VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS

Submitted by

DEEKSHA S (1BM21CS048)

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
June-2023 to September-2023

B. M. S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "OPERATING SYSTEMS" carried out by **DEEKSHA S (1BM21CS048),** 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 academic semester June-2023 to September-2023. The Lab report has been approved as it satisfies the academic requirements in respect of a OPERATING SYSTEMS **(22CS4PCOPS)** work prescribed for the said degree.

Name of the Lab-In charge:
Designation
Department of CSE
BMSCE, Bengaluru

Madhavi R.P.
Associate Professor
Department of CSE
BMSCE, Bengaluru

Index Sheet

	(c) First-fit	
10	Write a C program to simulate page replacement algorithms: (a) FIFO (b) LRU (c) Optimal	61-72
11	Write a C program to simulate disk scheduling algorithms: (a) FCFS (b) SCAN (c) C-SCAN	73-81

Course Outcome

CO1	Analyze time complexity of Recursive and Non-recursive algorithms using asymptotic notations.
CO2	Apply various design techniques for the given problem.
CO3	Apply the knowledge of complexity classes P, NP, and NP-Complete and prove certain problems are NP-Complete
CO4	Design efficient algorithms and conduct practical experiments to solve problems.

PROGRAM 1

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

(a) FCFS

(b) SJF

Program:

```
#include<stdio.h>
int at[20],cput[20];
void main(){
   int n,i,choice;
   printf("Enter the number of processes\n");
   scanf("%d",&n);
   printf("Enter arrival time and cpu time for each process respectively\n");
   for(i =0;i<n;i++){
      scanf("%d %d",&at[i],&cput[i]);
   }
   printf("Menu\n\n1.FCFS\n2.SJF(Non Preemptive)\n3.SRTF(Preemptive)\n4.Exit\n");
   while(1){
      scanf("%d",&choice);
      switch(choice){
      case 1: fcfs(n);
      break;
      case 2: sjf(n);
      break;
      case 3: srtf(n);
      break;
      case 4: exit(0);
```

```
        default:printf("Wring choice\n");
    }
    }
}

void srtf(int n) {
    int remaining_time[20], tat[20], wt[20], completion_time[20], smallest, time, i, count = 0;
    float awt=0,atat=0;
    for (i = 0; i < n; i++)
        remaining_time[i] = cput[i];

    time = 0;
    while (count != n) {
        smallest = -1;
        for (i = 0; i < n; i++) {
            if (at[i] <= time && remaining_time[i] > 0) {
                if (smallest == -1 || remaining_time[i] < remaining_time[smallest])
                    smallest = i;
            }
        }
        if (smallest == -1) {
            time++;
            continue;
        }
        remaining_time[smallest]--;
        if (remaining_time[smallest] == 0) {
            count++;
            completion_time[smallest] = time + 1;
```

```
            wt[smallest] = completion_time[smallest] - at[smallest] - cput[smallest];
            tat[smallest] = completion_time[smallest] - at[smallest];
        }
        time++;
    }
    for(i=0;i<n;i++){
        awt+=wt[i];
        atat += tat[i];
    }
    awt = awt/n;
    atat =atat/n;

    printf("\nProcess\tArrival Time\tCPU Time\tWaiting Time\tTurnaround Time\n");
    for (i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", i, at[i], cput[i], wt[i], tat[i]);
    }
    printf("\nAverage Waiting Time -- %f", awt);
    printf("\nAverage Turnaround Time -- %f\n", atat);



}

void sjf(int n){
    int cmpt[20],tat[20],wt[20],cput1[20];
    float awt=0, atat=0,sum_burst_time=0;
    int sum=0,i,j, smallest;
        printf("\t PROCESS \t TURNAROUND TIME\t WAITING TIME\n");
```

```
    for (i = 0; i < n; i++) {
            cput1[i]=cput[i];
    sum_burst_time += cput[i];
  }
    cput1[9]=9999;
    while(sum < sum_burst_time)
   {
    smallest = 9;
    for (i = 0; i < n; i++) {
      if (at[i] <= sum && cput1[i] > 0 && cput1[i] < cput1[smallest])
        smallest = i;
    }
    printf("\t P[%d] \t\t %d \t\t %d\n", smallest , sum + cput1[smallest] - at[smallest], sum -
at[smallest]);
    awt += sum - at[smallest];
    atat += sum + cput1[smallest] - at[smallest];
    sum += cput1[smallest];
    cput1[smallest] = 0;
  }
    awt = awt/n;
    atat =atat/n;

        printf("\nAverage Waiting Time -- %f", awt);
        printf("\nAverage Turnaround Time -- %f\n", atat);
}

void fcfs(int n){
    int cmpt[20],tat[20],wt[20],pname[20],temp;
```

```
float awt=0, atat=0;
int sum=0,i;
for(i=0;i<n;i++){
  pname[i]=i;
}
for(i=0;i<n;i++){
  if(at[i]==at[i+1] && cput[i]>cput[i+1]){
    temp = cput[i];
    cput[i]=cput[i+1];
    cput[i+1]=temp;
    temp = pname[i];
    pname[i]=pname[i+1];
    pname[i+1]=temp;
  }
}
for(i=0;i<n;i++){
  sum += cput[i];
  cmpt[i]=sum;
  tat[i]=cmpt[i]-at[i];
  wt[i]=tat[i]-cput[i];
}

for(i=0;i<n;i++){
    awt+=wt[i];
    atat += tat[i];
}
awt = awt/n;
atat =atat/n;
```

```
   printf("\t PROCESS \tARRIVAL TIME \tCPU TIME \t WAITING TIME\t TURNAROUND TIME\n");
   for(i=0;i<n;i++){
      printf("\n\t P%d \t\t %d \t\t %d \t\t %d \t\t %d", pname[i],at[i], cput[i], wt[i], tat[i]);
      }
      printf("\nAverage Waiting Time -- %f", awt);
      printf("\nAverage Turnaround Time -- %f\n", atat);
getch();
}
```

OUTPUT:

```
E:\Downloads\Lab2_OS.exe
Enter the number of processes
4
Enter arrival time and cpu time for each process respectively
0 3
1 6
4 4
6 2
Menu

1.FCFS
2.SJF(Non Preemptive)
3.SRTF(Preemptive)
4.Exit
1
        PROCESS        ARRIVAL TIME    CPU TIME        WAITING TIME    TURNAROUND TIME

        P0              0               3               0               3
        P1              1               6               2               8
        P2              4               4               5               9
        P3              6               2               7               9
Average Waiting Time -- 3.500000
Average Turnaround Time -- 7.250000
2
        PROCESS         TURNAROUND TIME         WAITING TIME
        P[0]            3               0
        P[1]            8               2
        P[3]            5               3
        P[2]            11              7

Average Waiting Time -- 3.000000
Average Turnaround Time -- 6.750000
3

Process Arrival Time    CPU Time        Waiting Time    Turnaround Time
0       0               3               0               3
1       1               6               8               14
2       4               4               0               4
3       6               2               2               4

Average Waiting Time -- 2.500000
Average Turnaround Time -- 6.250000
```

E:\Downloads\Lab2_OS.exe

```
Enter the number of processes
4
Enter arrival time and cpu time for each process respectively
0 10
3 2
5 1
10 5
Menu

1.FCFS
2.SJF(Non Preemptive)
3.SRTF(Preemptive)
4.Exit
1
        PROCESS        ARRIVAL TIME    CPU TIME         WAITING TIME    TURNAROUND TIME

        P0              0               10              0               10
        P1              3               2               7               9
        P2              5               1               7               8
        P3              10              5               3               8
Average Waiting Time -- 4.250000
Average Turnaround Time -- 8.750000
2
        PROCESS         TURNAROUND TIME         WAITING TIME
        P[0]            10                      0
        P[2]            6                       5
        P[1]            10                      8
        P[3]            8                       3

Average Waiting Time -- 4.000000
Average Turnaround Time -- 8.500000
3

Process Arrival Time    CPU Time        Waiting Time    Turnaround Time
0       0               10              3               13
1       3               2               0               2
2       5               1               0               1
3       10              5               3               8

Average Waiting Time -- 1.500000
Average Turnaround Time -- 6.000000
```

PROGRAM 2

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

(a) Priority (Non-pre-emptive)

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

Program:

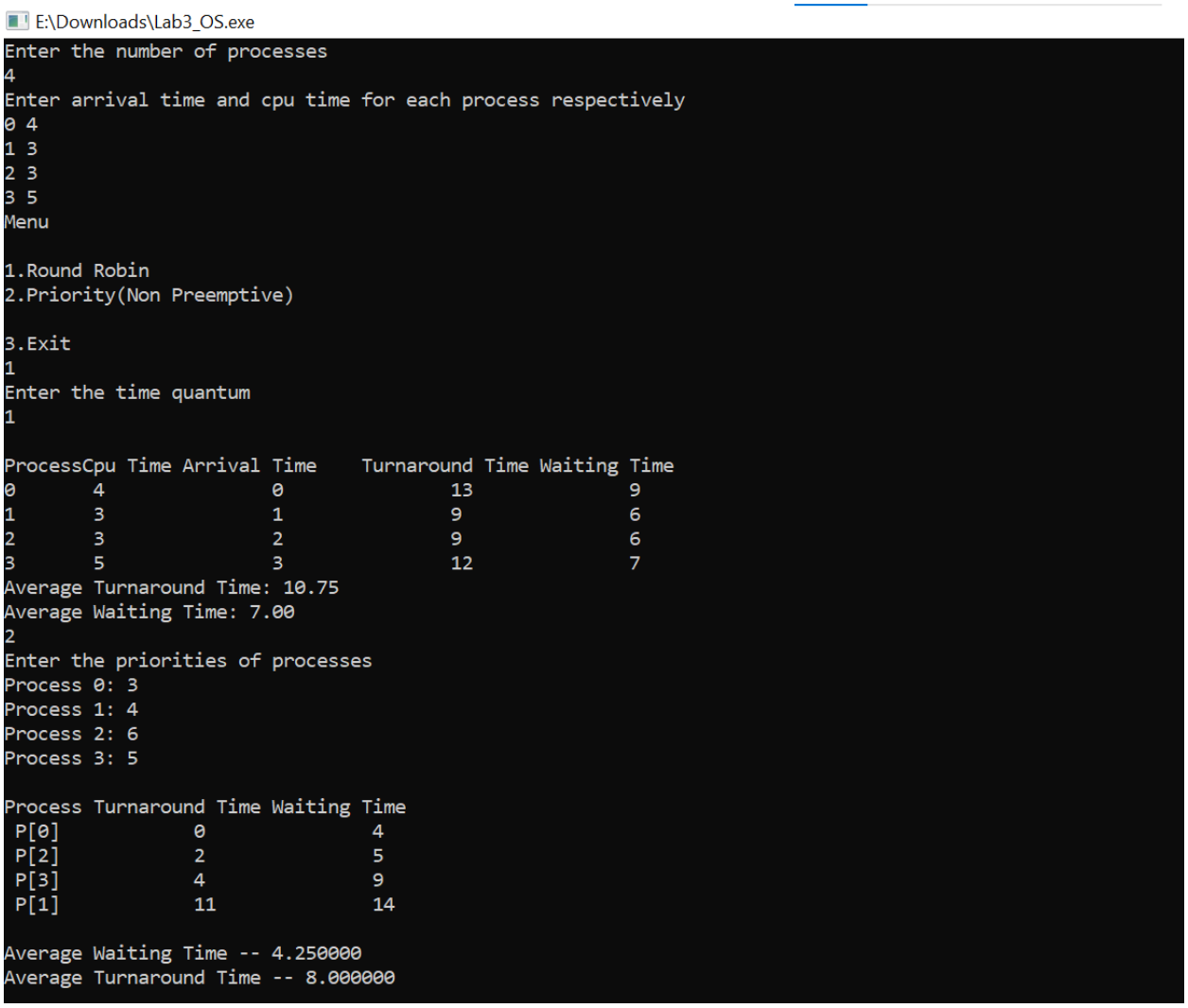
```
#include<stdio.h>
int at[20],cput[20];
void main()
{
   int n,i,choice,tq;
   printf("Enter the number of processes\n");
   scanf("%d",&n);
   printf("Enter arrival time and cpu time for each process respectively\n");
   for(i =0;i<n;i++){
      scanf("%d %d",&at[i],&cput[i]);
   }

   printf("Menu\n\n1.Round Robin\n2.Priority(Non Preemptive)\n\n3.Exit\n");
   while(1){
      scanf("%d",&choice);
      switch(choice){
      case 1: printf("Enter the time quantum \n");
      scanf("%d",&tq);
         roundRobin(n,tq);
      break;
      case 2: NonprePriority(n);
```

```
        break;
        case 3: exit(0);
        default:printf("Wring choice\n");
    }
    }
}

void roundRobin(int n, int tq){
    int i,remaining_time[20],wt[20],tat[20],completed = 0,time=0;
    float awt=0,atat=0;
    for(i=0;i<n;i++){
        remaining_time[i]=cput[i];
    }
    while (completed < n) {
        for (int i = 0; i < n; i++) {
            if (remaining_time[i] > 0 && at[i] <= time) {
                if (remaining_time[i] <= tq) {
                    time += remaining_time[i];
                    remaining_time[i] = 0;
                    completed++;

                    tat[i] = time - at[i];
                    wt[i] = tat[i] - cput[i];
                } else {
                    time += tq;
                    remaining_time[i] -= tq;
                }
            }
```

```
        }
    }
    for (int i = 0; i < n; i++) {
        atat += tat[i];
        awt += wt[i];
    }
    atat /= n;
    awt /= n;

    printf("\nProcess\Cpu Time\tArrival 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\n", i , cput[i], at[i], tat[i], wt[i]);
    }
    printf("Average Turnaround Time: %.2f\n", atat);
    printf("Average Waiting Time: %.2f\n", awt);
}

void NonprePriority(int n){
    int priority[20],wt[20],tat[20],hp=0,cmpt[20],cput1[20],sum=0,i,sum_burst_time=0;
    float awt=0,atat=0;
    printf("Enter the priorities of processes\n");
     for (int i = 0; i < n; i++) {
        printf("Process %d: ", i);
        scanf("%d", &priority[i]);
    }
    for (i = 0; i < n; i++) {
            cput1[i]=cput[i];
    sum_burst_time += cput[i];
```

```
  }
   printf("\nProcess\tTurnaround Time\tWaiting Time\n");
  cput1[9]=-1;
   while(sum < sum_burst_time)
   {
   hp = 9;
   for (i = 0; i < n; i++) {
     if (at[i] <= sum && cput1[i]>0 && priority[i] > priority[hp])
       hp = i;
   }
   printf(" P[%d] \t\t %d \t\t %d\n", hp , sum - at[hp], sum + cput1[hp] - at[hp]);
   awt += sum - at[hp];
   atat += sum + cput1[hp] - at[hp];
   sum += cput1[hp];
   cput1[hp] = 0;
  }

  awt = awt/n;
   atat =atat/n;

      printf("\nAverage Waiting Time -- %f", awt);
      printf("\nAverage Turnaround Time -- %f\n", atat);
}
```

OUTPUT:



```
E:\Downloads\Lab3_OS.exe
Enter the number of processes
4
Enter arrival time and cpu time for each process respectively
0 4
1 3
2 3
3 5
Menu

1.Round Robin
2.Priority(Non Preemptive)

3.Exit
1
Enter the time quantum
1

ProcessCpu Time Arrival Time    Turnaround Time Waiting Time
0       4               0               13              9
1       3               1               9               6
2       3               2               9               6
3       5               3               12              7
Average Turnaround Time: 10.75
Average Waiting Time: 7.00
2
Enter the priorities of processes
Process 0: 3
Process 1: 4
Process 2: 6
Process 3: 5

Process Turnaround Time Waiting Time
 P[0]            0               4
 P[2]            2               5
 P[3]            4               9
 P[1]            11              14

Average Waiting Time -- 4.250000
Average Turnaround Time -- 8.000000
```

E:\Downloads\Lab3_OS.exe

```
Enter the number of processes
4
Enter arrival time and cpu time for each process respectively
0 5
1 3
2 1
3 2
Menu

1.Round Robin
2.Priority(Non Preemptive)

3.Exit
1
Enter the time quantum
2

ProcessCpu Time Arrival Time    Turnaround Time Waiting Time
0       5               0               11              6
1       3               1               9               6
2       1               2               3               2
3       2               3               4               2
Average Turnaround Time: 6.75
Average Waiting Time: 4.00
2
Enter the priorities of processes
Process 0: 3
Process 1: 2
Process 2: 1
Process 3: 4

Process Turnaround Time Waiting Time
 P[0]           0               5
 P[3]           2               4
 P[1]           6               9
 P[2]           8               9

Average Waiting Time -- 4.000000
Average Turnaround Time -- 6.750000
```

PROGRAM 3

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

Program:

```
#include<stdio.h>
#define MAX 50
typedef struct
{
   int number;
   int p_id[MAX];
   int tat[MAX];
   int wt[MAX];
   int arrival_time[MAX];
   int cpu_time[MAX];
} Process;

void main()
{
   int n, i,j,pname[MAX],total_time=0,time=0;
   int totaltat[MAX], totalwt[MAX];
   float avgtat=0,avgwt=0;
   Process sp,up;
   printf("Enter the number of system processes: ");
   scanf("%d", &sp.number);

    printf("Enter the Arrival time and the Burst time for system processes:\n");
```

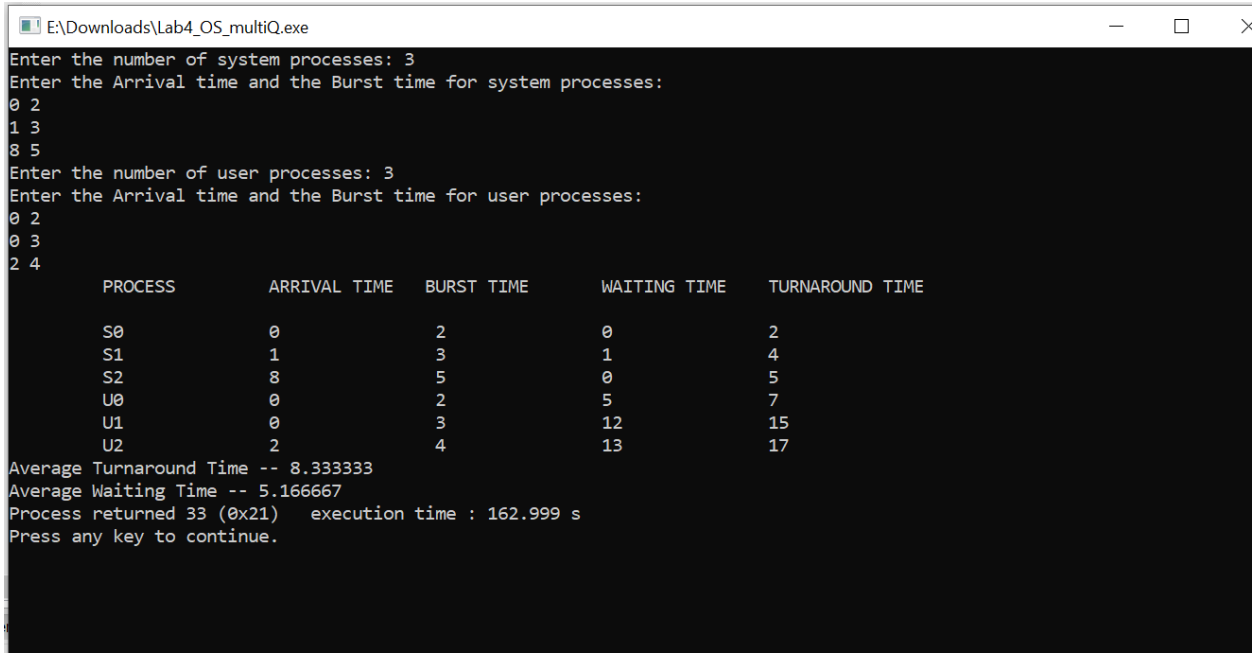
```
// Read process details
for (i = 0; i < sp.number; i++)
{
   //printf("Process %d\n", i + 1);
   scanf("%d", &sp.arrival_time[i]);
   scanf("%d", &sp.cpu_time[i]);

   sp.p_id[i] = 10+i + 1;
}
printf("Enter the number of user processes: ");
scanf("%d", &up.number);
printf("Enter the Arrival time and the Burst time for user processes:\n");
// Read process details
for (i = 0; i < up.number; i++)
{
   //printf("Process %d\n", i + 1);
   scanf("%d", &up.arrival_time[i]);
   scanf("%d", &up.cpu_time[i]);

   up.p_id[i] = 20+i + 1;
}

for(i=0;i<sp.number;i++)
   total_time+=sp.cpu_time[i];

for(i=0;i<up.number;i++)
   total_time+=up.cpu_time[i];
```

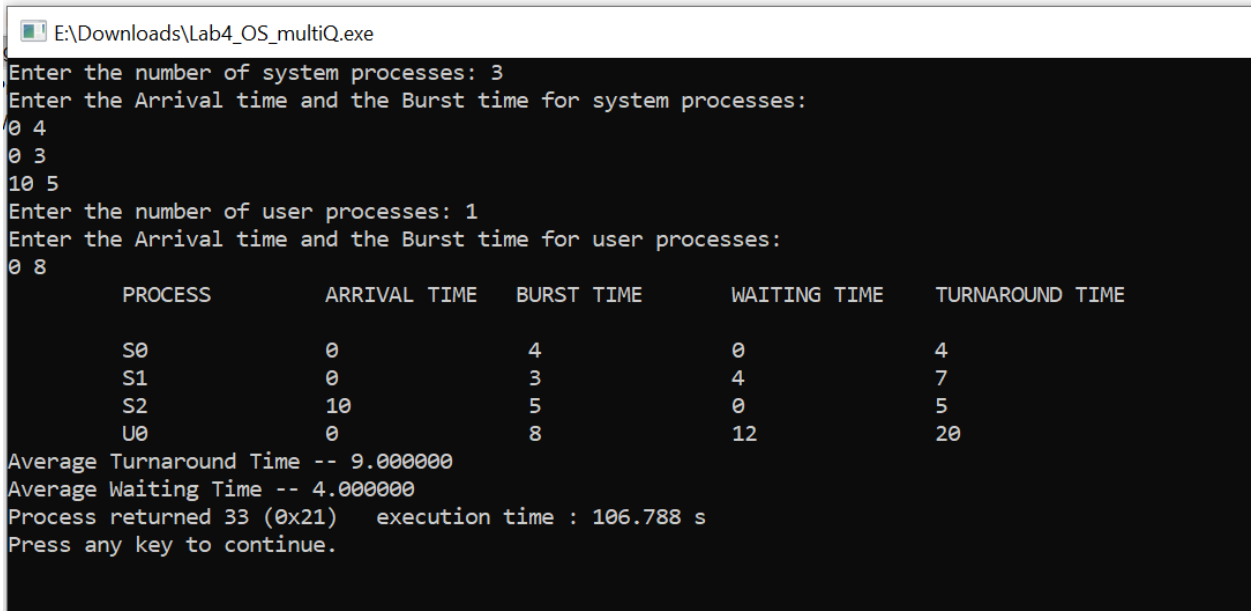
```
i=0,j=0;
while(time<total_time)
{
   if(sp.arrival_time[i]<=up.arrival_time[j])
   {
      time+=sp.cpu_time[i];
      sp.tat[i]=time-sp.arrival_time[i];
      sp.wt[i]=sp.tat[i]-sp.cpu_time[i];
      i++;
   }
   else if (up.arrival_time[j] < sp.arrival_time[i] && sp.arrival_time[i] > time)
{
   if ((time + up.cpu_time[j]) > sp.arrival_time[i])
{
   int ubt = up.cpu_time[j], sbt = sp.cpu_time[i];

   while (time < sp.arrival_time[i])
   {
      time++;
      ubt--;
   }

   time += sbt;
   sp.tat[i] = time - sp.arrival_time[i];
   sp.wt[i] = sp.tat[i] - sp.cpu_time[i];
   i++;
   time += ubt;
   up.tat[j] = time - up.arrival_time[j];
```

```
    up.wt[j] = up.tat[j] - up.cpu_time[j];
    j++;
  }
  else
  {
    time += up.cpu_time[j];
    up.tat[j] = time - up.arrival_time[j];
    up.wt[j] = up.tat[j] - up.cpu_time[j];
    j++;
  }
  }

    else if(up.arrival_time[j]<=sp.arrival_time[i]&&sp.arrival_time[i]<time)
    {
      time+=sp.cpu_time[i];
      sp.tat[i]=time-sp.arrival_time[i];
      sp.wt[i]=sp.tat[i]-sp.cpu_time[i];
      i++;
    }
    //printf("%d",sp.tat[0]);
  }
  printf("\t PROCESS \t ARRIVAL TIME \tBURST TIME \t WAITING TIME\t
TURNAROUND TIME\n");
  for(i=0;i<sp.number;i++)
  {
    printf("\n\t S%d \t\t %d \t\t %d \t\t %d \t\t %d", i,sp.arrival_time[i], sp.cpu_time[i], sp.wt[i],
sp.tat[i]);
    avgtat+=sp.tat[i];
```

```
        avgwt+=sp.wt[i];
    }
    for(i=0;i<up.number;i++)
    {
         printf("\n\t U%d \t\t %d \t\t %d \t\t %d \t\t %d", i,up.arrival_time[i], up.cpu_time[i],
up.wt[i], up.tat[i]);
        avgtat+=up.tat[i];
        avgwt+=up.wt[i];
    }
    avgtat/=(sp.number+up.number);
    avgwt/=(sp.number+up.number);
    printf("\nAverage Turnaround Time -- %f", avgtat);
    printf("\nAverage Waiting Time -- %f", avgwt);
}
```

OUTPUT:




E:\Downloads\Lab4_OS_multiQ.exe
Enter the number of system processes: 3
Enter the Arrival time and the Burst time for system processes:
0 4
0 3
10 5
Enter the number of user processes: 1
Enter the Arrival time and the Burst time for user processes:
0 8
PROCESS ARRIVAL TIME BURST TIME WAITING TIME TURNAROUND TIME
S0 0 4 0 4
S1 0 3 4 7
S2 10 5 0 5
U0 0 8 12 20
Average Turnaround Time -- 9.000000
Average Waiting Time -- 4.000000
Process returned 33 (0x21) execution time : 106.788 s
Press any key to continue.

PROGRAM 4

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

(a) Rate- Monotonic

(b) Earliest-deadline First

Program:

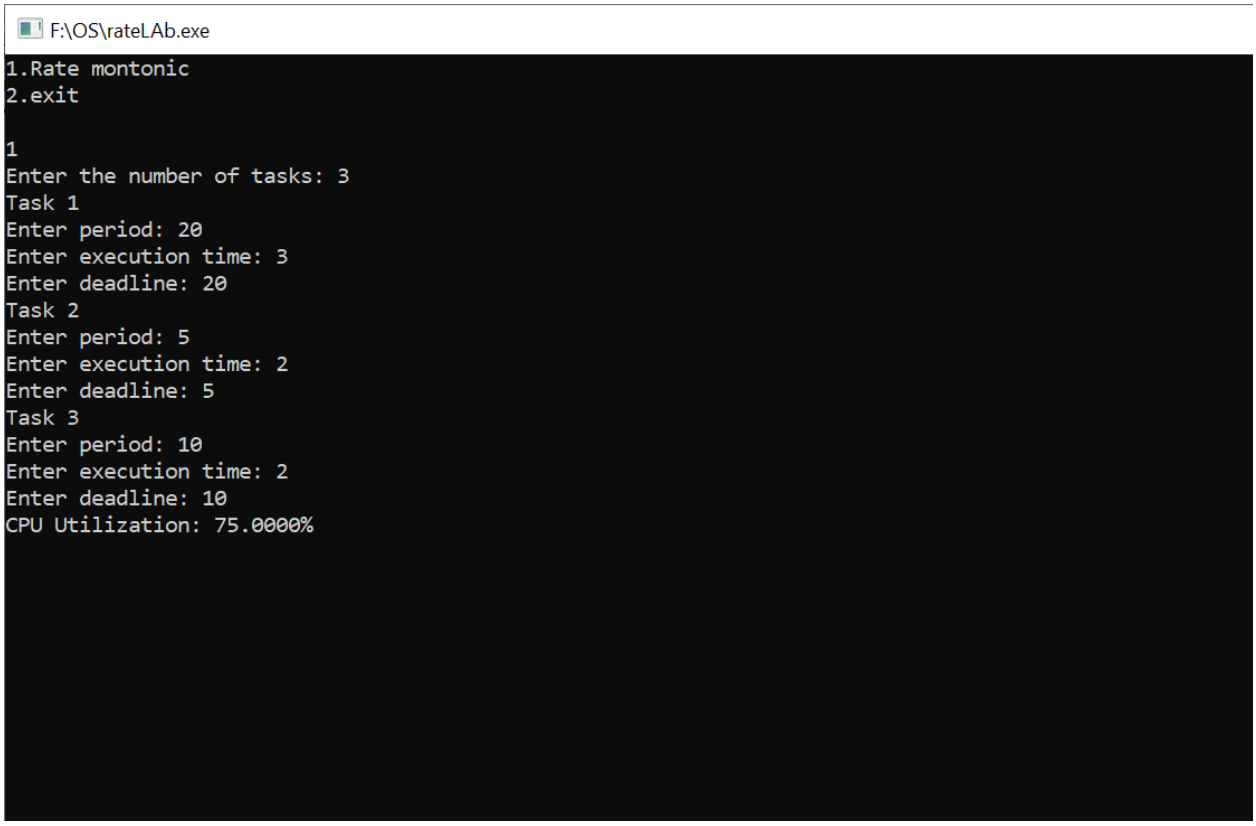
a) Rate- Monotonic:

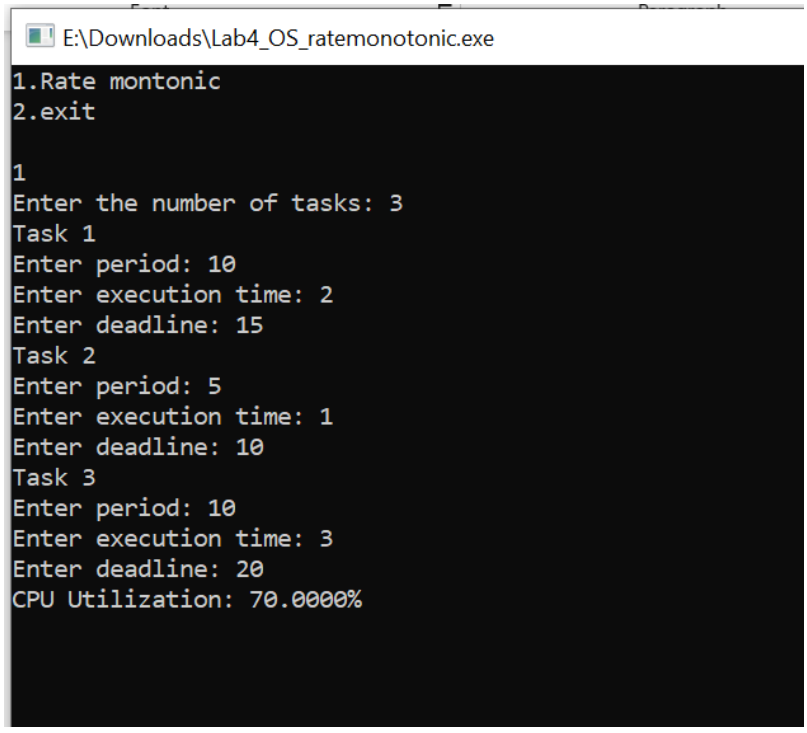
```
#include<stdio.h>
#define MAX_TASKS 100

typedef struct {
   int pid;
   int period;
   int exec_time;
   int deadline;
} Task;

float cpu_util(Task tasks[], int n) {
   float total_utilization = 0.0;

   for (int i = 0; i < n; i++) {
      float task_utilization = (float)tasks[i].exec_time / tasks[i].period;
      total_utilization += task_utilization;
   }

   float cpu_utilization = total_utilization * 100;
   return cpu_utilization;
```

```
}

rateMonotonic(){
    int n, i;

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

    Task tasks[MAX_TASKS];

    for (i = 0; i < n; i++) {
        printf("Task %d\n", i + 1);
        printf("Enter period: ");
        scanf("%d", &tasks[i].period);
        printf("Enter execution time: ");
        scanf("%d", &tasks[i].exec_time);
       /* printf("Enter execution time: ");
        scanf("%d", &tasks[i].exec_time;*/
        printf("Enter deadline: ");
        scanf("%d", &tasks[i].deadline);

        tasks[i].pid = i + 1;
    }

    float cpu_utilization = cpu_util(tasks, n);

    printf("CPU Utilization: %.4f%%\n", cpu_utilization);
}
```

```
void main(){
   int choice, n, i;
   printf("1.Rate montonic\n2.exit\n\n");
   while(1){
      scanf("%d",&choice);
      switch(choice){
         case 1: rateMonotonic();
         break;
         case 2:
            exit(0);
         default: printf("Wrong choice\n");
      }
   }
}
```

OUTPUT:


F:\OS\rateLAb.exe
1.Rate montonic
2.exit
1
Enter the number of tasks: 3
Task 1
Enter period: 20
Enter execution time: 3
Enter deadline: 20
Task 2
Enter period: 5
Enter execution time: 2
Enter deadline: 5
Task 3
Enter period: 10
Enter execution time: 2
Enter deadline: 10
CPU Utilization: 75.0000%

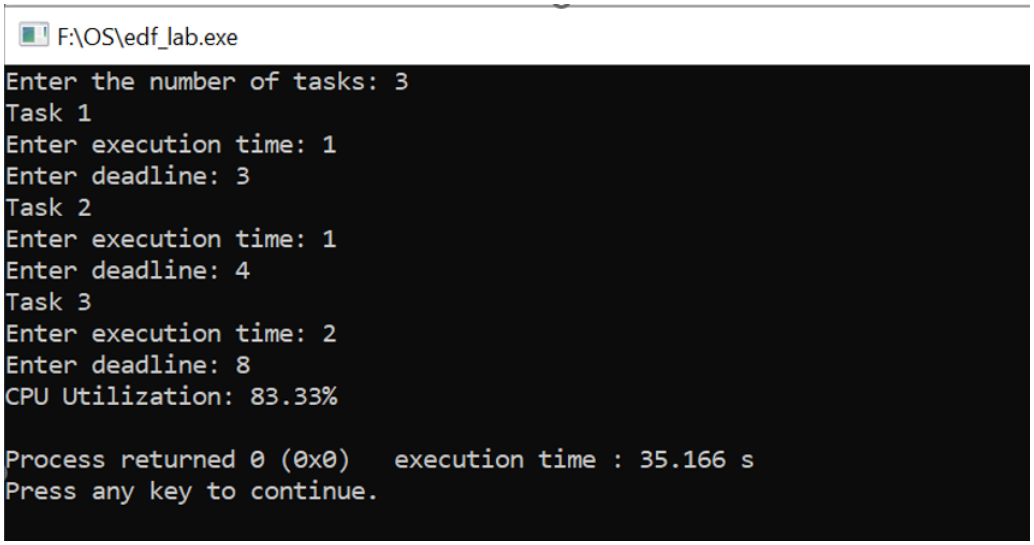

E:\Downloads\Lab4_OS_ratemonotonic.exe
1.Rate montonic
2.exit
1
Enter the number of tasks: 3
Task 1
Enter period: 10
Enter execution time: 2
Enter deadline: 15
Task 2
Enter period: 5
Enter execution time: 1
Enter deadline: 10
Task 3
Enter period: 10
Enter execution time: 3
Enter deadline: 20
CPU Utilization: 70.0000%

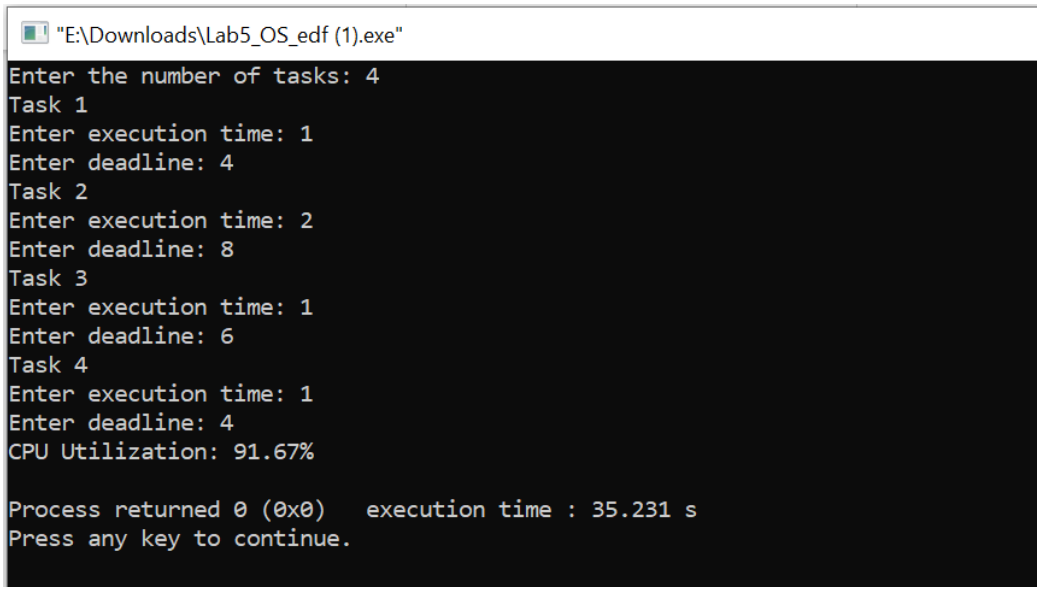
b) Earliest-deadline First:

```
#include <stdio.h>

#define MAX_TASKS 100

typedef struct {
    int task_id;
    int arrival_time;
    int execution_time;
    int deadline;
    int is_completed;
} Task;

float calc_cpu_util(Task tasks[], int n) {
    float total_util = 0.0;

    for (int i = 0; i < n; i++) {
        float task_util = (float)tasks[i].execution_time / tasks[i].deadline;
        total_util += task_util;
    }

    float cpu_util = total_util * 100;
    return cpu_util;
}

int main() {
    int n, i;
```

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

    Task tasks[MAX_TASKS];

    for (i = 0; i < n; i++) {
        printf("Task %d\n", i + 1);
        printf("Enter execution time: ");
        scanf("%d", &tasks[i].execution_time);
        printf("Enter deadline: ");
        scanf("%d", &tasks[i].deadline);

        tasks[i].task_id = i + 1;
        tasks[i].is_completed = 0;
    }

    float cpu_util = calc_cpu_util(tasks, n);

    printf("CPU Utilization: %.2f%%\n", cpu_util);

    return 0;
}
```

OUTPUT:


F:\OS\edf_lab.exe
Enter the number of tasks: 3
Task 1
Enter execution time: 1
Enter deadline: 3
Task 2
Enter execution time: 1
Enter deadline: 4
Task 3
Enter execution time: 2
Enter deadline: 8
CPU Utilization: 83.33%
Process returned 0 (0x0) execution time : 35.166 s
Press any key to continue.


"E:\Downloads\Lab5_OS_edf (1).exe"
Enter the number of tasks: 4
Task 1
Enter execution time: 1
Enter deadline: 4
Task 2
Enter execution time: 2
Enter deadline: 8
Task 3
Enter execution time: 1
Enter deadline: 6
Task 4
Enter execution time: 1
Enter deadline: 4
CPU Utilization: 91.67%
Process returned 0 (0x0) execution time : 35.231 s
Press any key to continue.

PROGRAM 5

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

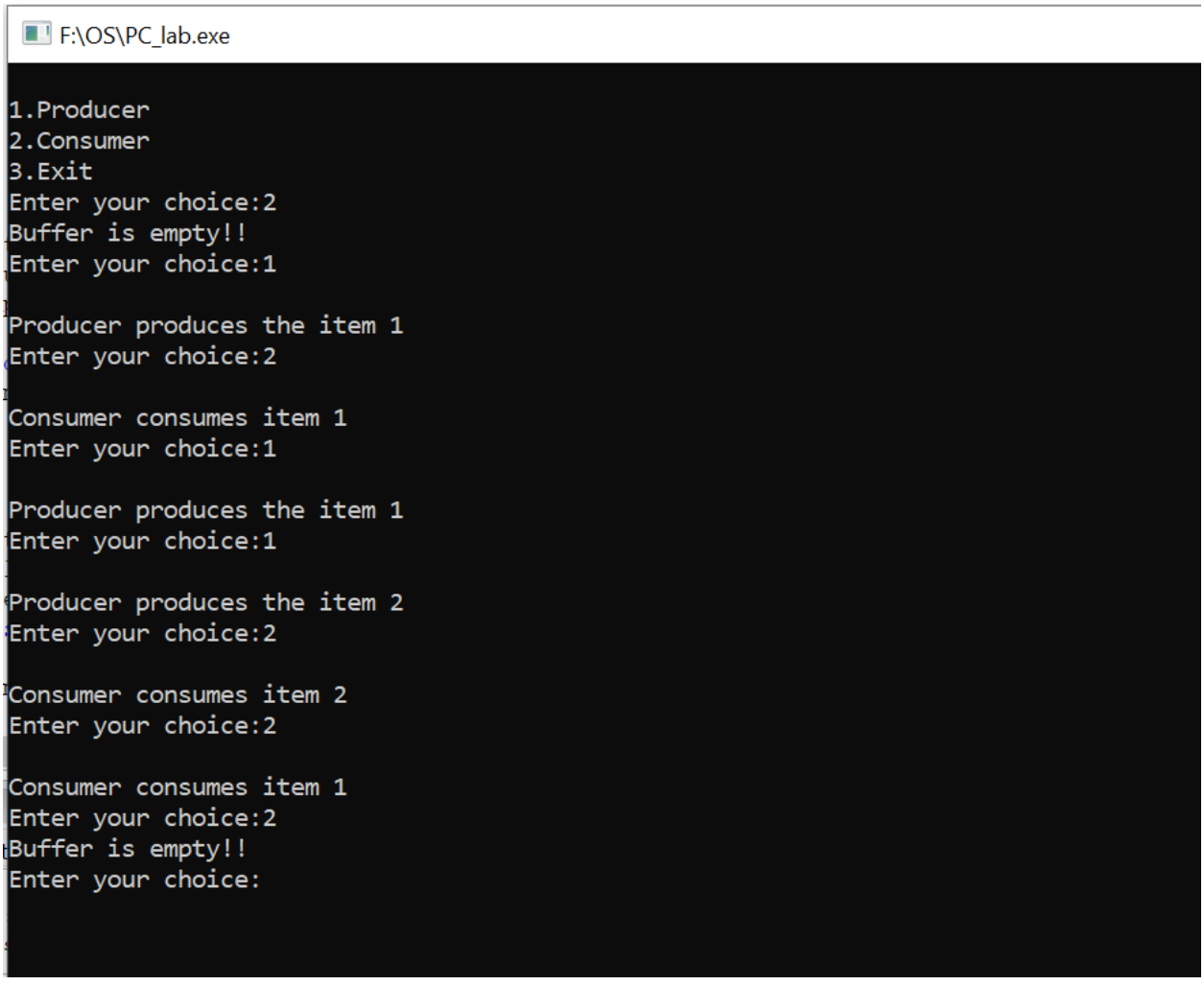
Program:

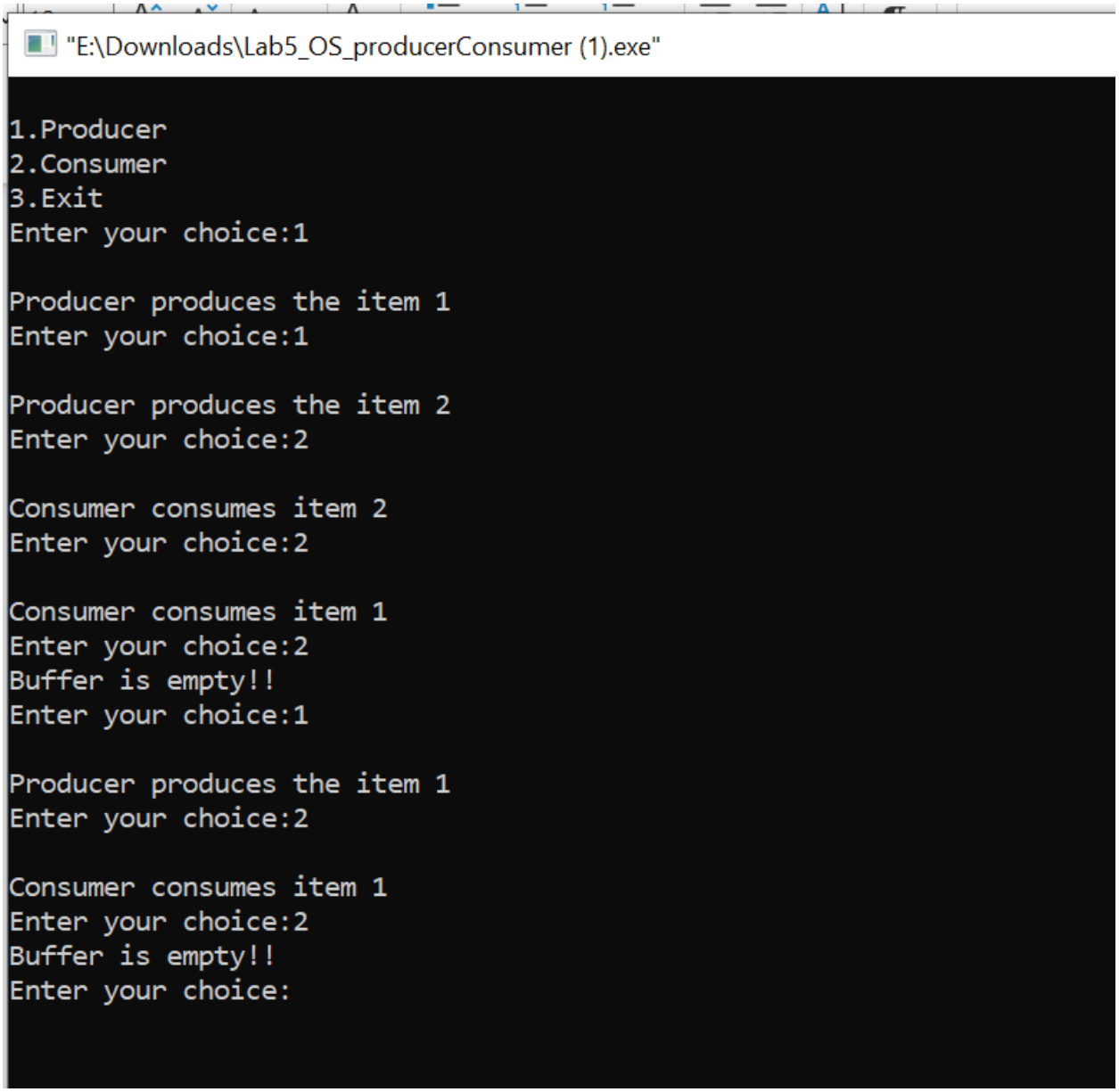
```
#include<stdio.h>
#include<stdlib.h>

int mutex=1,full=0,empty=3,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:
            exit(0);
            break;
      }
   }
   return 0;
}

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

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

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

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

OUTPUT:




"E:\Downloads\Lab5_OS_producerConsumer (1).exe"
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:1
Producer produces the item 1
Enter your choice:2
Consumer consumes item 1
Enter your choice:2
Buffer is empty!!
Enter your choice:

PROGRAM 6

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

Program:

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

#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };

sem_t mutex;
sem_t S[N];

void test(int phnum)
{
        if (state[phnum] == HUNGRY
                && state[LEFT] != EATING
                && state[RIGHT] != EATING) {
                // state that eating
                state[phnum] = EATING;
```

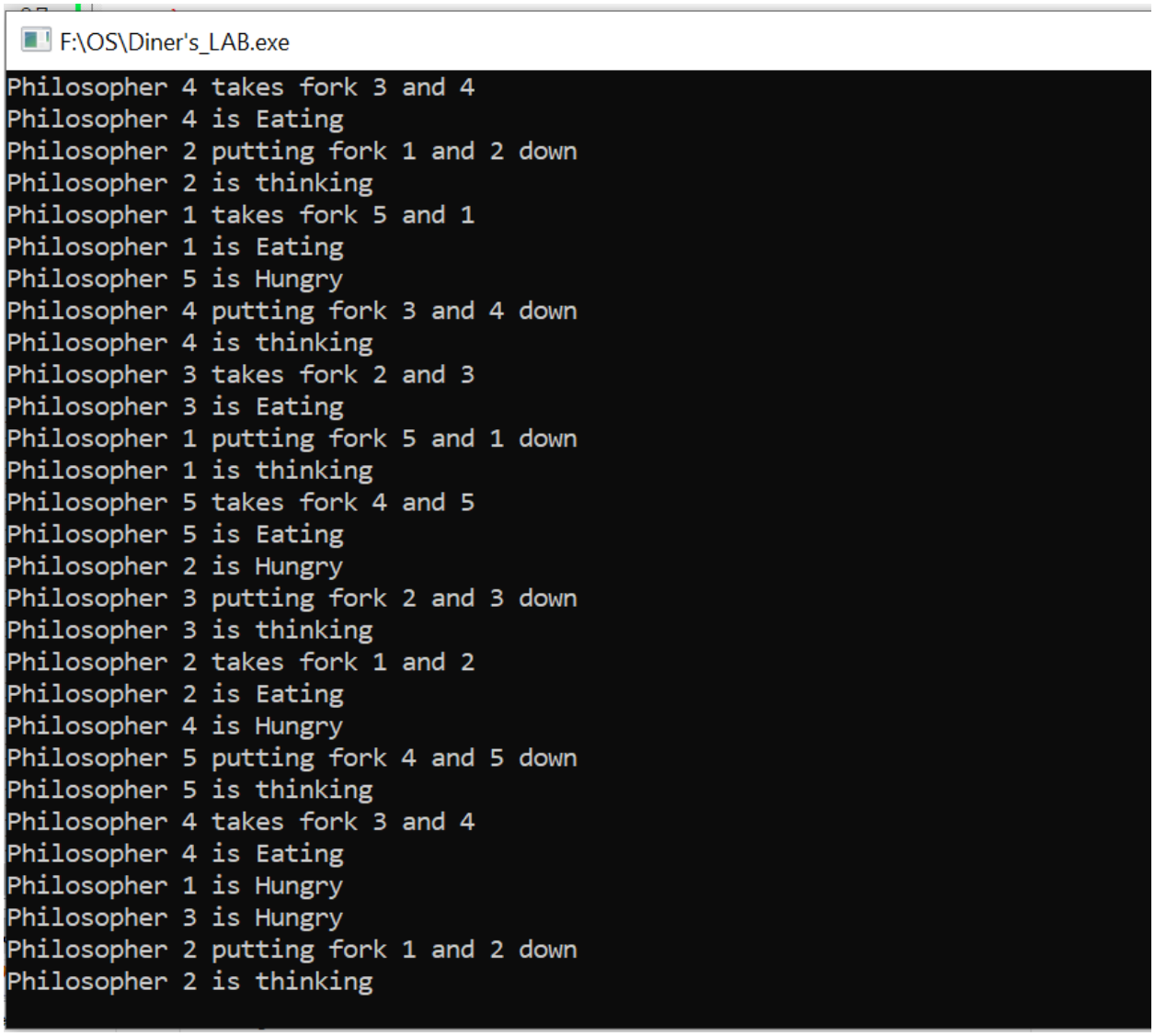
```
            sleep(2);

            printf("Philosopher %d takes fork %d and %d\n",
                                    phnum + 1, LEFT + 1, phnum + 1);

            printf("Philosopher %d is Eating\n", phnum + 1);

            sem_post(&S[phnum]);
        }
}

void take_fork(int phnum)
{

        sem_wait(&mutex);

        state[phnum] = HUNGRY;

        printf("Philosopher %d is Hungry\n", phnum + 1);

        test(phnum);

        sem_post(&mutex);

        sem_wait(&S[phnum]);

        sleep(1);
```

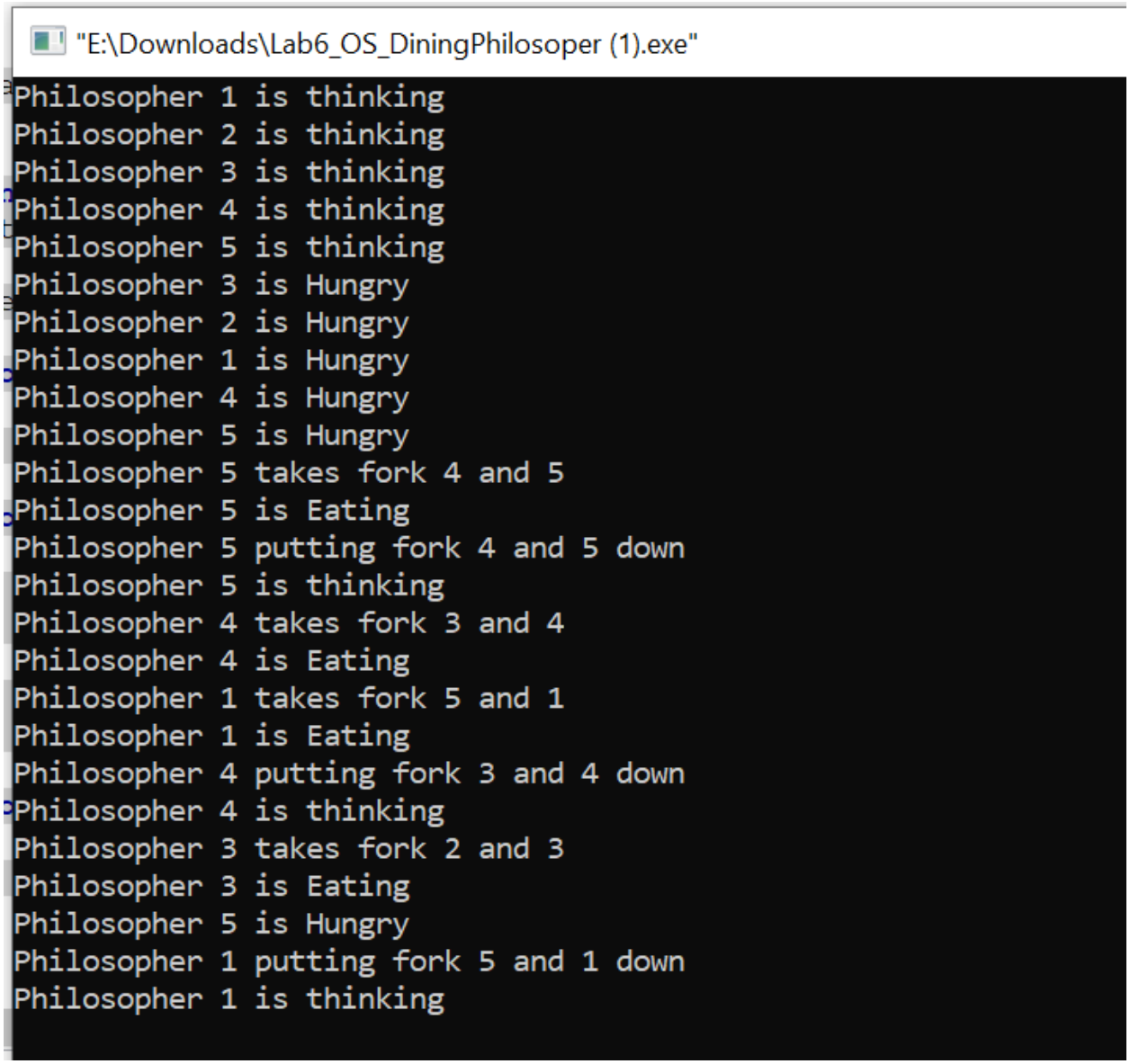
```
}

void put_fork(int phnum)
{

        sem_wait(&mutex);

        state[phnum] = THINKING;

        printf("Philosopher %d putting fork %d and %d down\n",
                phnum + 1, LEFT + 1, phnum + 1);
        printf("Philosopher %d is thinking\n", phnum + 1);

        test(LEFT);
        test(RIGHT);

        sem_post(&mutex);
}

void* philosopher(void* num)
{

        while (1) {

                int* i = num;

                sleep(1);
```

```
            take_fork(*i);

            sleep(0);

            put_fork(*i);
        }
}

int main()
{

        int i;
        pthread_t thread_id[N];

        sem_init(&mutex, 0, 1);

        for (i = 0; i < N; i++)

            sem_init(&S[i], 0, 0);

        for (i = 0; i < N; i++) {

            pthread_create(&thread_id[i], NULL,
                                philosopher, &phil[i]);

            printf("Philosopher %d is thinking\n", i + 1);
        }
```

```
        for (i = 0; i < N; i++)

                pthread_join(thread_id[i], NULL);
}
```

OUTPUT:




"E:\Downloads\Lab6_OS_DiningPhilosoper (1).exe"
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 3 is Hungry
Philosopher 2 is Hungry
Philosopher 1 is Hungry
Philosopher 4 is Hungry
Philosopher 5 is Hungry
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 5 is Hungry
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking

PROGRAM 7

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

Program:

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

void main()
{
   int alloc[10][10],max[10][10];
   int avail[10],work[10],total[10],ans[10];
   int i,j,k,s,need[10][10];
   int m,n;
   int count=0,c=0;
   char finish[10];

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

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

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

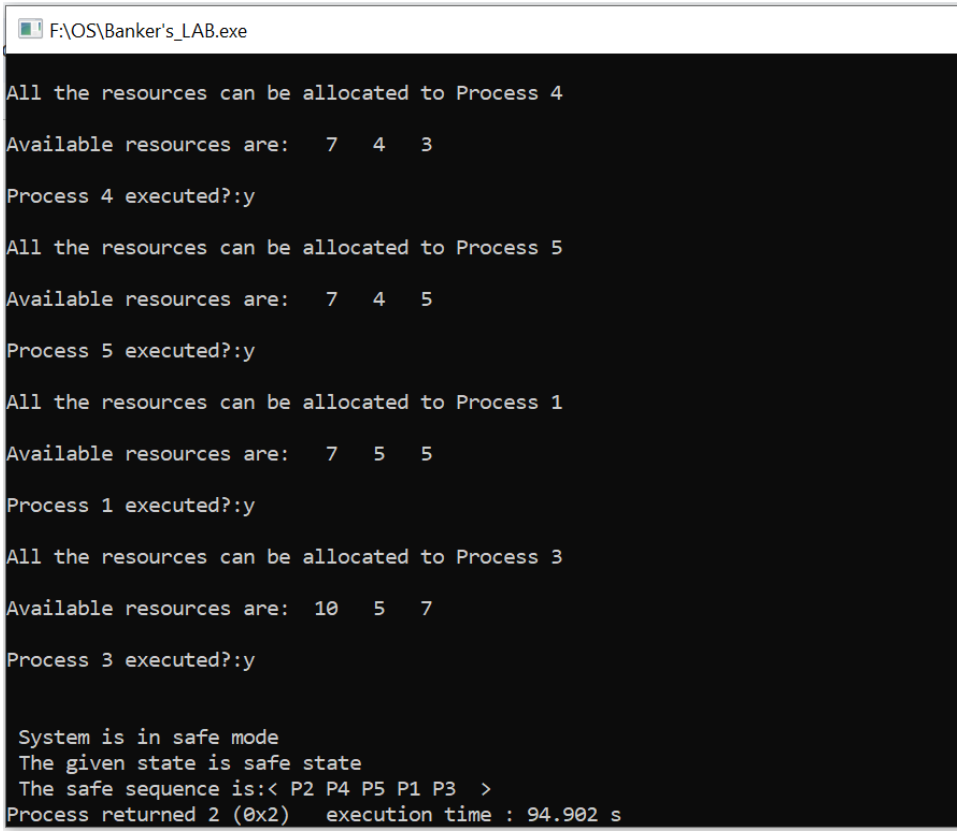
```
printf("Enter the allocation matrix:\n");
for(i=0;i<n;i++)
{
   for(j=0;j<m;j++)
      scanf("%d",&alloc[i][j]);
}
printf("Resource vector:");
for(i=0;i<m;i++)
   scanf("%d",&total[i]);

for(i=0;i<m;i++)
   avail[i]=0;

for(i=0;i<m;i++)
   work[i]=avail[i];
for(j=0;j<m;j++)
   work[j]=total[j]-work[j];

printf("\n\nNeed Matrix:\n");
for(i=0;i<n;i++)
{
    for(j=0;j<m;j++)
    {
       need[i][j]=max[i][j]-alloc[i][j];
       printf("%d ",need[i][j]);
    }
    printf("\n");
```

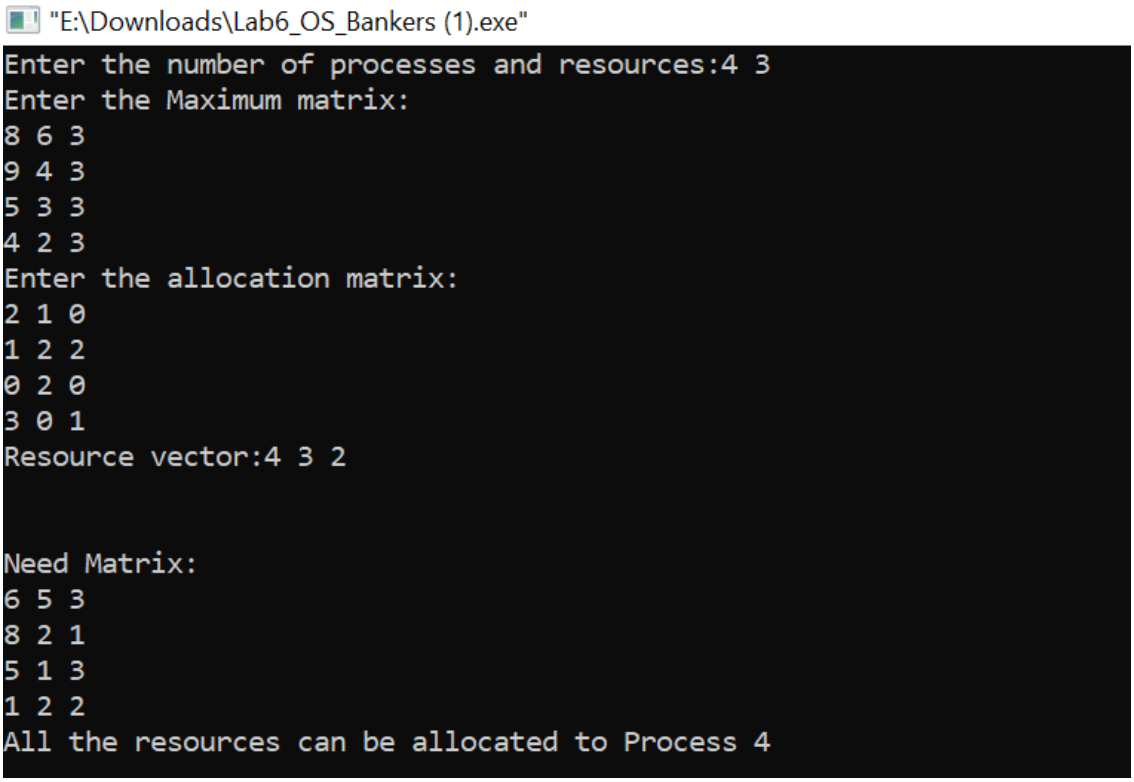
```
}
s=0;
A:
for(i=0;i<n;i++)
{
   c=0;
   for(j=0;j<m;j++)
      if((need[i][j]<=work[j])&&(finish[i]=='n'))
         c++;
   if(c==m)
   {
      printf("All the resources can be allocated to Process %d", i+1);
      printf("\n\nAvailable resources are:");
      for(k=0;k<m;k++)
      {
         work[k]+=alloc[i][k];
         printf("%4d",work[k]);
      }
      printf("\n");
      finish[i]='y';
      ans[s++]=i;
      printf("\nProcess %d executed?:%c \n\n",i+1,finish[i]); count++;
   }
}
if(count!=n)
   goto A;
else
{
```

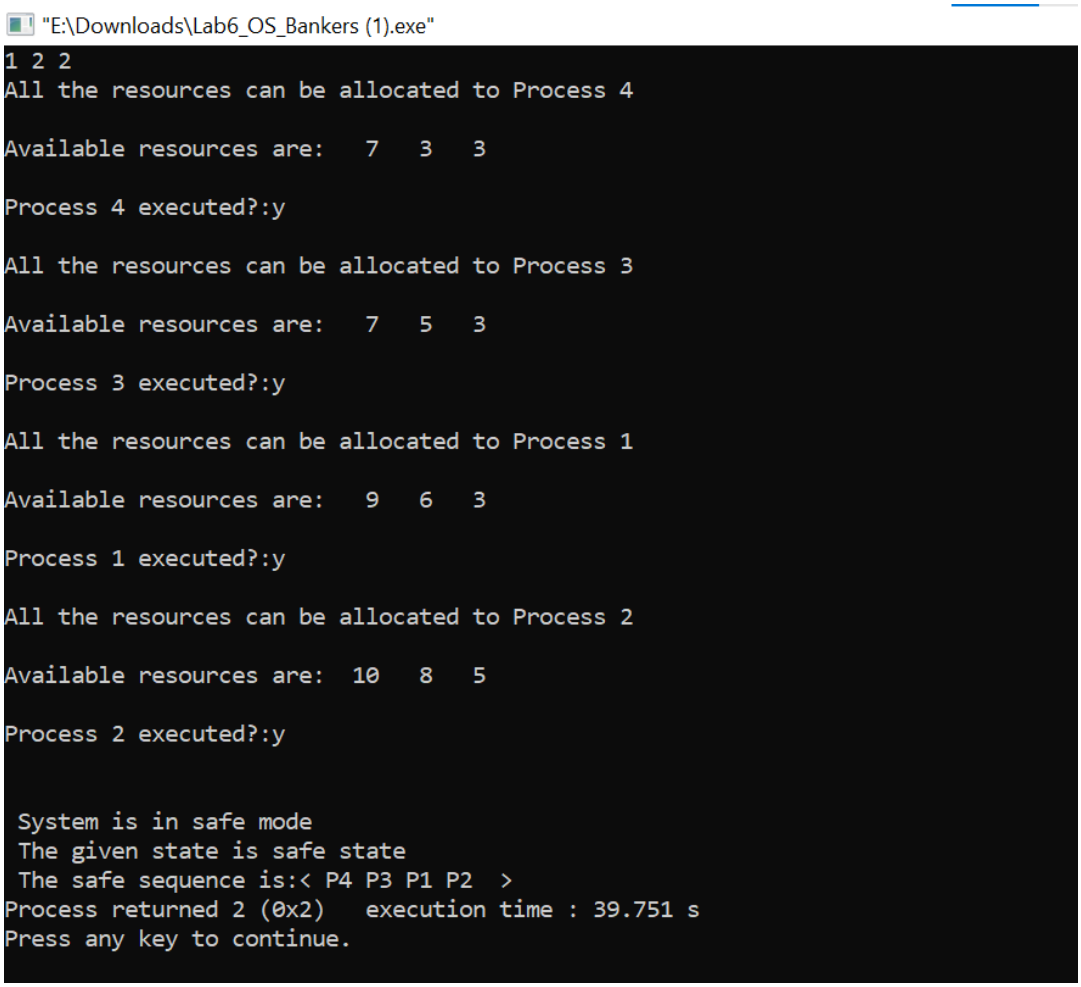
```
    printf("\n System is in safe mode");
    printf("\n The given state is safe state");
    printf("\n The safe sequence is:");
    printf("< ");
    for(i=0;i<n;i++)
      {
        printf("P%d ",ans[i]+1);
      }
    printf(" >");
  }

}
```

OUTPUT:

```
F:\OS\Banker's_LAB.exe
Enter the number of processes and resources:5 3
Enter the Maximum matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter the allocation matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Resource vector:3 3 2


Need Matrix:
7 4 3
1 2 2
6 0 0
0 1 1
4 3 1
All the resources can be allocated to Process 2

Available resources are:   5   3   2

Process 2 executed?:y

All the resources can be allocated to Process 4
```


F:\OS\Banker's_LAB.exe
All the resources can be allocated to Process 4
Available resources are: 7 4 3
Process 4 executed?:y
All the resources can be allocated to Process 5
Available resources are: 7 4 5
Process 5 executed?:y
All the resources can be allocated to Process 1
Available resources are: 7 5 5
Process 1 executed?:y
All the resources can be allocated to Process 3
Available resources are: 10 5 7
Process 3 executed?:y
System is in safe mode
The given state is safe state
The safe sequence is:< P2 P4 P5 P1 P3 >
Process returned 2 (0x2) execution time : 94.902 s


"E:\Downloads\Lab6_OS_Bankers (1).exe"
Enter the number of processes and resources:4 3
Enter the Maximum matrix:
8 6 3
9 4 3
5 3 3
4 2 3
Enter the allocation matrix:
2 1 0
1 2 2
0 2 0
3 0 1
Resource vector:4 3 2
Need Matrix:
6 5 3
8 2 1
5 1 3
1 2 2
All the resources can be allocated to Process 4


"E:\Downloads\Lab6_OS_Bankers (1).exe"
1 2 2
All the resources can be allocated to Process 4
Available resources are: 7 3 3
Process 4 executed?:y
All the resources can be allocated to Process 3
Available resources are: 7 5 3
Process 3 executed?:y
All the resources can be allocated to Process 1
Available resources are: 9 6 3
Process 1 executed?:y
All the resources can be allocated to Process 2
Available resources are: 10 8 5
Process 2 executed?:y
System is in safe mode
The given state is safe state
The safe sequence is:< P4 P3 P1 P2 >
Process returned 2 (0x2) execution time : 39.751 s
Press any key to continue.

PROGRAM 8

Write a C program to simulate deadlock detection

Program:

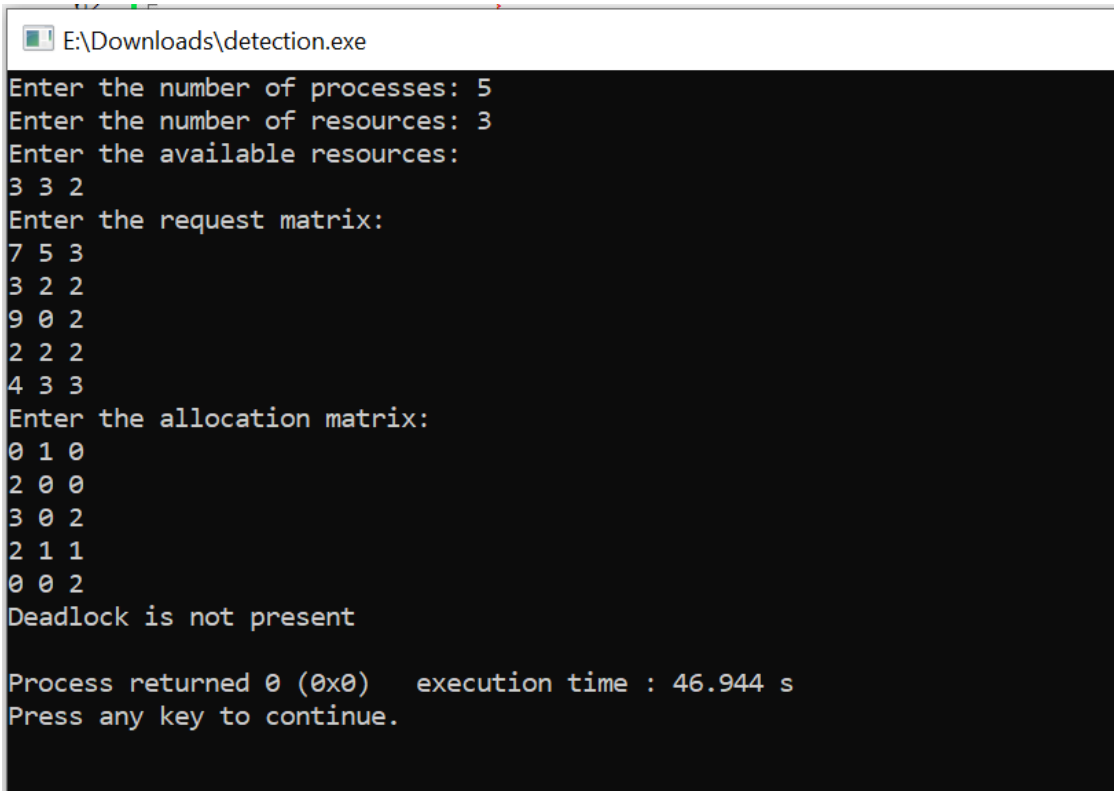
```
#include <stdio.h>

#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

int available[MAX_RESOURCES];
int max[MAX_PROCESSES][MAX_RESOURCES];
int allocation[MAX_PROCESSES][MAX_RESOURCES];
int need[MAX_PROCESSES][MAX_RESOURCES];
int num_processes, num_resources;

void inputData() {
    printf("Enter the number of processes: ");
    scanf("%d", &num_processes);

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

    printf("Enter the available resources:\n");
    for (int i = 0; i < num_resources; i++) {
        scanf("%d", &available[i]);
    }
    printf("Enter the maximum demand matrix:\n");
    for (int i = 0; i < num_processes; i++) {
```

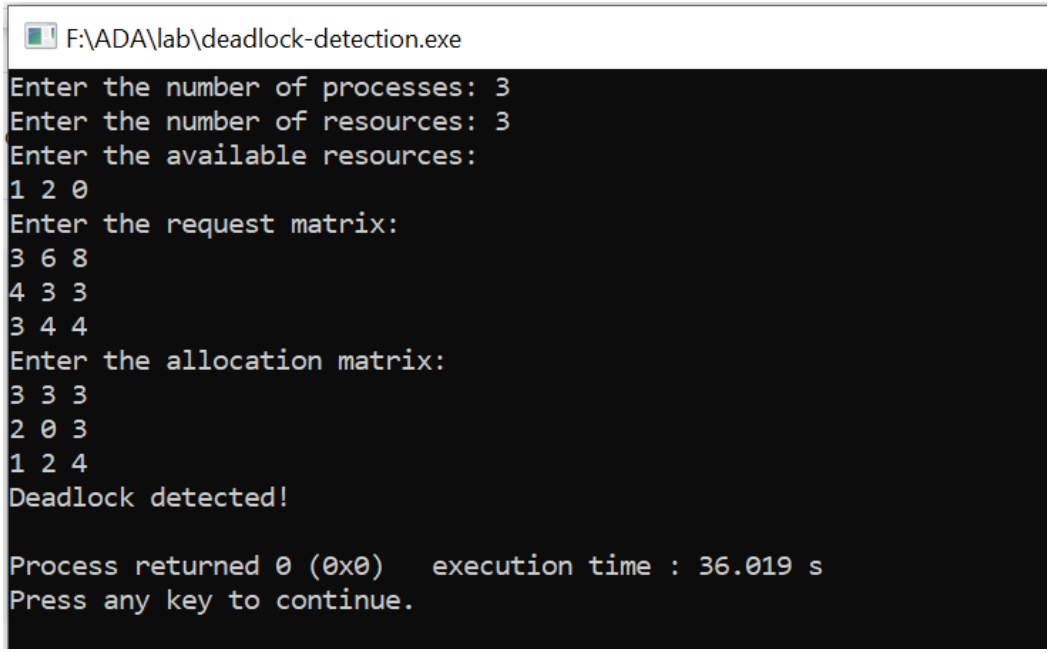
```
        for (int j = 0; j < num_resources; j++) {
            scanf("%d", &max[i][j]);
        }
    }
    printf("Enter the allocation matrix:\n");
    for (int i = 0; i < num_processes; i++) {
        for (int j = 0; j < num_resources; j++) {
            scanf("%d", &allocation[i][j]);
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
}
int checkSafety(int process_order[]) {
    int work[MAX_RESOURCES];
    int finish[MAX_PROCESSES] = {0};

    for (int i = 0; i < num_resources; i++) {
        work[i] = available[i];
    }
    int completed = 0;
    while (completed < num_processes) {
        int found = 0;
        for (int i = 0; i < num_processes; i++) {
            if (!finish[i]) {
                int j;
                for (j = 0; j < num_resources; j++) {
                    if (need[i][j] > work[j]) {
                        break;
```

```
                }
            }
            if (j == num_resources) {
                for (int k = 0; k < num_resources; k++) {
                    work[k] += allocation[i][k];
                }
                finish[i] = 1;
                process_order[completed] = i;
                completed++;
                found = 1;
            }
        }
    }
    if (!found) {
        return 0; // Deadlock detected
    }
  }
  return 1; // System is safe
}
int main() {
  int process_order[MAX_PROCESSES];
  inputData();
  int flag = checkSafety(process_order);
  if (flag == 1) {
    printf("Deadlock is not present \n");
  } else {
    printf("Deadlock detected!\n");
  }
```

```
    return 0;
}
```

OUTPUT:





PROGRAM 9

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

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

Program:

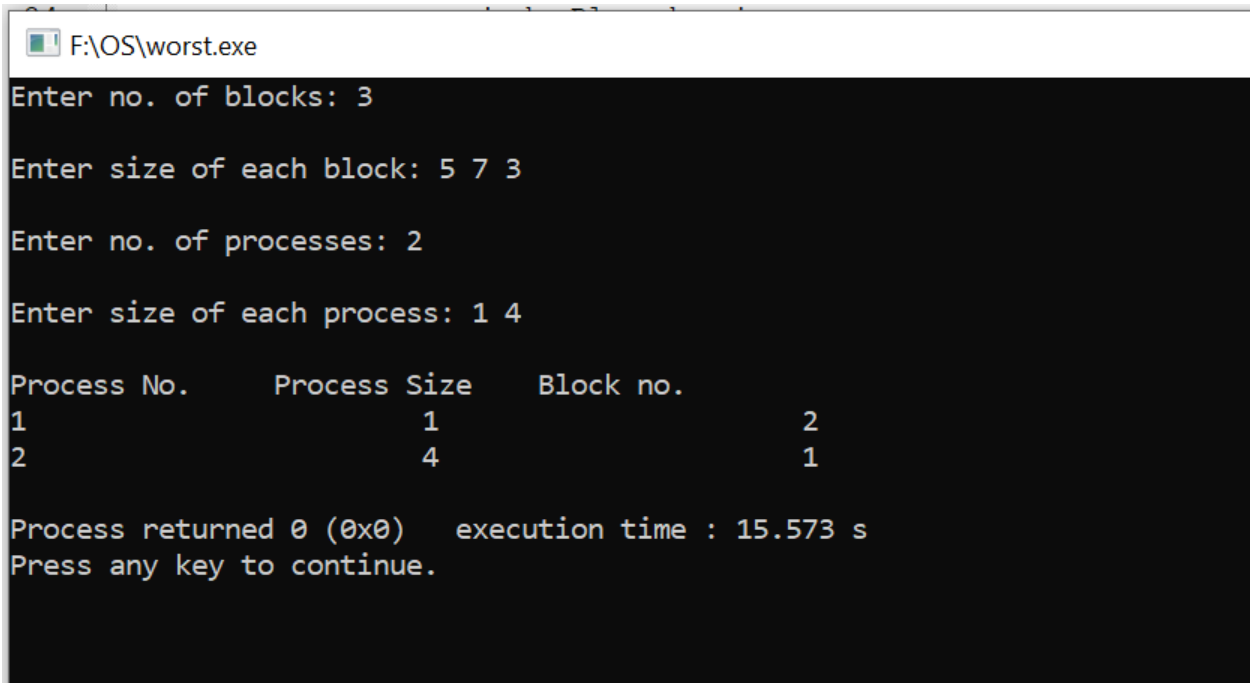
a) Worst-fit

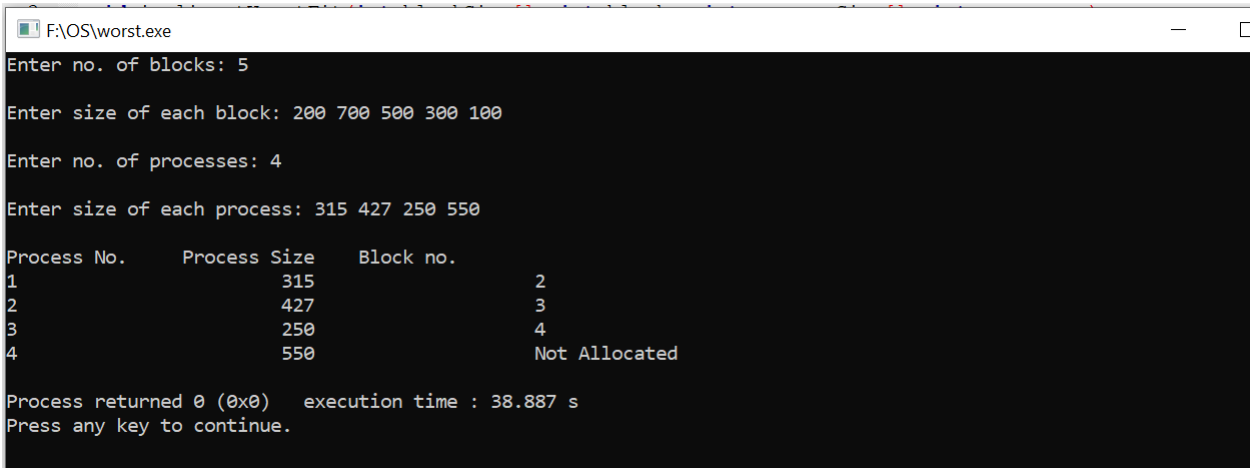
```
#include <stdio.h>


void implimentWorstFit(int blockSize[], int blocks, int processSize[], int processes)
{
   int allocation[processes];
   int occupied[blocks];
   for(int i = 0; i < processes; i++){
      allocation[i] = -1;
   }
   for(int i = 0; i < blocks; i++){
      occupied[i] = 0;
   }
   for (int i=0; i < processes; i++)
   {
int indexPlaced = -1;
for(int j = 0; j < blocks; j++)
{
  if(blockSize[j] >= processSize[i] && !occupied[j])
          {
             if (indexPlaced == -1)
                indexPlaced = j;
```

```
            else if (blockSize[indexPlaced] < blockSize[j])
                indexPlaced = j;
        }
      }

      if (indexPlaced != -1)
      {
          allocation[i] = indexPlaced;
          occupied[indexPlaced] = 1;
          blockSize[indexPlaced] -= processSize[i];
      }
   }
   printf("\nProcess No.\tProcess Size\tBlock no.\n");
   for (int i = 0; i < processes; i++)
   {
      printf("%d \t\t\t %d \t\t\t", i+1, processSize[i]);
      if (allocation[i] != -1)
          printf("%d\n",allocation[i] + 1);
      else
          printf("Not Allocated\n");
   }
}

int main()
{
   int i;
   int blocks;
   int processes;
```

```
    printf("Enter no. of blocks: ");
scanf("%d", &blocks);
int blockSize[blocks];
printf("\nEnter size of each block: ");
for(i = 0; i < blocks; i++)
scanf("%d", &blockSize[i]);
printf("\nEnter no. of processes: ");
scanf("%d", &processes);
   int processSize[processes];

printf("\nEnter size of each process: ");
for(i = 0; i < processes; i++)
scanf("%d", &processSize[i]);

    implimentWorstFit(blockSize, blocks, processSize, processes);

    return 0;
}
```

OUTPUT





b) Best-fit :

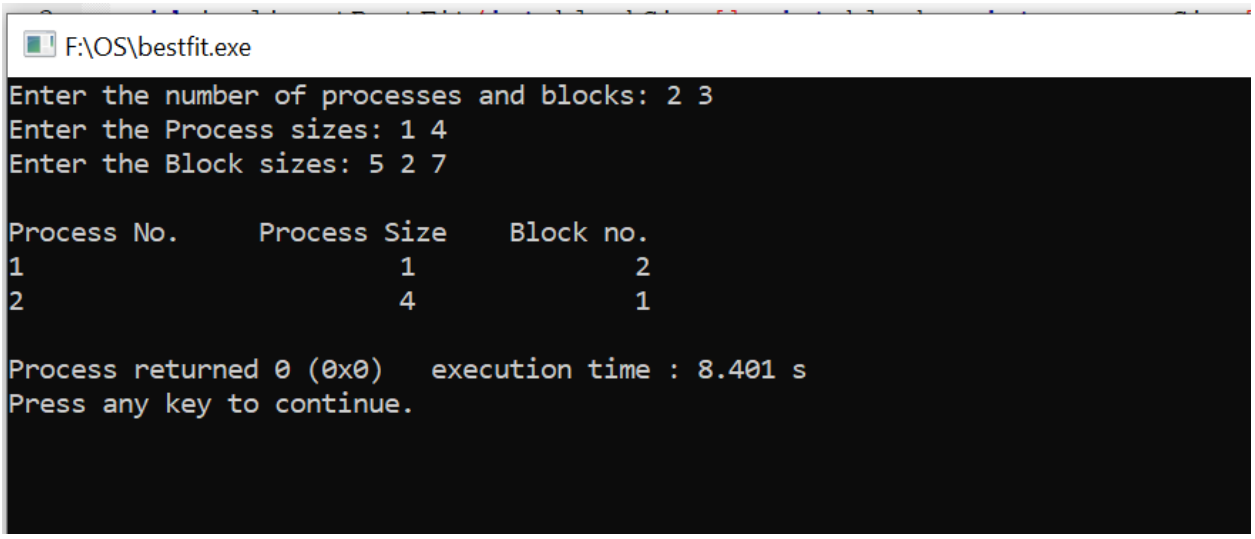
```
#include <stdio.h>
#define MAX 10
void implimentBestFit(int blockSize[], int blocks, int processSize[], int proccesses,int m)
{
   int allocation[proccesses];
   int occupied[blocks];
```

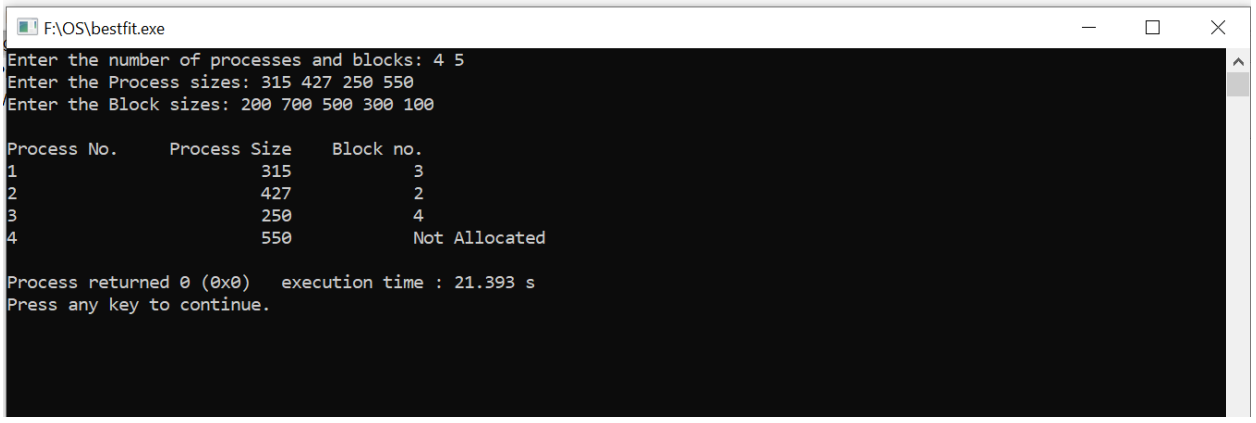
```
for(int i = 0; i < proccesses; i++){
   allocation[i] = -1;
}

for(int i = 0; i < blocks; i++){
   occupied[i] = 0;
}

for (int i = 0; i < proccesses; i++)
{

   int indexPlaced = -1;
   for (int j = 0; j < blocks; j++) {
      if (blockSize[j] >= processSize[i] && !occupied[j])
      {
         if (indexPlaced == -1)
            indexPlaced = j;
         else if (blockSize[j] < blockSize[indexPlaced])
            indexPlaced = j;
      }
   }

   if (indexPlaced != -1)
   {
      allocation[i] = indexPlaced;

      occupied[indexPlaced] = 1;
```

```
        }
    }

    printf("\nProcess No.\tProcess Size\tBlock no.\n");
    for (int i = 0; i < proccesses; i++)
    {
        printf("%d \t\t\t %d \t\t", i+1, processSize[i]);
        if (allocation[i] != -1)
            printf("%d\n",allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}

int main()
{
    int p,m,j;
    printf("Enter the number of processes and blocks: ");
    scanf("%d%d",&p,&m);
    int processSize[p],blockSize[m];
    printf("Enter the Process sizes: ");
    for(j=0;j<p;j++)
            scanf("%d",&processSize[j]);

    printf("Enter the Block sizes: ");
    for(j=0;j<m;j++)
            scanf("%d",&blockSize[j]);
```

```
    int blocks = sizeof(blockSize)/sizeof(blockSize[0]);
    int proccesses = sizeof(processSize)/sizeof(processSize[0]);


    implimentBestFit(blockSize, blocks, processSize, proccesses,m);


    return 0 ;
}
```

OUTPUT:



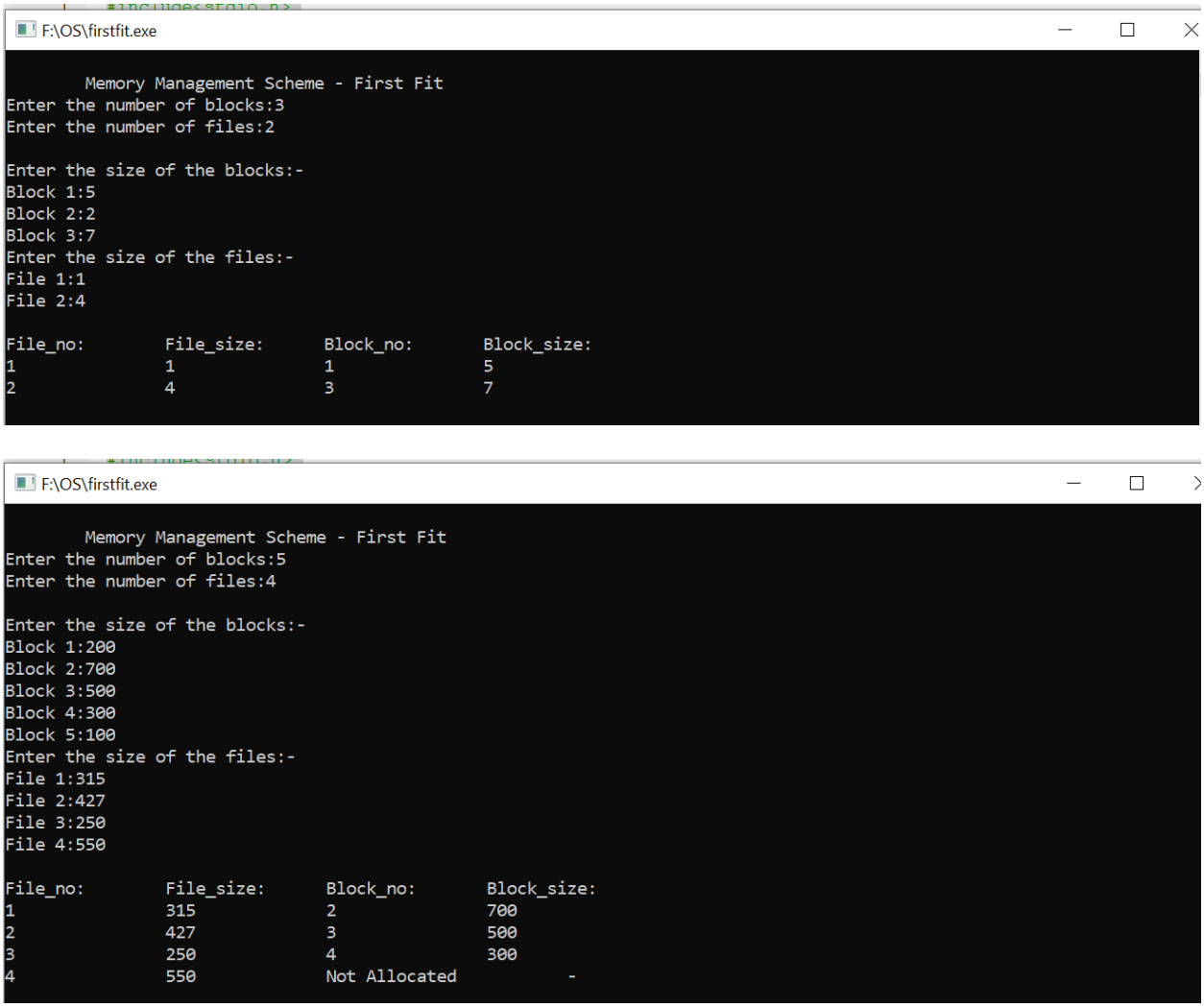


c) First-fit:

```
#include<stdio.h>
#include<conio.h>
#define max 25

void main() {
   int frag[max], b[max], f[max], i, j, nb, nf, temp;
   static int bf[max], ff[max];

   printf("\n\tMemory Management Scheme - First Fit");
   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 (i = 1; i <= nb; i++) {
      printf("Block %d:", i);
      scanf("%d", &b[i]);
   }

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

```
    for (i = 1; i <= nf; i++) {
        int allocated = 0;
        for (j = 1; j <= nb; j++) {
            if (bf[j] != 1) { // If bf[j] is not allocated
                temp = b[j] - f[i];
                if (temp >= 0) {
                    ff[i] = j;
                    bf[j] = 1; // Allocate block j to file i
                    frag[i] = b[j] - f[i]; // Remaining space in the block
                    allocated = 1;
                    printf("\n%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]]);
                    break; // Stop searching for blocks
                }
            }
        }
        if (!allocated) {
            printf("\n%d\t\t%d\t\tNot Allocated\t\t-", i, f[i]);
        }
    }
    getch();
}
```

OUTPUT:


F:\OS\firstfit.exe
Memory Management Scheme - First Fit
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
File_no: File_size: Block_no: Block_size:
1 1 1 5
2 4 3 7
F:\OS\firstfit.exe
Memory Management Scheme - First Fit
Enter the number of blocks:5
Enter the number of files:4
Enter the size of the blocks:-
Block 1:200
Block 2:700
Block 3:500
Block 4:300
Block 5:100
Enter the size of the files:-
File 1:315
File 2:427
File 3:250
File 4:550
File_no: File_size: Block_no: Block_size:
1 315 2 700
2 427 3 500
3 250 4 300
4 550 Not Allocated -

PROGRAM 10

Write a C program to simulate page replacement algorithms

a) FIFO b) LRU c) Optimal

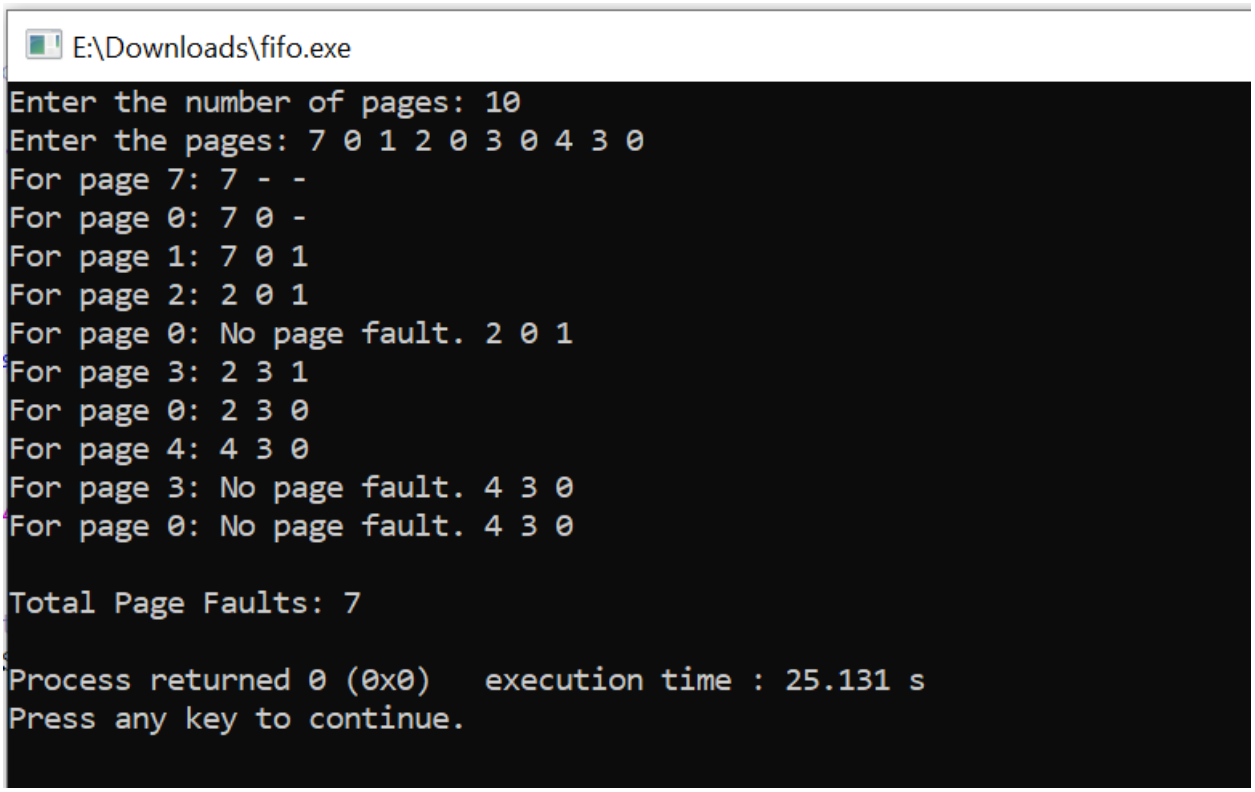
a) FIFO:

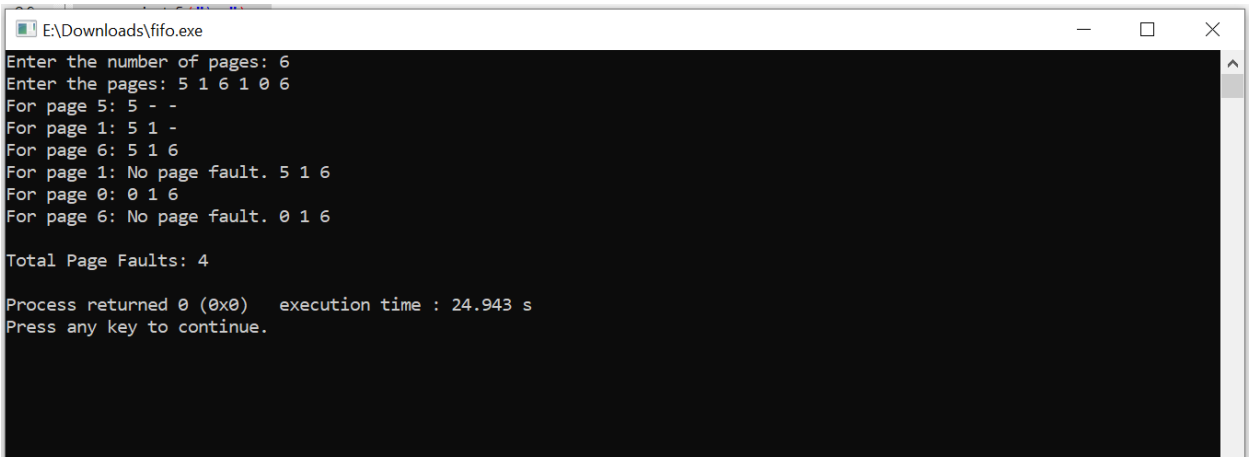
```
#include <stdio.h>
#define MAX_FRAMES 3 // Number of memory frames
int frames[MAX_FRAMES]; // Array to hold memory frames
int framePointer = 0;   // Pointer to the current frame being replaced
int pageFaultCount = 0; // Counter for page faults

int isPageInMemory(int page) {
    for (int i = 0; i < MAX_FRAMES; i++) {
        if (frames[i] == page) {
            return 1; // Page is in memory
        }
    }
    return 0; // Page is not in memory
}

void displayFrames() {
    //printf("Memory frames: ");
    for (int i = 0; i < MAX_FRAMES; i++) {
        if (frames[i] != -1) {
            printf("%d ", frames[i]);
        } else {
            printf("- ");
        }
```

```
    }
    printf("\n");
}

// Function to simulate page replacement using FIFO algorithm
void fifo(int pages[], int pageCount) {
    for (int i = 0; i < pageCount; i++) {
        printf("For page %d: ", pages[i]);

        if (!isPageInMemory(pages[i])) {
            frames[framePointer] = pages[i];
            framePointer = (framePointer + 1) % MAX_FRAMES;
            pageFaultCount++;
        } else {
            printf("No page fault. ");
        }

        displayFrames();
    }

    printf("\nTotal Page Faults: %d\n", pageFaultCount);
}

int main() {
    int pageCount;
    printf("Enter the number of pages: ");
    scanf("%d",&pageCount);
    int pages[pageCount];
```

```
    printf("Enter the pages: ");
    for(int i=0;i<pageCount;i++){
        scanf("%d",&pages[i]);
    }

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

    fifo(pages, pageCount);

    return 0;
}
```

OUTPUT:





b) LRU:

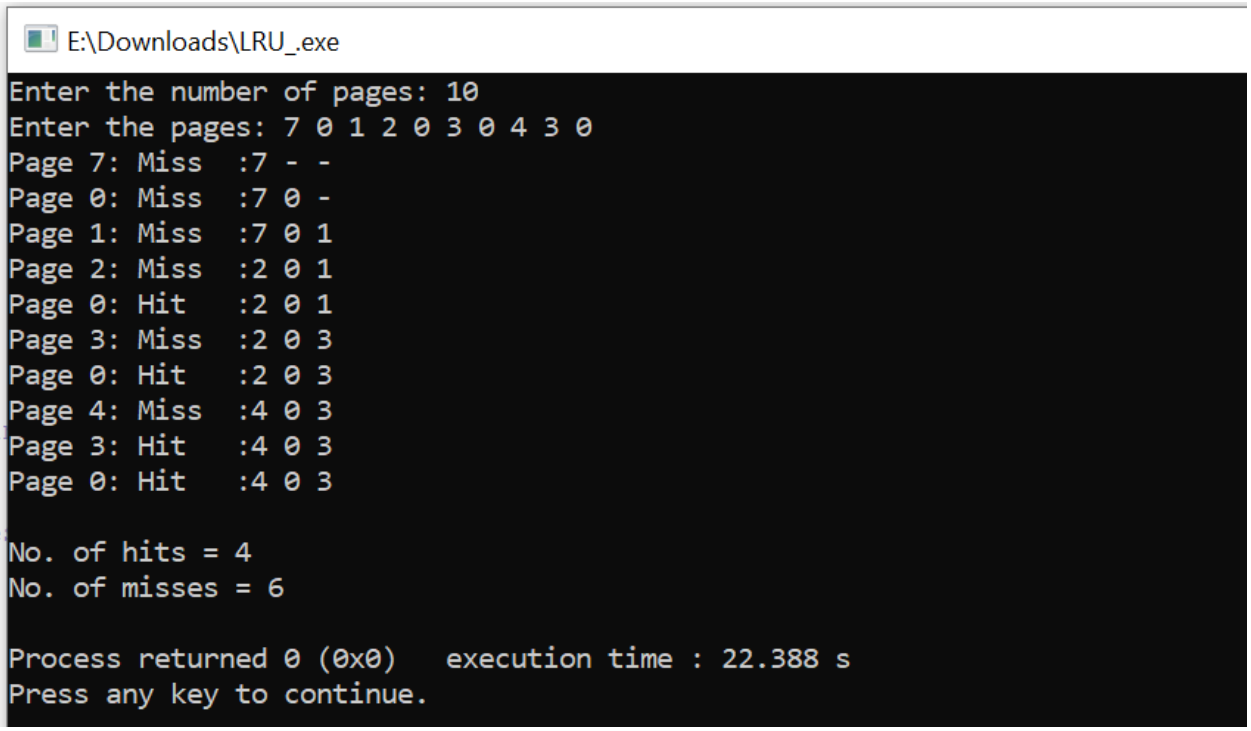
```
#include <stdio.h>
void displayFrames(int fr[], int fn) {
    for (int i = 0; i < fn; i++) {
        if (fr[i] != -1) {
            printf("%d ", fr[i]);
        } else {
            printf("- ");
        }
    }
    printf("\n");
}
int isPageInMemory(int page,int fn,int frames[fn])
{
    for (int i = 0; i < fn; i++) {
        if (frames[i] == page) {
            return 1;
        }
    }
```

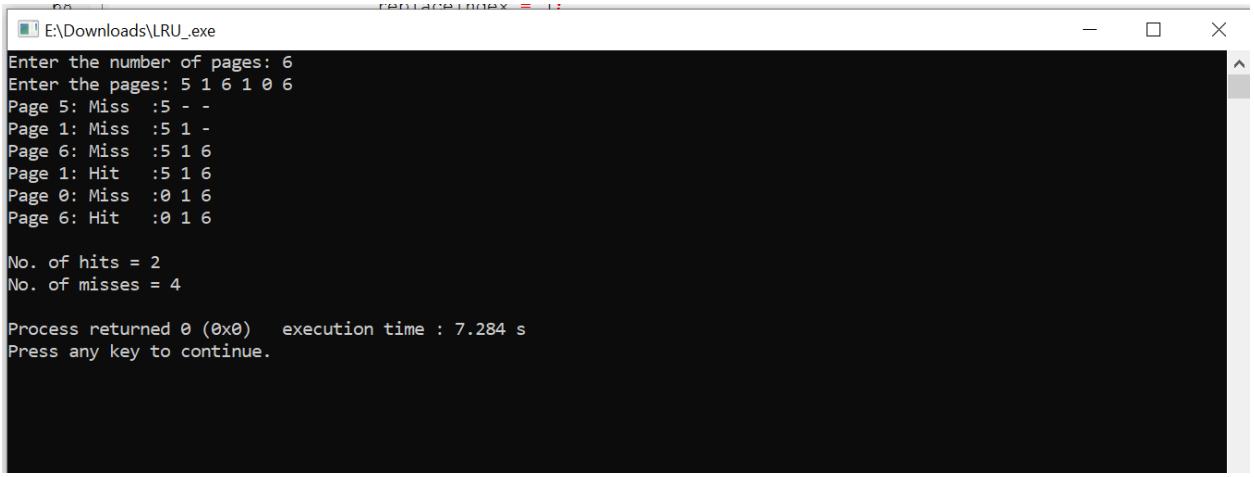
```
    return 0;
}
void lruPage(int pg[], int pn, int fn)
{
    int fr[fn];
    for (int i = 0; i < fn; i++)
        fr[i] = -1;

    int hit = 0;

    for (int i = 0; i < pn; i++)
    {

        if (isPageInMemory(pg[i],fn,fr))
        {
            hit++;
            printf("Page %d: Hit   :", pg[i]);
        }
        else
        {
            int emptyFrame = -1;
            for (int j = 0; j < fn; j++)
            {
                if (fr[j] == -1) {
                    fr[j] = pg[i];
                    emptyFrame = j;
                    break;
                }
```

```
    }

    if (emptyFrame != -1)
        printf("Page %d: Miss  :", pg[i]);
    else {
        int minCounter = pn + 1, replaceIndex = -1;
        for (int j = 0; j < fn; j++)
        {
            int k;
            for (k = i - 1; k >= 0; k--) {
                if (fr[j] == pg[k]) {
                    if (k < minCounter) {
                        minCounter = k;
                        replaceIndex = j;
                    }
                    break;
                }
            }
            if (k == -1) {
                replaceIndex = j;
                break;
            }
        }
        fr[replaceIndex] = pg[i];
        printf("Page %d: Miss  :", pg[i]);
    }
}
```

```
        displayFrames(fr, fn);
    }

    printf("\nNo. of hits = %d\n", hit);
    printf("No. of misses = %d\n", pn - hit);
}

int main() {
    int pn;
    printf("Enter the number of pages: ");
    scanf("%d", &pn);
    int pg[pn];
    printf("Enter the pages: ");
    for (int i = 0; i < pn; i++) {
        scanf("%d", &pg[i]);
    }
    int fn = 3;
    lruPage(pg, pn, fn);
    return 0;
}
```

OUTPUT:





c) OPTIMAL:

```
#include <stdio.h>

void displayFrames(int fr[], int fn) {
   //printf("Memory frames: ");
   for (int i = 0; i < fn; i++) {
      if (fr[i] != -1) {
```

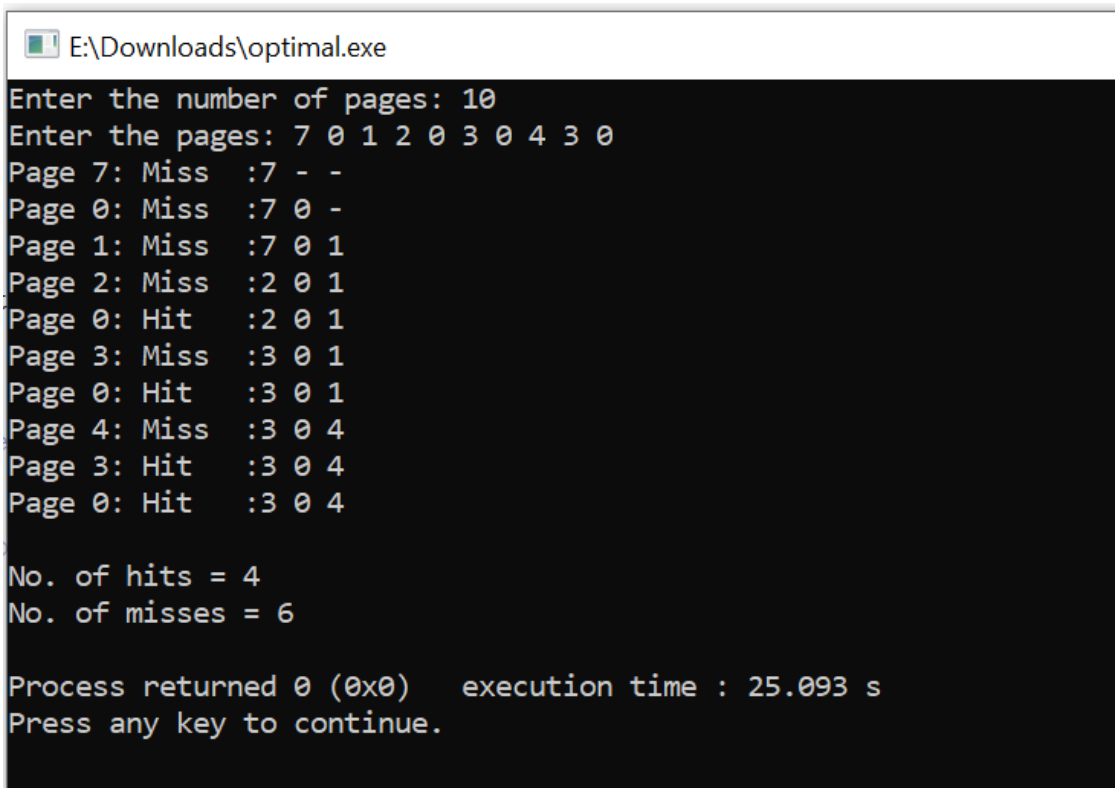
```
            printf("%d ", fr[i]);
        } else {
            printf("- ");
        }
    }
    printf("\n");
}

void optimalPage(int pg[], int pn, int fn) {
    // Create an array for given number of frames and initialize it as empty.
    int fr[fn];
    for (int i = 0; i < fn; i++) {
        fr[i] = -1;
    }

    // Traverse through page reference array and check for miss and hit.
    int hit = 0;
    for (int i = 0; i < pn; i++) {
        // Page found in a frame: HIT
        int found = 0;
        for (int j = 0; j < fn; j++) {
            if (fr[j] == pg[i]) {
                hit++;
                found = 1;
                break;
            }
        }
```

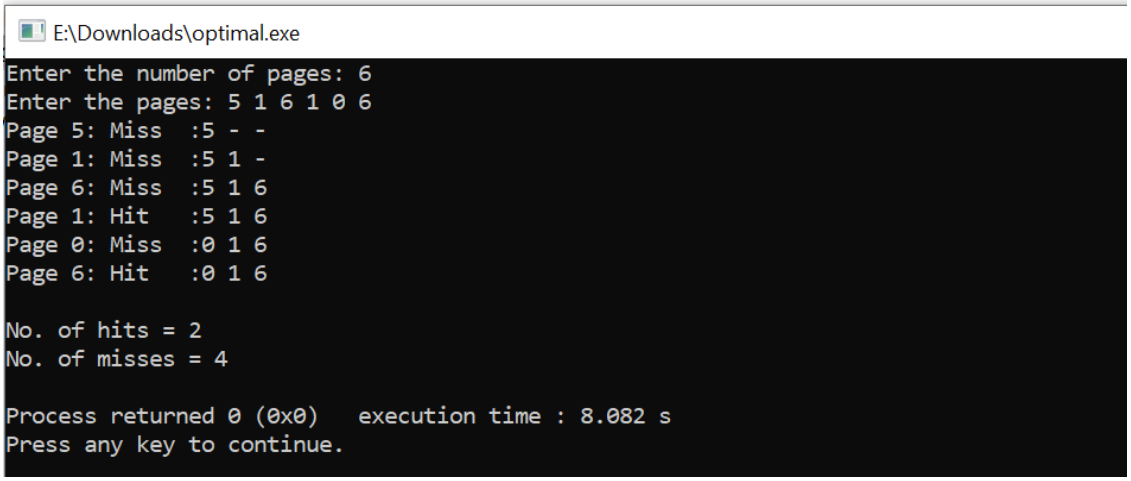
```
if (found) {
    printf("Page %d: Hit   :", pg[i]);
} else {
    // If there is space available in frames.
    int emptyFrame = -1;
    for (int j = 0; j < fn; j++) {
        if (fr[j] == -1) {
            fr[j] = pg[i];
            emptyFrame = j;
            break;
        }
    }

    if (emptyFrame != -1) {
        printf("Page %d: Miss  :", pg[i], pg[i]);
    } else {
        // Find the page to be replaced.
        int farthest = -1, replaceIndex = -1;
        for (int j = 0; j < fn; j++) {
            int k;
            for (k = i + 1; k < pn; k++) {
                if (fr[j] == pg[k]) {
                    if (k > farthest) {
                        farthest = k;
                        replaceIndex = j;
                    }
                    break;
                }
```

```
                }
                if (k == pn) {
                    replaceIndex = j;
                    break;
                }
            }
            fr[replaceIndex] = pg[i];
            printf("Page %d: Miss  :", pg[i], fr[replaceIndex]);
        }
    }

    // Display the contents of memory frames after each iteration.
    displayFrames(fr, fn);
  }

  printf("\nNo. of hits = %d\n", hit);
  printf("No. of misses = %d\n", pn - hit);
}

int main() {
  int pn;
  printf("Enter the number of pages: ");
  scanf("%d",&pn);
  int pg[pn];
  printf("Enter the pages: ");
  for(int i=0;i<pn;i++){
    scanf("%d",&pg[i]);
  }
```

```
    int fn = 3;
    optimalPage(pg, pn, fn);
    return 0;
}
```

OUTPUT:





PROGRAM 11

Write a C program to simulate disk scheduling algorithms a) FCFS b) SCAN c) C-SCAN

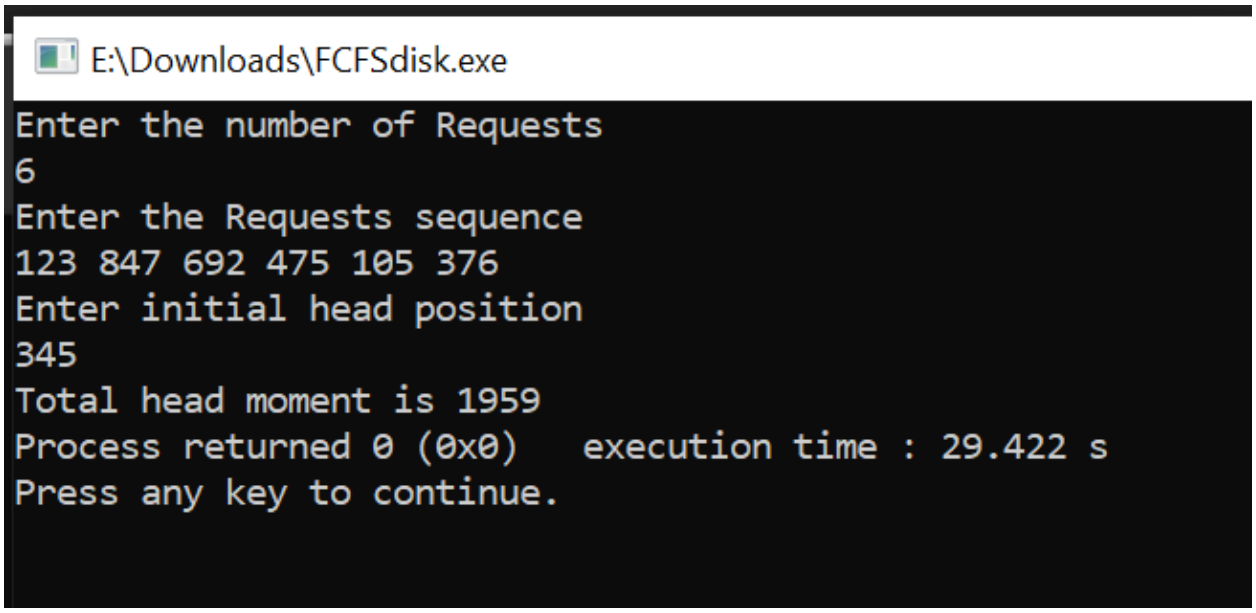
Programs:

a) FCFS:

```
/*FCFS*/
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int RQ[100], i, n, TotalHeadMoment = 0, initial;
   printf("Enter the number of Requests\n");
   scanf("%d", &n);
   printf("Enter the Requests sequence\n");
   for (i = 0; i < n; i++)
      scanf("%d", &RQ[i]);
   printf("Enter initial head position\n");
   scanf("%d", &initial);

   // logic for FCFS disk scheduling

   for (i = 0; i < n; i++)
   {
      TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
      initial = RQ[i];
   }

   printf("Total head moment is %d", TotalHeadMoment);
```

```
    return 0;
}
```

OUTPUT:



b) SCAN:

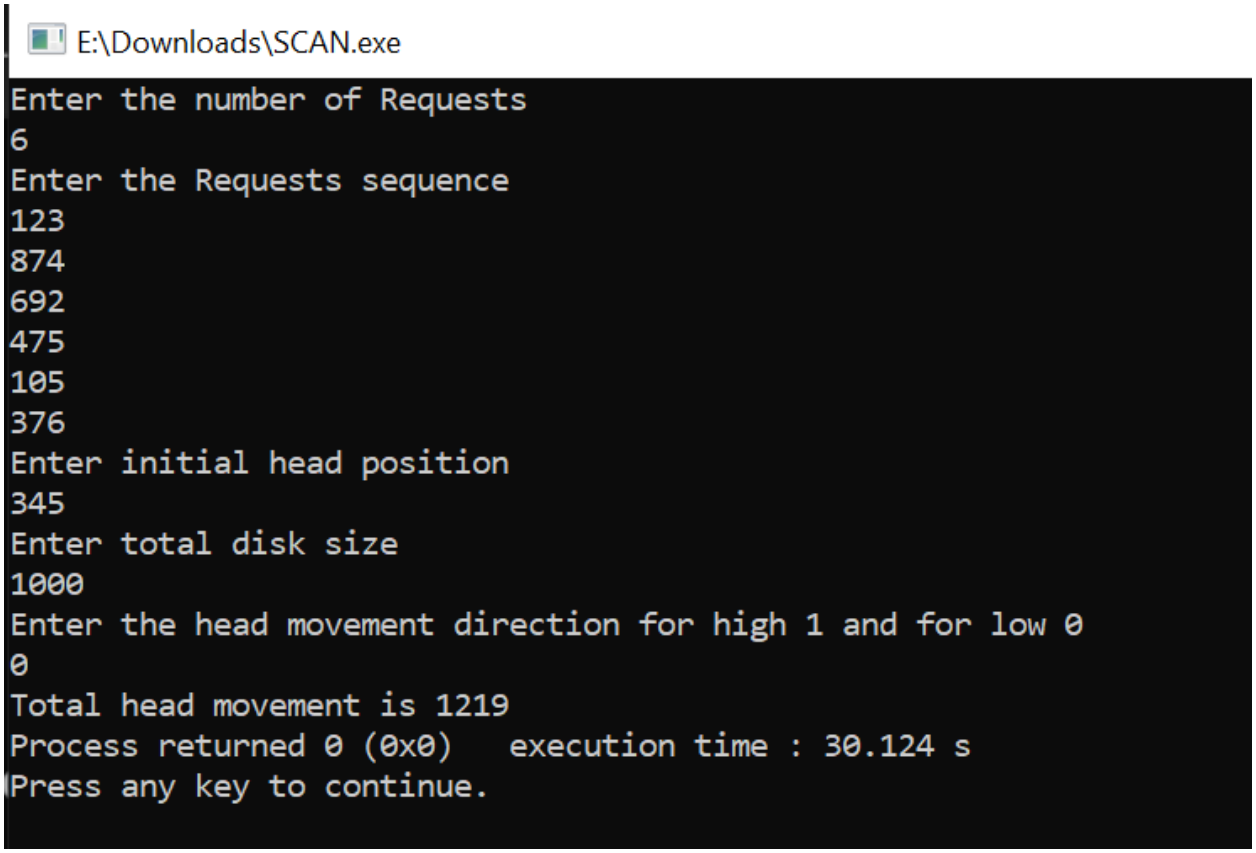
```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int RQ[100], i, j, n, TotalHeadMoment = 0, initial, size, move;
    printf("Enter the number of Requests\n");
    scanf("%d", &n);
    printf("Enter the Requests sequence\n");
    for (i = 0; i < n; i++)
        scanf("%d", &RQ[i]);
    printf("Enter initial head position\n");
```

```
scanf("%d", &initial);
printf("Enter total disk size\n");
scanf("%d", &size);
printf("Enter the head movement direction for high 1 and for low 0\n");
scanf("%d", &move);
for (i = 0; i < n; i++)
{
   for (j = 0; j < n - i - 1; j++)
   {
      if (RQ[j] > RQ[j + 1])
      {
         int temp;
         temp = RQ[j];
         RQ[j] = RQ[j + 1];
         RQ[j + 1] = temp;
      }
   }
}

int index;
for (i = 0; i < n; i++)
{
   if (initial < RQ[i])
   {
      index = i;
      break;
   }
}
```

```
if (move == 1)
{
   for (i = index; i < n; i++)
   {
      TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
      initial = RQ[i];
   }
   TotalHeadMoment = TotalHeadMoment + abs(size - RQ[i - 1] - 1);
   initial = size - 1;
   for (i = index - 1; i >= 0; i--)
   {
      TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
      initial = RQ[i];
   }
}
else
{
   for (i = index - 1; i >= 0; i--)
   {
      TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
      initial = RQ[i];
   }
   //  last movement for min size
   TotalHeadMoment = TotalHeadMoment + abs(RQ[i + 1] - 0);
   initial = 0;
   for (i = index; i < n; i++)
   {
      TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
```

```
            initial = RQ[i];
        }
    }

    printf("Total head movement is %d", TotalHeadMoment);
    return 0;
}
```

OUTPUT:



c) CSCAN:

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

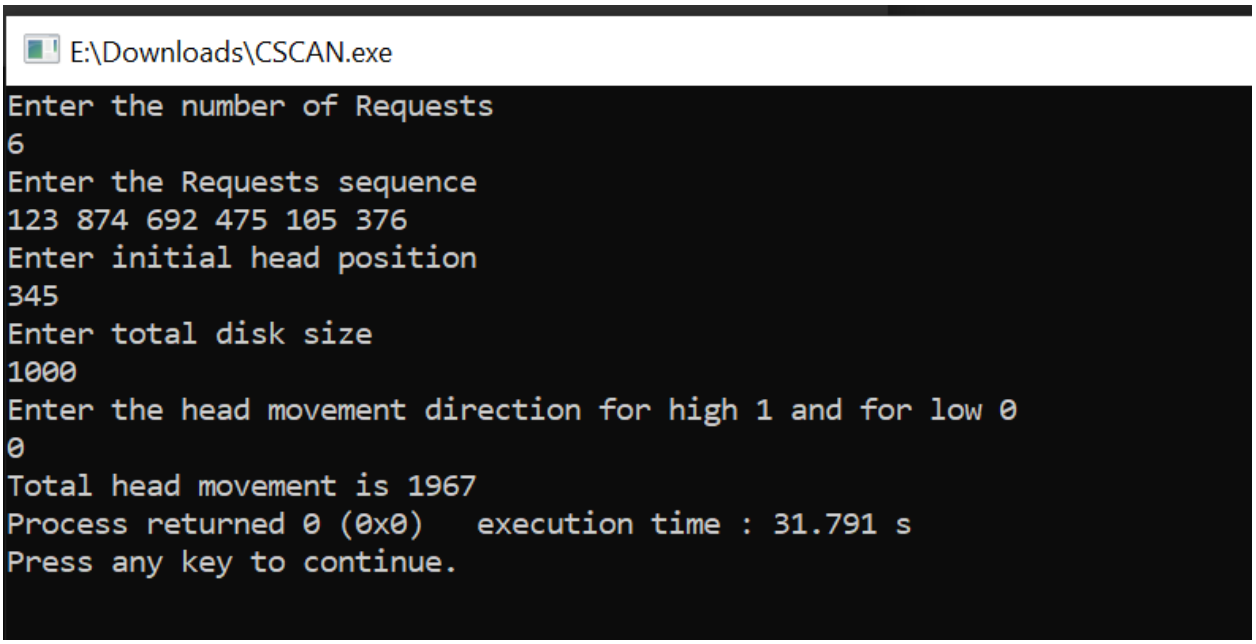
```
int main()
{
   int RQ[100], i, j, n, TotalHeadMoment = 0, initial, size, move;
   printf("Enter the number of Requests\n");
   scanf("%d", &n);
   printf("Enter the Requests sequence\n");
   for (i = 0; i < n; i++)
      scanf("%d", &RQ[i]);
   printf("Enter initial head position\n");
   scanf("%d", &initial);
   printf("Enter total disk size\n");
   scanf("%d", &size);
   printf("Enter the head movement direction for high 1 and for low 0\n");
   scanf("%d", &move);

   // logic for C-Scan disk scheduling

   /*logic for sort the request array */
   for (i = 0; i < n; i++)
   {
      for (j = 0; j < n - i - 1; j++)
      {
         if (RQ[j] > RQ[j + 1])
         {
            int temp;
            temp = RQ[j];
            RQ[j] = RQ[j + 1];
```

```
            RQ[j + 1] = temp;
        }
    }
}

int index;
for (i = 0; i < n; i++)
{
    if (initial < RQ[i])
    {
        index = i;
        break;
    }
}

// if movement is towards high value
if (move == 1)
{
    for (i = index; i < n; i++)
    {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
        initial = RQ[i];
    }
    //  last movement for max size
    TotalHeadMoment = TotalHeadMoment + abs(size - RQ[i - 1] - 1);
    /*movement max to min disk */
    TotalHeadMoment = TotalHeadMoment + abs(size - 1 - 0);
    initial = 0;
```

```
    for (i = 0; i < index; i++)
    {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
        initial = RQ[i];
    }
}
// if movement is towards low value
else
{
    for (i = index - 1; i >= 0; i--)
    {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
        initial = RQ[i];
    }
    //  last movement for min size
    TotalHeadMoment = TotalHeadMoment + abs(RQ[i + 1] - 0);
    /*movement min to max disk */
    TotalHeadMoment = TotalHeadMoment + abs(size - 1 - 0);
    initial = size - 1;
    for (i = n - 1; i >= index; i--)
    {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
        initial = RQ[i];
    }
}

printf("Total head movement is %d", TotalHeadMoment);
return 0;
```

}

OUTPUT:


E:\Downloads\CSCAN.exe
Enter the number of Requests
6
Enter the Requests sequence
123 874 692 475 105 376
Enter initial head position
345
Enter total disk size
1000
Enter the head movement direction for high 1 and for low 0
0
Total head movement is 1967
Process returned 0 (0x0)   execution time : 31.791 s
Press any key to continue.