
            {
              scanf ("%d", &b[i]);
              rem[i] = b[i];
            }
  printf ("Enter the deadlines:\n");
  for (int i = 0; i < n; i++)
            scanf ("%d", &d[i]);
  printf ("Enter the time periods:\n");
  for (int i = 0; i < n; i++)
            scanf ("%d", &pt[i]);
  for (int i = 0; i < n; i++)
            proc[i] = i + 1;
  sort (proc, d, b, pt, n);
  int l = lcmul (pt, n);
  printf ("\nEarliest Deadline Scheduling:\n");
  printf ("PID\t Burst\tDeadline\tPeriod\n");
  for (int i = 0; i < n; i++)
            printf ("%d\t\t%d\t\t%d\t\t%d\n", proc[i], b[i], d[i], pt[i]);
  printf ("Scheduling occurs for %d ms\n\n", l);
  int time = 0, prev = 0, x = 0;
  int nextDeadlines[n];
  for (int i = 0; i < n; i++)
```

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

Result:

```
Enter the number of processes:3
Enter the CPU burst times:
3 2 2
Enter the deadlines:
7 4 8
Enter the time periods:
20 5 10

Earliest Deadline Scheduling:
PID      Burst  Deadline        Period
2               2               4               5
1               3               7               20
3               2               8               10
Scheduling occurs for 20 ms

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

3.Proportional

```
#include<stdio.h>
#include<stdlib.h>
void sort_by_burst(int proc[], int burst[], int tickets[], int remaining[], int n) {
   for (int i = 0; i < n - 1; i++) {
      for (int j = 0; j < n - i - 1; j++) {
         if (burst[j] > burst[j + 1]) {
            // Swap process IDs
            int temp = proc[j];
            proc[j] = proc[j + 1];
            proc[j + 1] = temp;

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

            // Swap tickets
            temp = tickets[j];
            tickets[j] = tickets[j + 1];
            tickets[j + 1] = temp;

            // Swap remaining burst times
            temp = remaining[j];
            remaining[j] = remaining[j + 1];
            remaining[j + 1] = temp;
         }
      }
   }
}

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

   int proc[n], burst[n], tickets[n], remaining[n];
   int total_tickets = 0;

   printf("Enter the CPU burst times and number of tickets for each process:\n");
   for (int i = 0; i < n; i++) {
      proc[i] = i + 1;  // Process IDs
      printf("Process %d burst time: ", i + 1);
      scanf("%d", &burst[i]);
      printf("Process %d tickets: ", i + 1);
      scanf("%d", &tickets[i]);
      remaining[i] = burst[i];
      total_tickets += tickets[i];
   }

   sort_by_burst(proc, burst, tickets, remaining, n);
```

```
    printf("\nProportional Share Scheduling (Lottery Scheduling):\n");
    printf("PID\t Burst\tTickets\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t %d\t %d\n", proc[i], burst[i], tickets[i]);
    }

    int time = 0;
    while (1) {
        int active_processes = 0;
        for (int i = 0; i < n; i++) {
            if (remaining[i] > 0) {
                active_processes++;
            }
        }

        if (active_processes == 0) {
            break;
        }

        int winning_ticket = rand() % total_tickets;
        int cumulative_tickets = 0;
        int selected_process = -1;

        for (int i = 0; i < n; i++) {
            if (remaining[i] > 0) {
                cumulative_tickets += tickets[i];
                if (winning_ticket < cumulative_tickets) {
                    selected_process = i;
                    break;
                }
            }
        }
        if (selected_process != -1) {
            printf("%dms: Process %d is running.\n", time, proc[selected_process]);
            remaining[selected_process]--;
        } else {
            printf("%dms: CPU is idle.\n", time);
        }

        time++;
    }

    printf("Scheduling completed.\n");

    return 0;
}
```

Result:

```
PS C:\Users\hp\Desktop\os> ./a.exe
Enter the number of processes: 3
Enter the CPU burst times and number of tickets for each process:
Process 1 burst time: 2
Process 1 tickets: 10
Process 2 burst time: 4
Process 2 tickets: 30
Process 3 burst time: 5
Process 3 tickets: 20

Proportional Share Scheduling (Lottery Scheduling):
PID     Burst  Tickets
1       2      10
2       4      30
3       5      20
0ms: Process 3 is running.
1ms: Process 3 is running.
2ms: Process 2 is running.
3ms: Process 3 is running.
4ms: Process 2 is running.
5ms: Process 1 is running.
6ms: Process 2 is running.
7ms: Process 2 is running.
8ms: Process 3 is running.
9ms: CPU is idle.
10ms: Process 1 is running.
11ms: Process 3 is running.
Scheduling completed.
```

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 = 7, x = 0;
int main() {
   int n;
   void producer();
   void consumer();
   int wait(int);
   int signal(int);
   printf("\n1.Producer\n2.Consumer\n3.Exit");
   while (1) {
      printf("\nEnter your choice:");
      scanf("%d", &n);
      switch (n) {
         case 1:
            if ((mutex == 1) && (empty != 0))
               producer();
            else
               printf("Buffer is full!!");
            break;
         case 2:
            if ((mutex == 1) && (full != 0))
               consumer();
            else
               printf("Buffer is empty!!");
            break;
         case 3:
            printf("\nNumber of Items remaining in buffer: %d\n", x);
            exit(0);
            break;
      }
   }
   return 0;
}

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

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

void producer() {
   mutex = wait(mutex);
   full = signal(full);
```

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

void consumer() {
    mutex = wait(mutex);
    full = wait(full);
    empty = signal(empty);
    printf("\nConsumer consumes item %d", x);
    x--;
    mutex = signal(mutex);
}
```

Result:

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

Producer produces the item 1
Enter your choice:1

Producer produces the item 2
Enter your choice:2

Consumer consumes item 2
Enter your choice:2

Consumer consumes item 1
Enter your choice:2
Buffer is empty!!
Enter your choice:3

Number of Items remaining in buffer: 0
```

Program -6

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

Code:

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

#define THINKING 0
#define HUNGRY 1
#define EATING 2
#define N 5
#define MAX_PHILOSOPHERS 5
bool spoon[N] = {true, true, true, true, true}; // true means the spoon is available
int state[N] = {THINKING, THINKING, THINKING, THINKING, THINKING};

bool wait(int i)
{
   return !spoon[i]; // if spoon is not available, return true
}

void signal(int i)
{
   spoon[i] = true; // make the spoon available
}

void take_spoon(int i)
{
   spoon[i] = false; // take the spoon
}

void test(int i)
{
   if (state[i] == HUNGRY && !wait(i) && !wait((i + 1) % N))
   {
      state[i] = EATING;
      take_spoon(i);
      take_spoon((i + 1) % N);
      printf("Philosopher %d is Granted to Eat\n", i + 1);
   }
}

void put_spoon(int i)
{
   signal(i);
   signal((i + 1) % N);
   state[i] = THINKING;
   printf("Philosopher %d is Waiting\n", i + 1);
}

int completed()
{
   for (int i = 0; i < N; i++)
   {
      if (state[i] != THINKING)
```

```
        return 0;
    }
    printf("Dinner completed\n");
    return 1;
}

void allow_one_to_eat(int hungry[], int n)
{
    for (int i = 0; i < n; i++)
    {
        state[hungry[i]-1]=HUNGRY;
    }
    for (int i = 0; i < n; i++)
    {
        int pos = hungry[i] - 1;
        test(pos);
        if (state[pos] == EATING)
        {
            put_spoon(pos);
        }
        for (int j = 0; j < n; j++)
        {
            if (state[hungry[j] - 1] == HUNGRY)
            {
                printf("Philosopher %d is Waiting\n", hungry[j]);
            }
        }
    }
}

void allow_two_to_eat(int hungry[], int n)
{
    if (n < 2 || n > MAX_PHILOSOPHERS)
    {
        printf("Invalid number of philosophers.\n");
        return;
    }
    else if(n==2)
    {
        if(abs(hungry[0]-hungry[1])==1)
        {
            printf("Not possible");
            exit(0);
        }
        printf("P %d and P %d are granted to eat\n", hungry[0], hungry[1]);
    }
    else{
int combination_count = 1,a[n-2];

    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
              int p=0;

        if(abs(hungry[i]-hungry[j])==1)
        {
            continue;
        }
            printf("\n\n\ncombination %d\n\n", combination_count);
```

```
        printf("P %d and P %d are granted to eat\n", hungry[i], hungry[j]);

        for (int k = 0; k < n; k++) {
           if (k != i && k != j) {
              printf("P %d is waiting\n", hungry[k]);
              a[p]=k;
              p++;
           }
        }
        printf("\n");
        printf("P %d is waiting\n", hungry[i]);
        printf("P %d is waiting\n", hungry[j]);
        int com=1;
        if(abs(a[0]-a[1])!=1 && n>3)
        {
              if(n==5)
              {
                 printf("Combination %d.%d\n",combination_count,com);
              }
              printf("P %d and P %d are granted to eat\n", hungry[a[0]], hungry[a[1]]);
              if(n==5)
              {
                 printf("P %d is waiting\n", hungry[a[2]]);
              }
              printf("P %d is waiting\n", hungry[a[0]]);
              printf("P %d is waiting\n", hungry[a[1]]);
              com++;
        }
        if(abs(a[2]-a[1])!=1 && n>4)
        {
              printf("Combination %d.%d\n",combination_count,com);
              printf("P %d and P %d are granted to eat\n", hungry[a[2]], hungry[a[1]]);
              printf("P %d is waiting\n", hungry[a[0]]);
              printf("P %d is waiting\n", hungry[a[1]]);
              printf("P %d is waiting\n", hungry[a[2]]);
              com++;
        }
        if(abs(a[0]-a[2])!=1 && n>4)
        {
              printf("Combination %d.%d\n",combination_count,com);
              printf("P %d and P %d are granted to eat\n", hungry[a[0]], hungry[a[2]]);
              printf("P %d is waiting\n", hungry[a[1]]);
              printf("P %d is waiting\n", hungry[a[0]]);
              printf("P %d is waiting\n", hungry[a[2]]);

        }
        combination_count++;
     }
   }
   }
}

int main()
{
   while (1)
   {
      int total_philosophers, hungry_count;
```

```
    int hungry_positions[MAX_PHILOSOPHERS];

    printf("DINING PHILOSOPHER PROBLEM\n");
    printf("Enter the total no. of philosophers: %d\n", N);
    total_philosophers = N;

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

    if (hungry_count < 1 || hungry_count > total_philosophers)
    {
      printf("Invalid number of hungry philosophers.\n");
      return 1;
    }

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

      if (hungry_positions[i] < 1 || hungry_positions[i] > total_philosophers)
      {
        printf("Invalid philosopher position.\n");
        return 1;
      }
    }

    int choice;
    printf("\n1. One can eat at a time\n");
    printf("2. Two can eat at a time\n");
    printf("3. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice)
    {
      case 1:
        printf("Allow one philosopher to eat at any time\n");
        allow_one_to_eat(hungry_positions, hungry_count);
        break;
      case 2:
        printf("Allow two philosophers to eat at the same time\n");
        allow_two_to_eat(hungry_positions, hungry_count);
        break;
      case 3:
        exit(0);
      default:
        printf("Invalid choice\n");
    }
  }

  return 0;
}
```

Result:

```
DINING PHILOSOPHER PROBLEM
Enter the total no. of philosophers: 5
How many are hungry: 3
Enter philosopher 1 position: 2
Enter philosopher 2 position: 4
Enter philosopher 3 position: 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 2 is Granted to Eat
Philosopher 2 is Waiting
Philosopher 4 is Waiting
Philosopher 5 is Waiting
Philosopher 4 is Granted to Eat
Philosopher 4 is Waiting
Philosopher 5 is Waiting
Philosopher 5 is Granted to Eat
Philosopher 5 is Waiting
DINING PHILOSOPHER PROBLEM
Enter the total no. of philosophers: 5
How many are hungry: 3
Enter philosopher 1 position: 2
Enter philosopher 2 position: 4
Enter philosopher 3 position: 5

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
```

```
Allow two philosophers to eat at the same time



combination 1

P 2 and P 4 are granted to eat
P 5 is waiting

P 2 is waiting
P 4 is waiting



combination 2

P 2 and P 5 are granted to eat
P 4 is waiting

P 2 is waiting
P 5 is waiting
```

Program -7

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

Code:

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

#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

void calculateNeed(int need[MAX_PROCESSES][MAX_RESOURCES], int
max[MAX_PROCESSES][MAX_RESOURCES], int allot[MAX_PROCESSES][MAX_RESOURCES], int np, int nr) {
   for (int i = 0; i < np; i++)
      for (int j = 0; j < nr; j++)
         need[i][j] = max[i][j] - allot[i][j];
}

bool isSafe(int processes[], int avail[], int max[][MAX_RESOURCES], int allot[][MAX_RESOURCES], int np, int nr) {
   int need[MAX_PROCESSES][MAX_RESOURCES];
   calculateNeed(need, max, allot, np, nr);

   bool finish[MAX_PROCESSES] = {0};
   int safeSeq[MAX_PROCESSES];
   int work[MAX_RESOURCES];

   for (int i = 0; i < nr; i++)
      work[i] = avail[i];

   int count = 0;
   while (count < np) {
```

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

            if (j == nr) {
               for (int k = 0; k < nr; k++)
                  work[k] += allot[p][k];

               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.\nSafe sequence is: ");
   for (int i = 0; i < np; i++)
      printf("%d ", safeSeq[i]);
   printf("\n");

   return true;
}

void printResourceAllocationDetails(int np, int nr, int processes[], int max[][MAX_RESOURCES], int
allot[][MAX_RESOURCES], int avail[]) {
   printf("\nProcess\t\tAllocation\tMax\tNeed\tAvailable\n");
   for (int i = 0; i < np; i++) {
      printf("%d\t", processes[i]);

      // Print Allocation
      printf("\t");
      for (int j = 0; j < nr; j++)
         printf("%d ", allot[i][j]);

      // Print Max
      printf("\t\t");
      for (int j = 0; j < nr; j++)
         printf("%d ", max[i][j]);


      // Print Need
      printf("\t");
      for (int j = 0; j < nr; j++)
```

```
            printf("%d ", max[i][j] - allot[i][j]);

        // Print Available
        if (i == 0) {
            printf("\t");
            for (int j = 0; j < nr; j++)
                printf("%d ", avail[j]);
        }



        printf("\n");
    }
}

int main() {
    int np, nr;
    printf("Enter number of processes: ");
    scanf("%d", &np);
    printf("Enter number of resource types: ");
    scanf("%d", &nr);
    int processes[MAX_PROCESSES];
    for (int i = 0; i < np; i++) processes[i] = i;
    int avail[MAX_RESOURCES];
    printf("Enter available resources: ");
    for (int i = 0; i < nr; i++)
        scanf("%d", &avail[i]);
    int max[MAX_PROCESSES][MAX_RESOURCES];
    printf("Enter maximum resource matrix:\n");
    for (int i = 0; i < np; i++) {
        printf("Process %d: ", i);
        for (int j = 0; j < nr; j++)
            scanf("%d", &max[i][j]);
    }

    int allot[MAX_PROCESSES][MAX_RESOURCES];
    printf("Enter allocation resource matrix:\n");
    for (int i = 0; i < np; i++) {
        printf("Process %d: ", i);
        for (int j = 0; j < nr; j++)
            scanf("%d", &allot[i][j]);
    }
    printResourceAllocationDetails(np, nr, processes, max, allot, avail);
    isSafe(processes, avail, max, allot, np, nr);
    return 0;
}
```

Result:

```
Enter number of processes: 5 3
Enter number of resource types: Enter available resources: 3 3 2
Enter maximum resource matrix:
Process 0: 7 5 3
Process 1: 3 2 2
Process 2: 9 0 2
Process 3: 2 2 2
Process 4: 4 3 3
Enter allocation resource 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

Process         Allocation      Max     Need    Available
0               0 1 0           7 5 3   7 4 3   3 3 2
1               2 0 0           3 2 2   1 2 2
2               3 0 2           9 0 2   6 0 0
3               2 1 1           2 2 2   0 1 1
4               0 0 2           4 3 3   4 3 1
System is in safe state.
Safe sequence is: 1 3 4 0 2
```

Program -8

Question: Write a C program to simulate deadlock detection

Code:

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

#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

bool isSafe(int processes[], int avail[], int req[][MAX_RESOURCES], int allot[][MAX_RESOURCES], int np, int nr) {
   int need[MAX_PROCESSES][MAX_RESOURCES];


   bool finish[MAX_PROCESSES] = {0};
   int safeSeq[MAX_PROCESSES];
   int work[MAX_RESOURCES];

   for (int i = 0; i < nr; i++)
      work[i] = avail[i];


   int count = 0,d;
   while (count < np) {
      bool found = false;
      for (int p = 0; p < np; p++) {
         if (finish[p] == 0) {
            int j;
            for (j = 0; j < nr; j++)
               if (req[p][j] > work[j])
               {
                  d=p;
                  break;
               }
            if (j == nr) {
               for (int k = 0; k < nr; k++)
                  work[k] += allot[p][k];

               safeSeq[count++] = p;
               finish[p] = 1;
               found = true;
            }
         }
      }
      if (found == false) {
         printf("Deadlock Detected at processes <",d);
         for (int i=0; i<np; i++)
            if (finish[i] == 0)
               printf("%d ", i);
         printf(">");
```

```
            return false;
        }
    }

    printf("System is in safe state.\nSafe sequence is: ");
    for (int i = 0; i < np; i++)
        printf("%d ", safeSeq[i]);
    printf("\n");

    return true;
}

void printResourceAllocationDetails(int np, int nr, int processes[], int req[][MAX_RESOURCES], int
allot[][MAX_RESOURCES], int avail[]) {
    printf("\nProcess\t\tAllocation\tRequest\t\tAvailable\n");
    for (int i = 0; i < np; i++) {
        printf("%d\t", processes[i]);

        // Print Allocation
        printf("\t");
        for (int j = 0; j < nr; j++)
            printf("%d ", allot[i][j]);

        // Print Max
        printf("\t\t");
        for (int j = 0; j < nr; j++)
            printf("%d ", req[i][j]);

        // Print Available
        if (i == 0) {
            printf("\t\t");
            for (int j = 0; j < nr; j++)
                printf("%d ", avail[j]);
        }
        printf("\n");
    }
}

int main() {
    int np, nr;
    printf("Enter number of processes: ");
    scanf("%d", &np);
    printf("Enter number of resource types: ");
    scanf("%d", &nr);

    int processes[MAX_PROCESSES];
    for (int i = 0; i < np; i++) processes[i] = i;

    int avail[MAX_RESOURCES];
    printf("Enter available resources: ");
    for (int i = 0; i < nr; i++)
        scanf("%d", &avail[i]);
```

```
    int req[MAX_PROCESSES][MAX_RESOURCES];
    printf("Enter Request matrix:\n");
    for (int i = 0; i < np; i++) {
        printf("Process %d: ", i);
        for (int j = 0; j < nr; j++)
            scanf("%d", &req[i][j]);
    }

    int allot[MAX_PROCESSES][MAX_RESOURCES];
    printf("Enter allocation resource matrix:\n");
    for (int i = 0; i < np; i++) {
        printf("Process %d: ", i);
        for (int j = 0; j < nr; j++)
            scanf("%d", &allot[i][j]);
    }

    printResourceAllocationDetails(np, nr, processes, req, allot, avail);

    isSafe(processes, avail, req, allot, np, nr);

    return 0;
}
```

Result:

```
Enter number of processes: 5
Enter number of resource types: 3
Enter available resources: 0 0 0
Enter Request matrix:
Process 0: 0 0 0
Process 1: 2 0 2
Process 2: 0 0 1
Process 3: 1 0 0
Process 4: 0 0 2
Enter allocation resource matrix:
Process 0: 0 1 0
Process 1: 2 0 0
Process 2: 3 0 3
Process 3: 2 1 1
Process 4: 0 0 2

Process         Allocation      Request         Available
0               0 1 0           0 0 0           0 0 0
1               2 0 0           2 0 2
2               3 0 3           0 0 1
3               2 1 1           1 0 0
4               0 0 2           0 0 2
Deadlock Detected at processes <1 2 3 4 >
```

Program -9

Question: Write a C program to simulate the following contiguous memory allocationtechniques a) Worst-fit b) Best-fit c) First-fit

Code:

```
#include <stdio.h>

#define MAX 25

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

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

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

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

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

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

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

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

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

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

```
int main() {
   int b[MAX], f[MAX], nb, nf;
   printf("\nEnter the number of blocks:");
   scanf("%d", &nb);
   printf("Enter the number of files:");
   scanf("%d", &nf);
   printf("\nEnter the size of the blocks:-\n");
   for (int i = 1; i <= nb; i++) {
      printf("Block %d:", i);
      scanf("%d", &b[i]);
   }
   printf("Enter the size of the files :-\n");
   for (int i = 1; i <= nf; i++) {
      printf("File %d:", i);
      scanf("%d", &f[i]);
   }
   int b1[MAX], b2[MAX], b3[MAX];
   for (int i = 1; i <= nb; i++) {
      b1[i] = b[i];
      b2[i] = b[i];
      b3[i] = b[i];
   }
   firstFit(nb, nf, b1, f);
   bestFit(nb, nf, b2, f);
   worstFit(nb, nf, b3, f);
   return 0;
}
```

Result:

```
Enter the number of blocks:3
Enter the number of files:2

Enter the size of the blocks:-
Block 1:5
Block 2:2
Block 3:7
Enter the size of the files :-
File 1:1
File 2:4

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

Memory Management Scheme - Best Fit
File No File Size       Block No        Block Size      Fragment
1               1               2               2               1
2               4               1               5               1

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

Program -10

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

Code:

```
#include <stdio.h>
#include <stdlib.h>
void printFrames(int frames[], int n, int faults, int x) {
   printf("[");
   for (int i = 0; i < n; i++)
      printf(" %d ", frames[i]);
      if (x==1)
         printf("] Fault %d \n", faults);
      else
         printf("] \n");
}
void FIFO(int pages[], int n, int frame_size) {
   int frames[frame_size];
   for (int i = 0; i < frame_size; i++) frames[i] = -1;
   int index = 0, faults = 0;
   printf("FIFO: \n");
   for (int i = 0; i < n; i++)
   {
      int found=0;
      int x=0;
      for (int j = 0; j < frame_size; j++) {
         if (frames[j] == pages[i]) {
```

```
            found = 1;
            break;
         }
      }
      if (!found) {
         frames[index] = pages[i];
         index = (index + 1) % frame_size;
         faults++;
         x=1;
      }
      printFrames(frames, frame_size, faults, x);
   }
   printf("Total faults: %d\n", faults);
}
void LRU(int pages[], int n, int frame_size) {
   int frames[frame_size], last_used[frame_size];
   for (int i = 0; i < frame_size; i++) frames[i] = -1;
   for (int i = 0; i < frame_size; i++) last_used[i] = 0;
   int time = 0, faults = 0;

   printf("LRU: \n");
   for (int i = 0; i < n; i++)
   {
      int x=0;
      int found = 0;
      for (int j = 0; j < frame_size; j++) {
         if (frames[j] == pages[i]) {
            found = 1;
            last_used[j] = ++time;
            break;
         }
      }
      if (!found) {
         int lru_index = 0;
         for (int j = 1; j < frame_size; j++) {
            if (last_used[j] < last_used[lru_index])
               lru_index = j;
         }
         frames[lru_index] = pages[i];
         last_used[lru_index] = ++time;
         faults++;
         x=1;
      }
      printFrames(frames, frame_size, faults, x);
   }
   printf("Total faults: %d\n", faults);
}

void OPTIMAL(int pages[], int n, int frame_size) {
   int frames[frame_size];
   for (int i = 0; i < frame_size; i++)
      frames[i] = -1;
```

```
int faults = 0;

printf("OPTIMAL: \n");
for (int i = 0; i < n; i++)
{
   int x=0;
   int found = 0;
   for (int j = 0; j < frame_size; j++) {
      if (frames[j] == pages[i]) {
         found = 1;
         break;
      }
   }
   for(int p = 0; p < frame_size; p++)
   {
      if(frames[p]==-1 && found==0)
      {
         found=1;
         frames[p] = pages[i];
         faults++;
         x=1;
         break;
      }
   }
   if (!found) {
      int replace_index = 0, farthest = -1;
      for (int j = 0; j < frame_size; j++)
      {

         int k;
         for (k = i + 1; k < n; k++)
         {
            if (frames[j] == pages[k])
            {
               if (k > farthest)
               {
                  farthest = k;
                  replace_index = j;
               }
               break;
            }
         }
         if (k == n) {
            replace_index = j;
            break;
         }
      }
      frames[replace_index] = pages[i];
      faults++;
      x=1;
   }
   printFrames(frames, frame_size, faults, x);
```

```
    }
    printf("Total faults: %d\n", faults);
}

int main() {
    int n, frame_size;

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

    int *pages = (int *)malloc(n * sizeof(int));
    printf("Enter the page sequence: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &pages[i]);
    }

    printf("Enter the number of frames: ");
    scanf("%d", &frame_size);
    FIFO(pages, n, frame_size);
    LRU(pages, n, frame_size);
    OPTIMAL(pages, n, frame_size);

    free(pages);
    return 0;
}
```

Result:

```
Enter the number of pages: 19
Enter the page sequence: 0 9 0 1 8 1 8 7 8 7 1 2 8 7 8 2 3 8 3
Enter the number of frames: 3
FIFO:
[ 0  -1  -1 ] Fault 1
[ 0  9  -1 ] Fault 2
[ 0  9  -1 ]
[ 0  9  1 ] Fault 3
[ 8  9  1 ] Fault 4
[ 8  9  1 ]
[ 8  9  1 ]
[ 8  7  1 ] Fault 5
[ 8  7  1 ]
[ 8  7  1 ]
[ 8  7  1 ]
[ 8  7  2 ] Fault 6
[ 8  7  2 ]
[ 8  7  2 ]
[ 8  7  2 ]
[ 8  7  2 ]
[ 3  7  2 ] Fault 7
[ 3  8  2 ] Fault 8
[ 3  8  2 ]
Total faults: 8
```

```
LRU:
[ 0  -1  -1 ] Fault 1
[ 0  9  -1 ] Fault 2
[ 0  9  -1 ]
[ 0  9  1 ] Fault 3
[ 0  8  1 ] Fault 4
[ 0  8  1 ]
[ 0  8  1 ]
[ 7  8  1 ] Fault 5
[ 7  8  1 ]
[ 7  8  1 ]
[ 7  8  1 ]
[ 7  2  1 ] Fault 6
[ 8  2  1 ] Fault 7
[ 8  2  7 ] Fault 8
[ 8  2  7 ]
[ 8  2  7 ]
[ 8  2  3 ] Fault 9
[ 8  2  3 ]
[ 8  2  3 ]
Total faults: 9
OPTIMAL:
[ 0  -1  -1 ] Fault 1
[ 0  9  -1 ] Fault 2
[ 0  9  -1 ]
[ 0  9  1 ] Fault 3
[ 8  9  1 ] Fault 4
[ 8  9  1 ]
[ 8  9  1 ]
[ 8  7  1 ] Fault 5
[ 8  7  1 ]
[ 8  7  1 ]
[ 8  7  1 ]
[ 8  7  2 ] Fault 6
[ 8  7  2 ]
[ 8  7  2 ]
[ 8  7  2 ]
[ 8  7  2 ]
[ 8  3  2 ] Fault 7
[ 8  3  2 ]
[ 8  3  2 ]
Total faults: 7
```