VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS

Submitted by

SHASHANK M S (1BM21CS201)

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 **SHASHANK M S (1BM21CS201),** 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.

Sneha S Bagalkot
Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak
Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

Course Outcome

CO1	Apply the different concepts and functionalities of Operating System
CO2	Analyse various Operating system strategies and techniques
CO3	Demonstrate the different functionalities of Operating System.
CO4	Conduct practical experiments to implement the functionalities of Operating system.

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

FCFS

```
#include<stdio.h>

#include<conio.h>


void main()

{

int n,art[20],burst[20],wait[20],i,s=0,sum=0,tt[20],sum1=0;

  float avg,avg1;

    printf("\nEnter the number of processes:");

    scanf("%d",&n);

    printf("\nEnter the arrival time for %d processes\n",n);

    for(i=1;i<=n;i++)

            {

        printf("\nArrival time of %d process=",i);

        scanf("%d",&art[i]);

            }

     printf("\nEnter the Burst Time for %d processes\n",n);

    for(i=1;i<=n;i++)

            {

            printf("\nBurst Time of %d process=",i);

        scanf("%d",&burst[i]);
```

```
        }
    printf("\Gmatt Chart is\n");
    for(i=1;i<=n;i++)
        {
         tt[i]=s+burst[i]-art[i];
         wait[i]=tt[i]-burst[i];
        printf("\nProcess %d starts at %d and ends at %d",i,s,burst[i]+s);
        printf("\nTurn Around Time for %d process is:%d",i,tt[i]);
        printf("\nWaiting Time for %d process is:%d",i,wait[i]);
           s=s+burst[i];
         sum=sum+tt[i];
         sum1=sum1+wait[i];
        }

           avg=(float)sum/n;
    avg1=(float)sum1/n;
    printf("\nAverage Turn Around Time for FCFS CPU Scheduling is %f",avg);
     printf("\nAverage Waiting Time for FCFS CPU Scheduling is %f",avg1);
getch();
 }
```

SAMPLE OUTPUT

"C:\Users\Admin\Desktop\FCFS 1BM21CS205.exe"

```
Arrival time of 4 process=6

Enter the Burst Time for 4 processes

Burst Time of 1 process=3

Burst Time of 2 process=6

Burst Time of 3 process=4

Burst Time of 4 process=2
Gmatt Chart is

Process 1 starts at 0 and ends at 3
Turn Around Time for 1 process is:3
Waiting Time for 1 process is:0
Process 2 starts at 3 and ends at 9
Turn Around Time for 2 process is:8
Waiting Time for 2 process is:2
Process 3 starts at 9 and ends at 13
Turn Around Time for 3 process is:9
Waiting Time for 3 process is:5
Process 4 starts at 13 and ends at 15
Turn Around Time for 4 process is:9
Waiting Time for 4 process is:7
Average Turn Around Time for FCFS CPU Scheduling is 7.250000
Average Waiting Time for FCFS CPU Scheduling is 3.500000
Process returned 57 (0x39)   execution time : 20.641 s
Press any key to continue.
```

□ SJF (pre-emptive & Non-pre-emptive)

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

#define MAX_PROCESSES 10

struct Process {
    int pid;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int turnaround_time;
    int waiting_time;
};

void sjf_nonpreemptive(struct Process processes[], int n) {
    int i,j,count=0,m;
    for(i=0;i<n;i++)
    {
    if(processes[i].arrival_time==0)
    count++;
}
if(count==n||count==1)
{
if(count==n)
{
for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (processes[j].burst_time > processes[j + 1].burst_time) {
                struct Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
}
else
{
for (i = 1; i < n - 1; i++) {
```

```
        for (j = 1; j <= n - i - 1; j++) {
            if (processes[j].burst_time > processes[j + 1].burst_time) {
                struct Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
}


}

    int total_time = 0;
    double total_turnaround_time = 0;
    double total_waiting_time = 0;

    for (i = 0; i < n; i++) {
        total_time += processes[i].burst_time;
        processes[i].turnaround_time = total_time - processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;

        total_turnaround_time += processes[i].turnaround_time;
        total_waiting_time += processes[i].waiting_time;
    }

    printf("Process\tTurnaround Time\tWaiting Time\n");
    for (i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\n", processes[i].pid, processes[i].turnaround_time,
processes[i].waiting_time);
    }

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

void sjf_preemptive(struct Process processes[], int n) {
    int total_time = 0,i;
    int completed = 0;
```

```
    while (completed < n) {
        int shortest_burst = -1;
        int next_process = -1;

        for (i = 0; i < n; i++) {
            if (processes[i].arrival_time <= total_time && processes[i].remaining_time > 0) {
                if (shortest_burst == -1 || processes[i].remaining_time < shortest_burst) {
                    shortest_burst = processes[i].remaining_time;
                    next_process = i;
                }
            }
        }

        if (next_process == -1) {
            total_time++;
            continue;
        }

        processes[next_process].remaining_time--;
        total_time++;

        if (processes[next_process].remaining_time == 0) {
            completed++;
            processes[next_process].turnaround_time = total_time -
processes[next_process].arrival_time;
            processes[next_process].waiting_time = processes[next_process].turnaround_time -
processes[next_process].burst_time;
        }
    }

    double total_turnaround_time = 0;
    double total_waiting_time = 0;

    printf("Process\tTurnaround Time\tWaiting Time\n");
    for (i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\n", processes[i].pid, processes[i].turnaround_time,
processes[i].waiting_time);

        total_turnaround_time += processes[i].turnaround_time;
        total_waiting_time += processes[i].waiting_time;
```

```
    }

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

int main() {
    int n, quantum,i,choice;
    struct Process processes[MAX_PROCESSES];

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

    for (i = 0; i < n; i++) {
        printf("Process %d\n", i + 1);
        printf("Enter arrival time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Enter burst time: ");
        scanf("%d", &processes[i].burst_time);
        processes[i].pid = i + 1;
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].turnaround_time = 0;
        processes[i].waiting_time = 0;
    }
    while(1)
    {
         printf("\nSelect a scheduling algorithm:\n");
    printf("1. SJF Non-preemptive\n");
    printf("2. SRTF\n");
    printf("3. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            printf("\nSJF Non-preemptive Scheduling:\n");
            sjf_nonpreemptive(processes, n);
            break;
        case 2:
            printf("\nSJF Preemptive Scheduling:\n");
```

```
            sjf_preemptive(processes, n);
            break;
        case 3:exit(0);
            break;
        default:
            printf("Invalid choice!\n");
            return 1;
    }
    }

    return 0;
}
```

SAMPLE OUTPUT

C:\Users\STUDENT\Desktop\scheduling.exe

```
Enter the number of processes: 6
Process 1
Enter arrival time: 0
Enter burst time: 8
Process 2
Enter arrival time: 0
Enter burst time: 9
Process 3
Enter arrival time: 3
Enter burst time: 3
Process 4
Enter arrival time: 5
Enter burst time: 4
Process 5
Enter arrival time: 7
Enter burst time: 10
Process 6
Enter arrival time: 3
Enter burst time: 12

Select a scheduling algorithm:
1. SJF Non-preemptive
2. SRTF
3. Exit
Enter your choice: 1
```

C:\Users\STUDENT\Desktop\scheduling.exe

```
SJF Non-preemptive Scheduling:
Process Turnaround Time Waiting Time
1       8               0
2       17              8
3       17              14
4       19              15
5       27              17
6       43              31
Average Turnaround Time: 21.83
Average Waiting Time: 14.17

Select a scheduling algorithm:
1. SJF Non-preemptive
2. SRTF
3. Exit
Enter your choice: 2

SJF Preemptive Scheduling:
Process Turnaround Time Waiting Time
1       15              7
2       24              15
3       3               0
4       5               1
5       27              17
6       43              31
Average Turnaround Time: 19.50
Average Waiting Time: 11.83
```

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

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

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

#define MAX_PROCESSES 10

struct Process {
    int pid;
    int arrival_time;
    int burst_time;
    int priority;
    int remaining_time;
    int turnaround_time;
    int waiting_time;
};

void priority_nonpreemptive(struct Process processes[], int n) {
    int i,j,count=0,m;
    for(i=0;i<n;i++)
    {
    if(processes[i].arrival_time==0)
    count++;
}
if(count==n||count==1)
{
if(count==n)
{
for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (processes[j].priority > processes[j + 1].priority) {
                struct Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
}
```

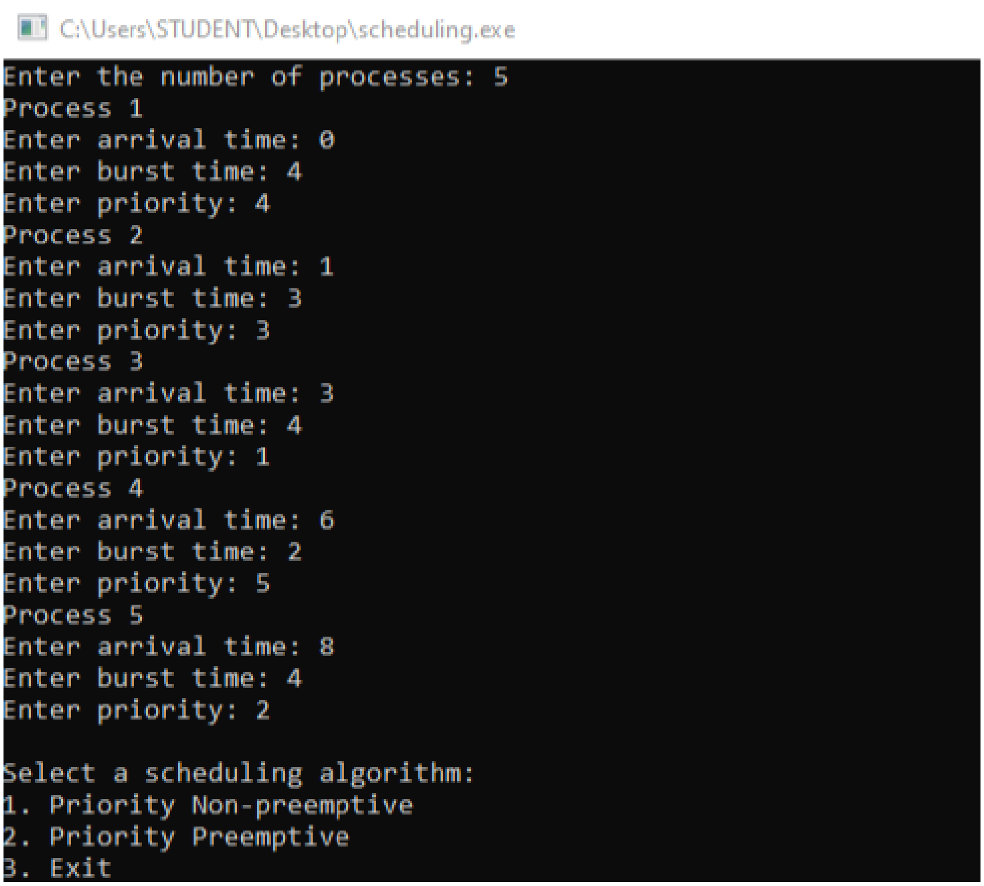
```
else
{
    for (i = 1; i < n - 1; i++) {
        for (j = 1; j <= n - i - 1; j++) {
            if (processes[j].priority > processes[j + 1].priority) {
                struct Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
}
}

    int total_time = 0;
    double total_turnaround_time = 0;
    double total_waiting_time = 0;

    for (i = 0; i < n; i++) {
        total_time += processes[i].burst_time;
        processes[i].turnaround_time = total_time - processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;

        total_turnaround_time += processes[i].turnaround_time;
        total_waiting_time += processes[i].waiting_time;
    }

    printf("Process\tTurnaround Time\tWaiting Time\n");
    for (i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\n", processes[i].pid, processes[i].turnaround_time,
processes[i].waiting_time);
    }

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

void priority_preemptive(struct Process processes[], int n) {
    int total_time = 0,i;
    int completed = 0;
```

```
    while (completed < n) {
        int highest_priority = -1;
        int next_process = -1;

        for (i = 0; i < n; i++) {
            if (processes[i].arrival_time <= total_time && processes[i].remaining_time > 0) {
                if (highest_priority == -1 || processes[i].priority < highest_priority) {
                    highest_priority = processes[i].priority;
                    next_process = i;
                }
            }
        }

        if (next_process == -1) {
            total_time++;
            continue;
        }

        processes[next_process].remaining_time--;
        total_time++;

        if (processes[next_process].remaining_time == 0) {
            completed++;
            processes[next_process].turnaround_time = total_time -
processes[next_process].arrival_time;
            processes[next_process].waiting_time = processes[next_process].turnaround_time -
processes[next_process].burst_time;
        }
    }

    double total_turnaround_time = 0;
    double total_waiting_time = 0;

    printf("Process\tTurnaround Time\tWaiting Time\n");
    for (i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\n", processes[i].pid, processes[i].turnaround_time,
processes[i].waiting_time);

        total_turnaround_time += processes[i].turnaround_time;
```

```
        total_waiting_time += processes[i].waiting_time;
    }

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

int main() {
    int n, quantum,i,choice;
    struct Process processes[MAX_PROCESSES];

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

    for (i = 0; i < n; i++) {
        printf("Process %d\n", i + 1);
        printf("Enter arrival time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Enter burst time: ");
        scanf("%d", &processes[i].burst_time);
        printf("Enter priority: ");
        scanf("%d", &processes[i].priority);
        processes[i].pid = i + 1;
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].turnaround_time = 0;
        processes[i].waiting_time = 0;
    }
    while(1)
    {
         printf("\nSelect a scheduling algorithm:\n");

    printf("1. Priority Non-preemptive\n");
    printf("2. Priority Preemptive\n");
    printf("3. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            printf("\nPriority Non-preemptive Scheduling:\n");
```

```
            priority_nonpreemptive(processes, n);
            break;
        case 2:
            printf("\nPriority Preemptive Scheduling:\n");
            priority_preemptive(processes, n);
            break;
        case 3:exit(0);
            break;
        default:
            printf("Invalid choice!\n");
            return 1;
    }
    }

    return 0;
}
```

SAMPLE OUTPUT



```
C:\Users\STUDENT\Desktop\scheduling.exe
Enter the number of processes: 5
Process 1
Enter arrival time: 0
Enter burst time: 4
Enter priority: 4
Process 2
Enter arrival time: 1
Enter burst time: 3
Enter priority: 3
Process 3
Enter arrival time: 3
Enter burst time: 4
Enter priority: 1
Process 4
Enter arrival time: 6
Enter burst time: 2
Enter priority: 5
Process 5
Enter arrival time: 8
Enter burst time: 4
Enter priority: 2

Select a scheduling algorithm:
1. Priority Non-preemptive
2. Priority Preemptive
3. Exit
```

C:\Users\STUDENT\Desktop\scheduling.exe

```
2. Priority Preemptive
3. Exit
Enter your choice: 1

Priority Non-preemptive Scheduling:
Process Turnaround Time Waiting Time
1       4               0
3       5               1
5       4               0
2       14              11
4       11              9
Average Turnaround Time: 7.60
Average Waiting Time: 4.20

Select a scheduling algorithm:
1. Priority Non-preemptive
2. Priority Preemptive
3. Exit
Enter your choice: 2

Priority Preemptive Scheduling:
Process Turnaround Time Waiting Time
1       15              11
3       4               0
5       4               0
2       7               4
4       11              9
Average Turnaround Time: 8.20
Average Waiting Time: 4.80
```

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

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

struct P{
int AT,BT,ST[20],WT,FT,TAT,pos;
};

int quant;
int main(){
int n,i,j;
printf("Enter the no. of processes :");
scanf("%d",&n);
struct P p[n];

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

printf("Enter the process numbers \n");
for(i=0;i<n;i++)
scanf("%d",&(p[i].pos));

printf("Enter the Arrival time of processes \n");
for(i=0;i<n;i++)
scanf("%d",&(p[i].AT));
```

```
printf("Enter the Burst time of processes \n");
for(i=0;i<n;i++)
scanf("%d",&(p[i].BT));




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



int index=-1;
for(i=0;i<n;i++){
        b[i]=p[i].BT;
        a[i]=p[i].AT;
        for(j=0;j<20;j++){
        s[i][j]=-1;
        }
}

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

while(c!=0){

mini=INT_MAX;
```

```
flag=false;

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

        }
}

if(!flag){
        time++;
        continue;
}

j=0;

while(s[index][j]!=-1){
j++;
}

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

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

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

if(b[index]==0){
c--;
p[index].FT=time;
p[index].WT=p[index].FT-p[index].AT-p[index].BT;
tot_wt+=p[index].WT;
p[index].TAT=p[index].BT+p[index].WT;
tot_tat+=p[index].TAT;

}


}
printf("Process number ");
printf("Arrival time ");
printf("Burst time ");
printf("\tStart time");
```

```
j=0;
while(j!=10){
j+=1;
printf(" ");
}
printf("\t\tFinal time");
printf("\tWait Time ");
printf("\tTurnAround Time \n");


for(i=0;i<n;i++){
printf("%d \t\t",p[i].pos);
printf("%d \t\t",p[i].AT);
printf("%d \t",p[i].BT);
j=0;
int v=0;
while(s[i][j]!=-1){
printf("%d ",p[i].ST[j]);
j++;
v+=3;
}
while(v!=40){
printf(" ");
v+=1;
}
printf("%d \t\t",p[i].FT);
printf("%d \t\t",p[i].WT);
printf("%d \n",p[i].TAT);
```

}

double avg_wt,avg_tat;

avg_wt=tot_wt/(float)n;

avg_tat=tot_tat/(float)n;

printf("The average wait time is : %lf\n",avg_wt);

printf("The average TurnAround time is : %lf\n",avg_tat);

return 0;

}

SAMPLE OUTPUT

```
Enter the no. of processes :5
Enter the quantum
2
Enter the process numbers
1
2
3
4
5
Enter the Arrival time of processes
0
1
2
3
4
Enter the Burst time of processes
5
3
1
2
3
Process number Arrival time Burst time  Start time          Final time     Wait Time      TurnAround Time

1             0             5      0 5 12                    13            8              13
2             1             3      2 11                      12            8              11
3             2             1      4                         5             2              3
4             3             2      7                         9             4              6
5             4             3      9 13                      14            7              10
The average wait time is : 5.800000
The average TurnAround time is : 8.600000
```

3. Write a C program to simulate a 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.

Process	Arrival Time	Burst Time	System(0)/User(1)
P1	0	3	0
P2	2	2	0
P3	4	4	1
P4	4	2	1
P5	8	2	0
P6	10	3	1

```
#include <stdio.h>
#include<stdlib.h>
#include <stdbool.h>
#define MAX_QUEUE_SIZE 100
int totalTime=0;
int userProcess=0,systemProcess=0;

// Structure to represent a process
typedef struct {
    int processID;
    int arrivalTime;
    int burstTime;
    int remainingTime;
    int priority; // 0 for system process, 1 for user process
} Process;

// Function to execute a process
```

```
void executeProcess(Process process) {
        int i;
    printf("Executing Process %d\n", process.processID);
    // Simulating the execution time of the process
    for (i = 1; i <= process.burstTime; i++) {
        printf("Process %d: %d/%d\n", process.processID, i, process.burstTime);
    }
    printf("Process %d executed\n", process.processID);
}

// Function to perform FCFS scheduling for a queue of processes
void scheduleFCFS(Process system[],Process user[]) {
        int i,j;
    for(i=0;i<systemProcess;i++)
    {
        for(j=i+1;j<systemProcess;j++)
        {
            if(system[i].arrivalTime>system[j].arrivalTime)
            {
                Process temp=system[i];
                system[i]=system[j];
                system[j]=temp;
                }
            }
        }
        for(i=0;i<userProcess;i++)
    {
        for(j=i+1;j<userProcess;j++)
        {
            if(user[i].arrivalTime>user[j].arrivalTime)
            {
                Process temp=user[i];
                user[i]=user[j];
                user[j]=temp;
                }
            }
        }
        int completed=0;
        int currentProcess=-1;
        bool isUserProcess=false;
        int size=userProcess+systemProcess;
            while(1)
         {
```

```
			int count=0;
			for(i=0;i<systemProcess;i++)
			{
				if(system[i].remainingTime<=0)
				{
					count++;
				}
			}
			for(j=0;j<userProcess;j++)
		{
			  if(user[j].remainingTime<=0)
			{
					count++;
				}
			}
			if(count==size)
			{
				printf("\n end of processess");
				exit(0);
			}
			for(i=0;i<systemProcess;i++)
			{
				if(totalTime>=system[i].arrivalTime &&
system[i].remainingTime>0)
				{
					currentProcess=i;
					isUserProcess=false;
					break;
				}
			}
			if(currentProcess==-1)
			{
			  for(j=0;j<userProcess;j++)
			  {
				if(totalTime>=user[j].arrivalTime && user[j].remainingTime>0)
				{
					currentProcess=j;
					isUserProcess=true;
					break;
				}
			  }
			}
			if(currentProcess==-1)
```

```
            {
                totalTime++;
                printf("\n %d idle time...",totalTime);
                if(totalTime==1000)
                {
                    exit(0);
                }
                continue;
            }
                if(isUserProcess==true)
                {
                    user[currentProcess].remainingTime--;
                    printf("\n User process %d will excecute at %d
",user[currentProcess].processID,(totalTime));
                    totalTime++;
                    isUserProcess=false;
                    currentProcess=-1;
                    if(user[currentProcess].remainingTime==0)
                {
                        completed++;
                }
                }else{
                    int temp=totalTime;
                  while(system[currentProcess].remainingTime--){
                    totalTime++;
                    }
                    if(system[currentProcess].remainingTime==0)
                {
                        completed++;
                }
                    printf("\n System process %d will excecute from %d to %d
",system[currentProcess].processID,temp,(totalTime));
                    isUserProcess=false;
                    currentProcess=-1;
                }
        }
}

int main() {
  int numProcesses,i;
  Process processes[MAX_QUEUE_SIZE];

  // Reading the number of processes
```

```
    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);
    // Reading process details
    for (i = 0; i < numProcesses; i++) {
        printf("Process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &processes[i].arrivalTime);
        printf("Burst Time: ");
        scanf("%d", &processes[i].burstTime);
        printf("System(0)/User(1): ");
        scanf("%d", &processes[i].priority);
        processes[i].processID = i + 1;
        processes[i].remainingTime=processes[i].burstTime;
        if(processes[i].priority==1)
        {
            userProcess++;
                    }else{
                            systemProcess++;
                    }
    }

    Process systemQueue[MAX_QUEUE_SIZE];
    int systemQueueSize = 0;
    Process userQueue[MAX_QUEUE_SIZE];
    int userQueueSize = 0;
    for (i = 0; i < numProcesses; i++) {
        if (processes[i].priority == 0) {
            systemQueue[systemQueueSize++] = processes[i];
        } else {
            userQueue[userQueueSize++] = processes[i];
        }
    }
    printf("Order of Excecution :\n");
    scheduleFCFS(systemQueue,userQueue);
    return 0;
}
```

SAMPLE OUTPUT

```
Enter the number of processes: 6
Process 1:
Arrival Time: 0
Burst Time: 3
System(0)/User(1): 0
Process 2:
Arrival Time: 2
Burst Time: 2
System(0)/User(1): 0
Process 3:
Arrival Time: 4
Burst Time: 4
System(0)/User(1): 1
Process 4:
Arrival Time: 4
Burst Time: 2
System(0)/User(1): 1
Process 5:
Arrival Time: 8
Burst Time: 2
System(0)/User(1): 0
Process 6:
Arrival Time: 10
Burst Time: 3
System(0)/User(1): 1
Order of Excecution :

 System process 1 will excecute from 0 to 3
 System process 2 will excecute from 3 to 5
 User process 3 will excecute at 5
 User process 3 will excecute at 6
 User process 3 will excecute at 7
 System process 5 will excecute from 8 to 10
 User process 3 will excecute at 10
 User process 4 will excecute at 11
 User process 4 will excecute at 12
 User process 6 will excecute at 13
 User process 6 will excecute at 14
 User process 6 will excecute at 15
 end of processess
```

4.a.Simulate Rate Monotonic Scheduling for the following and show the order of execution of processes in CPU timeline:

Process	Execution Time	Period
P1	3	20
P2	2	5
P3	2	10

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

#define MAX_PROCESS 10

int num_of_process = 3, count, remain, time_quantum;
int execution_time[MAX_PROCESS], period[MAX_PROCESS],
remain_time[MAX_PROCESS], deadline[MAX_PROCESS],
remain_deadline[MAX_PROCESS];
int burst_time[MAX_PROCESS], wait_time[MAX_PROCESS],
completion_time[MAX_PROCESS], arrival_time[MAX_PROCESS];

// collecting details of processes
void get_process_info(int selected_algo)
{
    printf("Enter total number of processes (maximum %d): ", MAX_PROCESS);
    scanf("%d", &num_of_process);
    if (num_of_process < 1)
    {
        printf("Do you really want to schedule %d processes? -_-", num_of_process);
        exit(0);
    }
    if (selected_algo == 2)
    {
        printf("\nEnter Time Quantum: ");
        scanf("%d", &time_quantum);
        if (time_quantum < 1)
        {
            printf("Invalid Input: Time quantum should be greater than 0\n");
            exit(0);
        }
    }
```

```
    for (int i = 0; i < num_of_process; i++)
    {
        printf("\nProcess %d:\n", i + 1);
        if (selected_algo == 1)
        {
            printf("==> Burst time: ");
            scanf("%d", &burst_time[i]);
        }
        else if (selected_algo == 2)
        {
            printf("=> Arrival Time: ");
            scanf("%d", &arrival_time[i]);
            printf("=> Burst Time: ");
            scanf("%d", &burst_time[i]);
            remain_time[i] = burst_time[i];
        }
        else if (selected_algo > 2)
        {
            printf("==> Execution time: ");
            scanf("%d", &execution_time[i]);
            remain_time[i] = execution_time[i];
            if (selected_algo == 4)
            {
                printf("==> Deadline: ");
                scanf("%d", &deadline[i]);
            }
            else
            {
                printf("==> Period: ");
                scanf("%d", &period[i]);
            }
        }
    }
}

// get maximum of three numbers
int max(int a, int b, int c)
{
    int max;
    if (a >= b && a >= c)
        max = a;
    else if (b >= a && b >= c)
        max = b;
```

```
    else if (c >= a && c >= b)
        max = c;
    return max;
}

// calculating the observation time for scheduling timeline
int get_observation_time(int selected_algo)
{
    if (selected_algo < 3)
    {
        int sum = 0;
        for (int i = 0; i < num_of_process; i++)
        {
            sum += burst_time[i];
        }
        return sum;
    }
    else if (selected_algo == 3)
    {
        return max(period[0], period[1], period[2]);
    }
    else if (selected_algo == 4)
    {
        return max(deadline[0], deadline[1], deadline[2]);
    }
}

// print scheduling sequence
void print_schedule(int process_list[], int cycles)
{
    printf("\nScheduling:\n\n");
    printf("Time: ");
    for (int i = 0; i < cycles; i++)
    {
        if (i < 10)
            printf("| 0%d ", i);
        else
            printf("| %d ", i);
    }
    printf("|\n");

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

```
            printf("P[%d]: ", i + 1);
            for (int j = 0; j < cycles; j++)
            {
                if (process_list[j] == i + 1)
                    printf("|####");
                else
                    printf("|    ");
            }
            printf("|\n");
        }
    }

    void rate_monotonic(int time)
    {
        int process_list[100] = {0}, min = 999, next_process = 0;
        float utilization = 0;
        for (int i = 0; i < num_of_process; i++)
        {
            utilization += (1.0 * execution_time[i]) / period[i];
        }
        int n = num_of_process;
        if (utilization > n * (pow(2, 1.0 / n) - 1))
        {
            printf("\nGiven problem is not schedulable under the said scheduling algorithm.\n");
            exit(0);
        }


        for (int i = 0; i < time; i++)
        {
            min = 1000;
            for (int j = 0; j < num_of_process; j++)
            {
                if (remain_time[j] > 0)
                {
                    if (min > period[j])
                    {
                        min = period[j];
                        next_process = j;
                    }
                }
            }
```

```
        if (remain_time[next_process] > 0)
        {
            process_list[i] = next_process + 1; // +1 for catering 0 array index.
            remain_time[next_process] -= 1;
        }

        for (int k = 0; k < num_of_process; k++)
        {
            if ((i + 1) % period[k] == 0)
            {
                remain_time[k] = execution_time[k];
                next_process = k;
            }
        }
    }
    print_schedule(process_list, time);
}



int main(int argc, char *argv[])
{
    int option = 0;

    printf("3. Rate Monotonic Scheduling\n");

    printf("Select > ");
    scanf("%d", &option);
    printf("---------------------------------------------\n");

    get_process_info(option); // collecting processes detail
    int observation_time = get_observation_time(option);


     if (option == 3)
        rate_monotonic(observation_time);
    return 0;
}
```

SAMPLE OUTPUT

```
"C:\Users\Admin\Desktop\4th Sem\Lab\OS LAB\Rate Monotonic-1.exe"
3. Rate Monotonic Scheduling
Select > 3
-----------------------------
Enter total number of processes (maximum 10): 3

Process 1:
==> Execution time: 3
==> Period: 20

Process 2:
==> Execution time: 2
==> Period: 5

Process 3:
==> Execution time: 2
==> Period: 10

Scheduling:

Time: | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
P[1]: |    |    |    |    |####|    |    |####|####|    |    |    |    |    |    |    |    |    |    |    |
P[2]: |####|####|    |    |    |####|####|    |    |    |####|####|    |    |    |####|####|    |    |    |
P[3]: |    |    |####|####|    |    |    |    |    |    |    |    |####|####|    |    |    |    |    |    |

Process returned 0 (0x0)   execution time : 68.961 s
Press any key to continue.
```

4.b.Simulate Earliest Deadline First for the following and show the order of execution of processes in CPU timeline:

Process	Execution Time	Deadline	Period

P1	3	7	20
P2	2	4	5
P3	2	8	10

```
#include <stdio.h>
#define arrival			0
#define execution		1
#define deadline		2
#define period			3
#define abs_arrival		4
#define execution_copy	5
#define abs_deadline		6

typedef struct
{
	int T[7],instance,alive;

}task;

#define IDLE_TASK_ID 1023
#define ALL 1
#define CURRENT 0

void get_tasks(task *t1,int n);
int hyperperiod_calc(task *t1,int n);
float cpu_util(task *t1,int n);
```

```
int gcd(int a, int b);
int lcm(int *a, int n);
int sp_interrupt(task *t1,int tmr,int n);
int min(task *t1,int n,int p);
void update_abs_arrival(task *t1,int n,int k,int all);
void update_abs_deadline(task *t1,int n,int all);
void copy_execution_time(task *t1,int n,int all);


int timer = 0;

int main()
{
	task *t;
	int n, hyper_period, active_task_id;
	float cpu_utilization;
	printf("Enter number of tasks\n");
	scanf("%d", &n);
	t = malloc(n * sizeof(task));
	get_tasks(t, n);
	cpu_utilization = cpu_util(t, n);
	printf("CPU Utilization %f\n", cpu_utilization);

	if (cpu_utilization < 1)
		printf("Tasks can be scheduled\n");
```

```
else
        printf("Schedule is not feasible\n");

hyper_period = hyperperiod_calc(t, n);
copy_execution_time(t, n, ALL);
update_abs_arrival(t, n, 0, ALL);
update_abs_deadline(t, n, ALL);

while (timer <= hyper_period)
{

        if (sp_interrupt(t, timer, n))
        {
                active_task_id = min(t, n, abs_deadline);
        }

        if (active_task_id == IDLE_TASK_ID)
        {
                printf("%d  Idle\n", timer);
        }

        if (active_task_id != IDLE_TASK_ID)
        {

                if (t[active_task_id].T[execution_copy] != 0)
```

```
                {
                        t[active_task_id].T[execution_copy]--;
                        printf("%d  Task %d\n", timer, active_task_id + 1);
                }

                if (t[active_task_id].T[execution_copy] == 0)
                {
                        t[active_task_id].instance++;
                        t[active_task_id].alive = 0;
                        copy_execution_time(t, active_task_id, CURRENT);
                        update_abs_arrival(t, active_task_id,
t[active_task_id].instance, CURRENT);
                        update_abs_deadline(t, active_task_id, CURRENT);
                        active_task_id = min(t, n, abs_deadline);
                }
        }
        ++timer;
    }
    free(t);
    return 0;
}
void get_tasks(task *t1, int n)
{
    int i = 0;
    while (i < n)
    {
```

```
        printf("Enter Task %d parameters\n", i + 1);
        printf("Arrival time: ");
        scanf("%d", &t1->T[arrival]);
        printf("Execution time: ");
        scanf("%d", &t1->T[execution]);
        printf("Deadline time: ");
        scanf("%d", &t1->T[deadline]);
        printf("Period: ");
        scanf("%d", &t1->T[period]);
        t1->T[abs_arrival] = 0;
        t1->T[execution_copy] = 0;
        t1->T[abs_deadline] = 0;
        t1->instance = 0;
        t1->alive = 0;
        t1++;
        i++;
    }
}

int hyperperiod_calc(task *t1, int n)
{
    int i = 0, ht, a[10];
    while (i < n)

    {
```

```
        a[i] = t1->T[period];
        t1++;
        i++;
    }
    ht = lcm(a, n);

    return ht;
}

int gcd(int a, int b)
{
    if (b == 0)
        return a;
    else
        return gcd(b, a % b);
}

int lcm(int *a, int n)
{
    int res = 1, i;
    for (i = 0; i < n; i++)
    {
        res = res * a[i] / gcd(res, a[i]);
    }
    return res;
```

```
}

int sp_interrupt(task *t1, int tmr, int n)
{
    int i = 0, n1 = 0, a = 0;
    task *t1_copy;
    t1_copy = t1;
    while (i < n)
    {
        if (tmr == t1->T[abs_arrival])
        {
            t1->alive = 1;
            a++;
        }
        t1++;
        i++;
    }

    t1 = t1_copy;
    i = 0;

    while (i < n)
    {
        if (t1->alive == 0)
            n1++;
```

```
        t1++;
        i++;
    }

    if (n1 == n || a != 0)
    {
        return 1;
    }

    return 0;
}

void update_abs_deadline(task *t1, int n, int all)
{
    int i = 0;
    if (all)
    {
        while (i < n)
        {
            t1->T[abs_deadline] = t1->T[deadline] + t1->T[abs_arrival];
            t1++;
            i++;
        }
    }
    else
```

```
        {
                t1 += n;
                t1->T[abs_deadline] = t1->T[deadline] + t1->T[abs_arrival];
        }
}

void update_abs_arrival(task *t1, int n, int k, int all)
{
        int i = 0;
        if (all)
        {
                while (i < n)
                {
                        t1->T[abs_arrival] = t1->T[arrival] + k * (t1->T[period]);
                        t1++;
                        i++;
                }
        }
        else
        {
                t1 += n;
                t1->T[abs_arrival] = t1->T[arrival] + k * (t1->T[period]);
        }
}
```

```
void copy_execution_time(task *t1, int n, int all)
{
	int i = 0;
	if (all)
	{
		while (i < n)
		{
			t1->T[execution_copy] = t1->T[execution];
			t1++;
			i++;
		}
	}
	else
	{
		t1 += n;
		t1->T[execution_copy] = t1->T[execution];
	}
}

int min(task *t1, int n, int p)
{
	int i = 0, min = 0x7FFF, task_id = IDLE_TASK_ID;
	while (i < n)
	{
		if (min > t1->T[p] && t1->alive == 1)
```

```
        {
            min = t1->T[p];
            task_id = i;
        }
        t1++;
        i++;
    }
    return task_id;
}

float cpu_util(task *t1, int n)
{
    int i = 0;
    float cu = 0;
    while (i < n)
    {
        cu = cu + (float)t1->T[execution] / (float)t1->T[deadline];
        t1++;
        i++;
    }
    return cu;
}
```

SAMPLE OUTPUT

"C:\Users\Admin\Desktop\4th Sem\Lab\OS LAB\EDF-1.exe"

```
Enter number of tasks
3
Enter Task 1 parameters
Arrival time: 0
Execution time: 3
Deadline time: 7
Period: 20
Enter Task 2 parameters
Arrival time: 0
Execution time: 2
Deadline time: 4
Period: 5
Enter Task 3 parameters
Arrival time: 0
Execution time: 2
Deadline time: 8
Period: 10
CPU Utilization 1.178571
Schedule is not feasible
0  Task 2
1  Task 2
2  Task 1
3  Task 1
4  Task 1
5  Task 3
6  Task 3
7  Task 2
8  Task 2
9  Idle
10  Task 2
11  Task 2
12  Task 3
13  Task 3
14  Idle
15  Task 2
16  Task 2
17  Idle
18  Idle
19  Idle
20  Task 2

Process returned 0 (0x0)   execution time : 24.796 s
Press any key to continue.
```

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

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

```
#include<conio.h>
int mutex=1;
int full=0;
int empty=10;
int cnt=0;
int wait(int s)
{
while(s<=0);
s--;
return s;
}
int signal(int s)
{
s++;
return s;
}
void producer()
{
empty=wait(empty);
mutex=wait(mutex);
cnt++;
printf("Producer produces an item %d\n",cnt);
mutex=signal(mutex);
full=signal(full);
}
void consumer()
{
```

```
full=wait(full);
mutex=wait(mutex);
printf("Consumer consumes an item %d\n",cnt);
cnt--;
    mutex=signal(mutex);
empty=signal(empty);
}
void main()
{
int choice;
printf("1.Produce\n2.Consume\n3.Exit\n");
while(1)
{
printf("Enter your choice:\n");
scanf("%d",&choice);
switch(choice)
{
case 1:if(empty==0)
{
printf("Buffer is full\n");
}
else{
producer();
}
break;
case 2:if(full==0)
{
printf("Buffer is empty\n");
```

```
}
else{
consumer();
}
break;
case 3:exit(0);
     break;
default:printf("Invalid choice\n");
}
}
getch();

}
```

SAMPLE OUTPUT

```
"C:\Users\STUDENT\Desktop\P C.exe"
1.Produce
2.Consume
3.Exit
Enter your choice:
1
Producer produces an item 1
Enter your choice:
1
Producer produces an item 2
Enter your choice:
2
Consumer consumes an item 2
Enter your choice:
1
Producer produces an item 2
Enter your choice:
2
Consumer consumes an item 2
Enter your choice:
2
Consumer consumes an item 1
Enter your choice:
2
Buffer is empty
Enter your choice:
1
Producer produces an item 1
Enter your choice:
1
Producer produces an item 2
Enter your choice:
1
Producer produces an item 3
Enter your choice:
1
Producer produces an item 4
Enter your choice:
1
Producer produces an item 5
Enter your choice:
1
Producer produces an item 6
Enter your choice:
1
Producer produces an item 7
Enter your choice:
1
Producer produces an item 8
Enter your choice:
1
Producer produces an item 9
Enter your choice:
1
Producer produces an item 10
Enter your choice:
1
Buffer is full
Enter your choice:
```

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

problem.

```
#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]) has no effect
        // during takefork
        // used to wake up hungry philosophers
        // during putfork
        sem_post(&S[phnum]);
    }
}

// take up chopsticks
void take_fork(int phnum)
{

    sem_wait(&mutex);

    // state that hungry
    state[phnum] = HUNGRY;

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

    // eat if neighbours are not eating
    test(phnum);

    sem_post(&mutex);
```

```
    // if unable to eat wait to be signalled
    sem_wait(&S[phnum]);

    sleep(1);
}

// put down chopsticks
void put_fork(int phnum)
{

    sem_wait(&mutex);

    // state that thinking
    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];

    // initialize the semaphores
    sem_init(&mutex, 0, 1);

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

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

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

        // create philosopher processes
        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);
}
```

SAMPLE OUTPUT

```
"C:\Users\STUDENT\Desktop\Dining Philosopher.exe"
Philosopher 2 is thinking
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 3 is Hungry
Philosopher 5 is Hungry
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 2 is Hungry
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 4 is Hungry
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 1 is Hungry
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 3 is Hungry
Philosopher 5 is Hungry
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 2 is Hungry
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 4 is Hungry
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 is Hungry
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 3 is Hungry
Philosopher 5 is Hungry
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 2 is Hungry
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
```

7.BANKERS ALGORITHM

Use bankers algorithm to check if the following state is safe/unsafe:

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Is the system in a safe state? If Yes, then what is the safe sequence? What will happen if process P1 requests one additional instance of resource type A and two instances of resource type C?

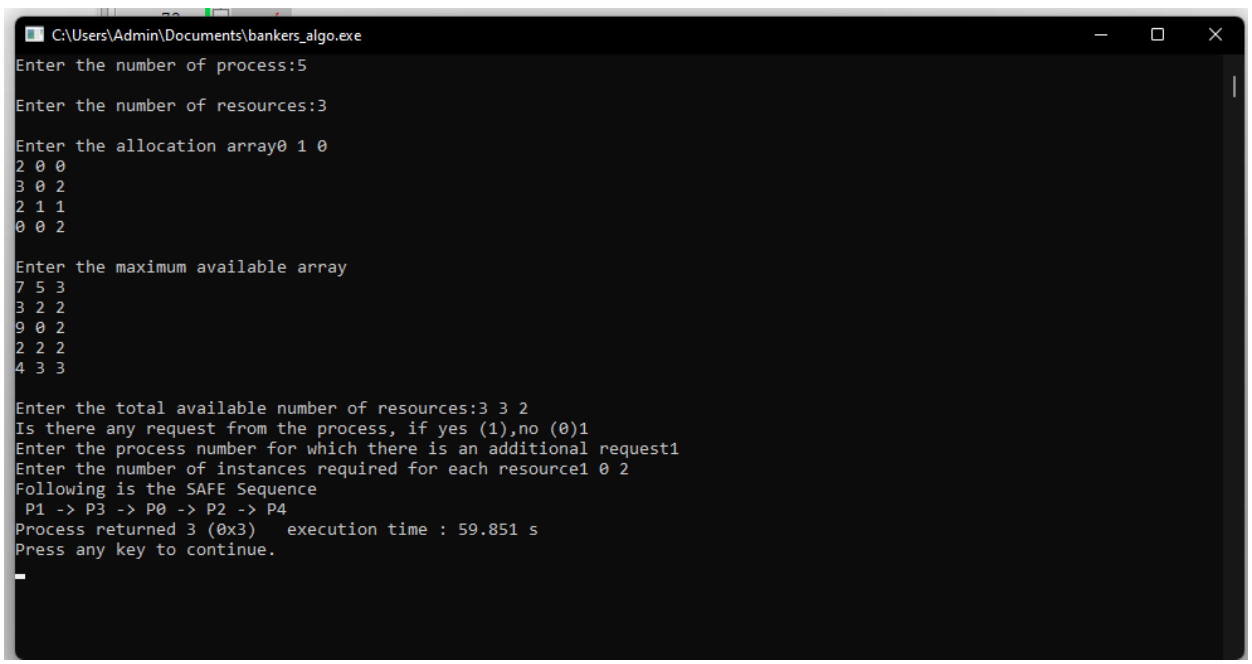
```
#include <stdio.h>
int n, m, i, j, k,alloc[10][10],max[10][10],avail[10],ch,t,add[10];
void main()
{
    printf("Enter the number of process:");
    scanf("%d",&n);
    printf("\nEnter the number of resources:");
    scanf("%d",&m);
    printf("\nEnter the allocation array");
    for(i=0;i<n;i++)
    {
        for(j=0;j<m;j++)
        {
            scanf("%d",&alloc[i][j]);
        }
    }
    printf("\nEnter the maximum available array");
    for(i=0;i<n;i++)
    {
        for(j=0;j<m;j++)
        {
            scanf("%d",&max[i][j]);
```

```
        }
    }
    printf("\nEnter the total available number of resources:");
    for(i=0;i<m;i++)
    {
        scanf("%d",&avail[i]);
    }
    printf("Is there any request from the process, if yes (1),no (0)");
    scanf("%d",&ch);
    if(ch==1)
    {
        printf("Enter the process number for which there is an additional request");
        scanf("%d",&t);
        printf("Enter the number of instances required for each resource");
        for(i=0;i<m;i++)
        {
            scanf("%d",&add[i]);
        }
        for(i=0;i<m;i++)
        {
            alloc[t][i]+=add[i];
        }
        if(max[t][0]<alloc[t][0]||max[t][1]<alloc[t][1]||max[t][2]<alloc[t][2])
            printf("It is not a valid request");
        else
        {

            for(i=0;i<m;i++)
        {
            avail[i]-=add[i];
        }
        bankers();
        }
    }
    else
        bankers();
}

void bankers()
{
```

```
int f[n], ans[n], ind = 0;
for (k = 0; k < n; k++)
{
    f[k] = 0;
}
int need[n][m];
for (i = 0; i < n; i++)
{
    for (j = 0; j < m; j++)
        need[i][j] = max[i][j] - alloc[i][j];
}
int y = 0;
for (k = 0; k < 5; k++)
{
    for (i = 0; i < n; i++)
    {
        if (f[i] == 0)
        {
            int flag = 0;
            for (j = 0; j < m; j++)
            {
                if (need[i][j] > avail[j])
                {
                    flag = 1;
                    break;
                }
            }
            if (flag == 0)
            {
                ans[ind++] = i;
                for (y = 0; y < m; y++)
                    avail[y] += alloc[i][y];
                f[i] = 1;
                break;
            }
        }
    }
}
```

```
    int flag = 1;
    for (int i = 0; i < n; i++)
    {
        if (f[i] == 0)
        {
            flag = 0;
            printf("The following system is not safe");
            break;
        }
    }
    if (flag == 1)
    {
        printf("Following is the SAFE Sequence\n");
        for (i = 0; i < n - 1; i++)
            printf(" P%d ->", ans[i]);
        printf(" P%d", ans[n - 1]);
    }
}
```

SAMPLE OUTPUT



```
C:\Users\Admin\Documents\bankers_algo.exe
Enter the number of process:5

Enter the number of resources:3

Enter the allocation array0 1 0
2 0 0
3 0 2
2 1 1
0 0 2

Enter the maximum available array
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3

Enter the total available number of resources:3 3 2
Is there any request from the process, if yes (1),no (0)1
Enter the process number for which there is an additional request1
Enter the number of instances required for each resource1 0 2
Following is the SAFE Sequence
 P1 -> P3 -> P0 -> P2 -> P4
Process returned 3 (0x3)   execution time : 59.851 s
Press any key to continue.
```

8. Write a C program to simulate deadlock detection.

```
#include<stdio.h>
static int mark[20];
int i, j, np, nr,k;

int main()
{
int alloc[10][10],request[10][10],avail[10],r[10],w[10];


printf ("\nEnter the no of the process: ");
scanf("%d",&np);
printf ("\nEnter the no of resources: ");
scanf("%d",&nr);
for(i=0;i<nr; i++)
{
printf("\nTotal Amount of the Resource R % d: ",i+1);
scanf("%d", &r[i]);
}
printf("\nEnter the request matrix:");


for(i=0;i<np;i++)
for(j=0;j<nr;j++)
scanf("%d",&request[i][j]);


printf("\nEnter the allocation matrix:");
for(i=0;i<np;i++)
for(j=0;j<nr;j++)
scanf("%d",&alloc[i][j]);
/*Available Resource calculation*/
for(j=0;j<nr;j++)
{
avail[j]=r[j];
for(i=0;i<np;i++)
{
avail[j]-=alloc[i][j];


}
}


//marking processes with zero allocation
```

```
for(i=0;i<np;i++)
{
int count=0;
 for(j=0;j<nr;j++)
   {
      if(alloc[i][j]==0)
        count++;
      else
        break;
    }
 if(count==nr)
 mark[i]=1;
}
// initialize W with avail


for(j=0;j<nr; j++)
    w[j]=avail[j];


//mark processes with request less than or equal to W
for(k=0;k<np;k++)
{
for(i=0;i<np; i++)
{
int canbeprocessed= 0;
 if(mark[i]!=1)
{
   for(j=0;j<nr;j++)
    {
      if(request[i][j]<=w[j])
      canbeprocessed=1;
      else
         {
         canbeprocessed=0;
         break;
          }
     }
if(canbeprocessed)
{
mark[i]=1;
for(j=0;j<nr;j++)
w[j]+=alloc[i][j];
```

```
break;
}
}
}
}
//checking for unmarked processes
int deadlock=0;
for(i=0;i<np;i++)
{
printf("%d",mark[i]);
if(mark[i]!=1)
deadlock=1;
}
if(deadlock==1)
printf("\n Deadlock detected");
else
printf("\n No Deadlock possible");
}
```

SAMPLE OUTPUT

```
Enter the no of the process: 5

Enter the no of resources: 3

Total Amount of the Resource R  1: 7

Total Amount of the Resource R  2: 2

Total Amount of the Resource R  3: 6

Enter the request matrix:
0 0 0
2 0 2
0 0 0
1 0 0
0 0 2

Enter the allocation matrix:
0 1 0
2 0 0
3 0 3
2 1 1
0 0 2
11111
 No Deadlock possible
--------------------------------
Process exited after 49.84 seconds with return value 22
Press any key to continue . . .
```

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

a) Worst-fit

b) Best-fit

c) First-fit

Simulate the following situation:

Example

Consider a swapping system in which memory consists of the following whole sizes in memory order: 10K, 4k, 20k, 18k, 7k, 9k, 12k, and 15k. Which hole is taken for successive segment request of i)12k, ii)10k, iii)9k for first fit? Now repeat the question for best fit and worst fit.

First Fit		
12k	→	20k
10k	→	10k
9k	→	18k

Best Fit		
12k	→	12k
10k	→	10k
9k	→	9k

Worst Fit		
12k	→	20k
10k	→	18k
9k	→	15k

```
#include <stdio.h>
#include<stdlib.h>
#define max 25
void readInput(int *nb, int *nf, int b[], int f[]);
void bestFit(int nb, int nf, int b[], int f[], int bf[], int ff[], int frag[]);
void worstFit(int nb, int nf, int b[], int f[], int bf[], int ff[], int frag[]);
void firstFit(int nb, int nf, int b[], int f[], int bf[], int ff[], int frag[]);
void displayResults(int nf, int f[], int ff[], int b[], int frag[]);
int main()
{
    int nb, nf, ch;
    int b[max], f[max], bf[max] ={0}, ff[max]={0}, frag[max]={0};
    readInput(&nb, &nf, b, f);
    printf("1.Best Fit 2.Worst Fit 3.First Fit 4. Exit\n");
    scanf("%d",&ch);

    switch(ch)
    {
```

```
        case 1: bestFit(nb, nf, b, f, bf, ff, frag);
                break;
        case 2: worstFit(nb, nf, b, f, bf, ff, frag);
                break;
        case 3: firstFit(nb, nf, b, f, bf, ff, frag);
                break;
        case 4: exit(0);
                break;
        default: printf("Inavlid choice\n");
                 break;
    }
    displayResults(nf, f, ff, b, frag);
    return 0;
}

void readInput(int *nb, int *nf, int b[], int f[])
{
    int i;
    printf("Enter 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]);
    }
}

void bestFit(int nb, int nf, int b[], int f[], int bf[], int ff[], int frag[])
```

```
{
    int i, j, temp, lowest = 999;

    for (i = 1; i <= nf; i++)
    {
        for (j = 1; j <= nb; j++)
        {
            if (bf[j] != 1) //if bf[j] is not allocated
            {
                temp = b[j] - f[i];
                if (temp >= 0)
                {
                    if (lowest>temp)
                    {
                        ff[i] = j;
                        lowest = temp;
                    }
                }
            }
        }
        frag[i] = lowest;
        bf[ff[i]] = 1;
        lowest = 999;
    }
}

void worstFit(int nb, int nf, int b[], int f[], int bf[], int ff[], int frag[])
{
    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)
                {
                    if (lowest == 10000 || temp > lowest)
```

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

void firstFit(int nb, int nf, int b[], int f[], int bf[], int ff[], int frag[])
{
  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;
          break;
        }
      }
    }
    frag[i] = temp;
    bf[ff[i]] = 1;
  }
}

void displayResults(int nf, int f[], int ff[], int b[], int frag[])
{
  int i;
```

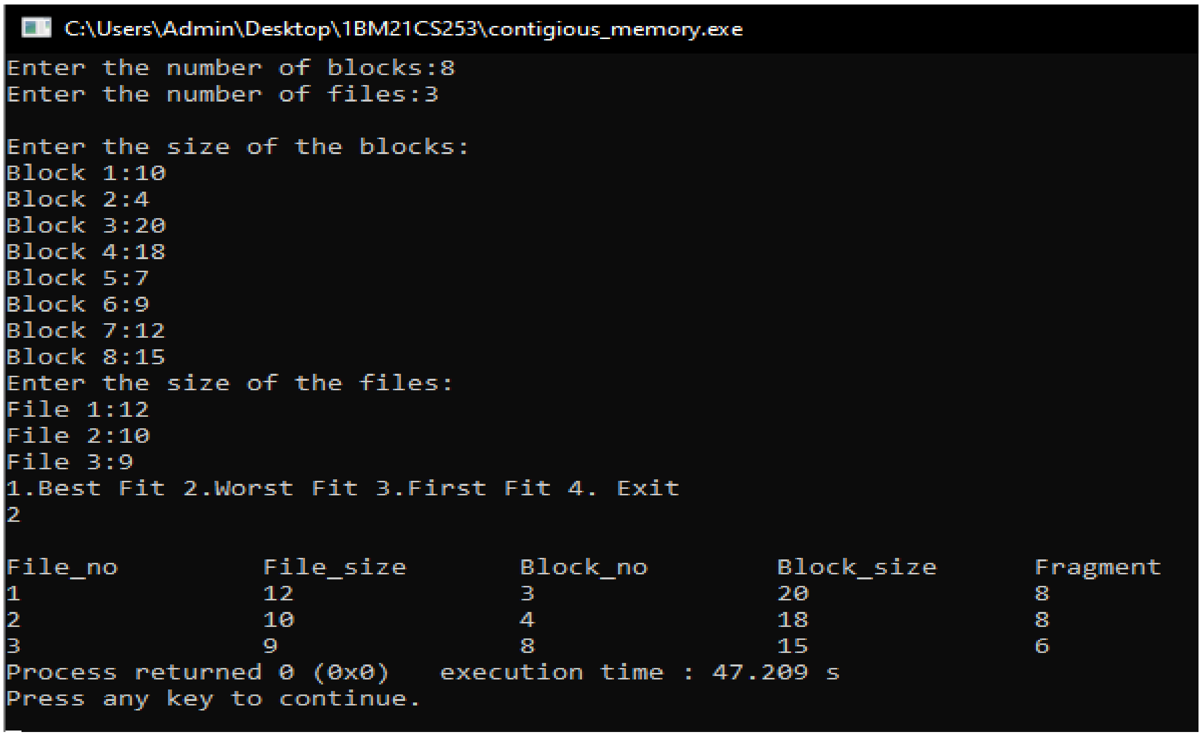
```
    printf("\nFile_no\t\tFile_size\tBlock_no\tBlock_size\tFragment");
    for (i = 1; i <= nf; i++)
    {
        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]);
    }
}
```

SAMPLE OUTPUT:

FOR BEST FIT

```
Enter the number of blocks:8
Enter the number of files:3

Enter the size of the blocks:
Block 1:10
Block 2:4
Block 3:20
Block 4:18
Block 5:7
Block 6:9
Block 7:12
Block 8:15
Enter the size of the files:
File 1:12
File 2:10
File 3:9
1.Best Fit 2.Worst Fit 3.First Fit 4. Exit
1

File_no         File_size       Block_no        Block_size      Fragment
1               12              7               12              0
2               10              1               10              0
3               9               6               9               0
Process returned 0 (0x0)   execution time : 88.838 s
Press any key to continue.
```

FOR WORST FIT



```
C:\Users\Admin\Desktop\1BM21CS253\contigious_memory.exe
Enter the number of blocks:8
Enter the number of files:3

Enter the size of the blocks:
Block 1:10
Block 2:4
Block 3:20
Block 4:18
Block 5:7
Block 6:9
Block 7:12
Block 8:15
Enter the size of the files:
File 1:12
File 2:10
File 3:9
1.Best Fit 2.Worst Fit 3.First Fit 4. Exit
2

File_no         File_size       Block_no        Block_size      Fragment
1               12              3               20              8
2               10              4               18              8
3               9               8               15              6
Process returned 0 (0x0)   execution time : 47.209 s
Press any key to continue.
```

FOR FIRST FIT

```
C:\Users\Admin\Desktop\1BM21CS253\contigious_memory.exe
Enter the number of blocks:8
Enter the number of files:3

Enter the size of the blocks:
Block 1:10
Block 2:4
Block 3:20
Block 4:18
Block 5:7
Block 6:9
Block 7:12
Block 8:15
Enter the size of the files:
File 1:12
File 2:10
File 3:9
1.Best Fit 2.Worst Fit 3.First Fit 4. Exit
3

File_no         File_size       Block_no        Block_size      Fragment
1               12              3               20              8
2               10              1               10              0
3               9               4               18              9
Process returned 0 (0x0)   execution time : 83.570 s
Press any key to continue.
```

10. Write a C program to simulate disk scheduling algorithms

a) FCFS

b) SCAN

c) C-SCAN

```
#include<stdio.h>

#include<conio.h>

int head,a[20],range,n;

void fcfs()

{

int headm=0,temp,i;

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

{

if(i==0)

{

if(a[i]<head)
```

```
headm=headm+(head-a[i]);
else
headm=headm+(a[i]-head);
}
else
{
if(a[i-1]<a[i])
headm=headm+(a[i]-a[i-1]);
else
headm=headm+(a[i-1]-a[i]);
}
}
printf("\nFCFS-Total head movement=%d\n",headm);
}
void scan()
{
int headm=0,i,dir,temp,cnt=0;
printf("\nEnter the direction, upward/right=1, downward/left=-1:");
scanf("%d",&dir);
if(dir==1)
{
for(i=0;i<n;i++)
{
if(a[i]<head)
{
cnt++;
continue;
}
```

```
else if(i==cnt)
headm=headm+(a[i]-head);
else
headm=headm+(a[i]-a[i-1]);
}
headm=headm+(range-a[i-1]);
headm+=(range-a[cnt-1]);
for(i=cnt-1;i>0;i--)
{
headm+=(a[i]-a[i-1]);
}
}
else
{
for(i=0;i<n;i++)
{
if(a[i]>head)
break;
else
cnt++;
}
headm+=(head-a[cnt-1]);
for(i=cnt-1;i>0;i--)
{
headm+=(a[i]-a[i-1]);
}
headm+=(a[0]-0);
headm+=(a[cnt]-0);
```

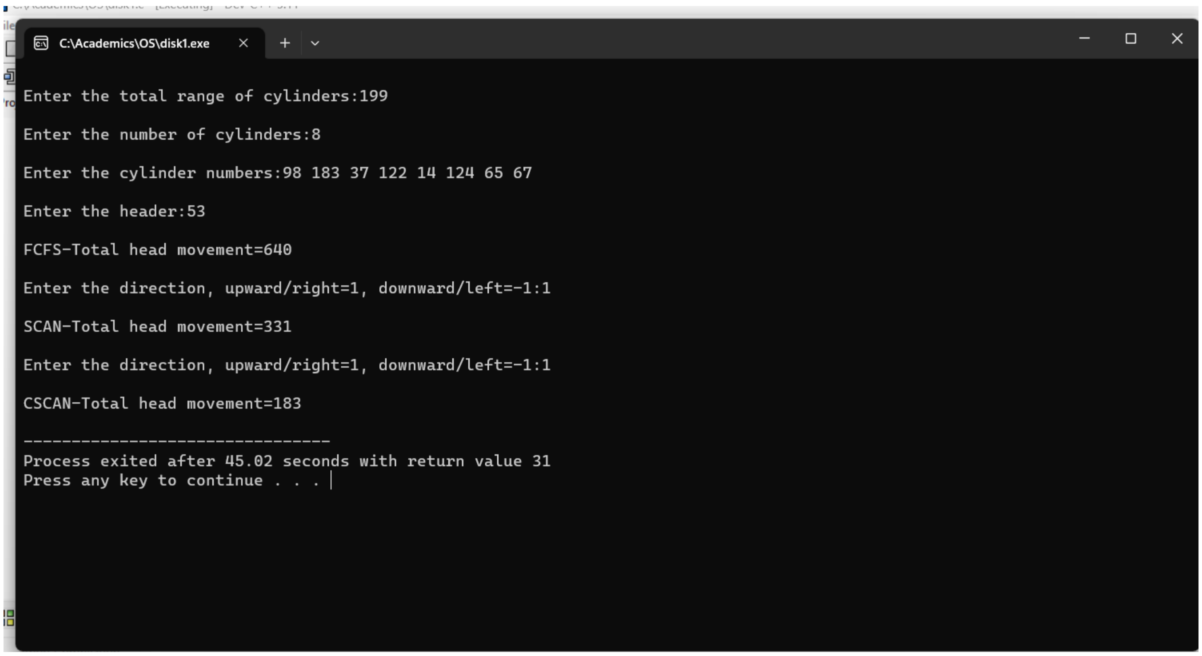
```
for(i=cnt;i<n-1;i++)
{
headm+=(a[i+1]-a[i]);
}
}
printf("\nSCAN-Total head movement=%d\n",headm);

}
void cscan()
{
int headm=0,i,dir,temp,cnt=0;
printf("\nEnter the direction, upward/right=1, downward/left=-1:");
scanf("%d",&dir);
if(dir==1)
{
for(i=0;i<n;i++)
{
if(a[i]<head)
{
cnt++;
continue;
}
else if(i==cnt)
headm=headm+(a[i]-head);
else
headm=headm+(a[i]-a[i-1]);
}
headm=headm+(range-a[i-1]);
```

```
for(i=cnt-1;i>0;i--)
{
headm+=(a[i]-a[i-1]);
}
headm+=(a[i]-0);
}
else
{
for(i=0;i<n;i++)
{
if(a[i]>head)
break;
else
cnt++;
}
headm+=(head-a[cnt-1]);
for(i=cnt-1;i>0;i--)
{
headm+=(a[i]-a[i-1]);
}
headm+=(a[0]-0);
for(i=cnt;i<n-1;i++)
{
headm+=(a[i+1]-a[i]);
}
headm=headm+(range-a[i]);


}
```

```
printf("\nCSCAN-Total head movement=%d\n",headm);


}
void main()
{
int i,j,temp;
printf("\nEnter the total range of cylinders:");
scanf("%d",&range);
printf("\nEnter the number of cylinders:");
scanf("%d",&n);
printf("\nEnter the cylinder numbers:");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
printf("\nEnter the header:");
scanf("%d",&head);
fcfs();
for(i=0;i<n-1;i++)
{
for(j=0;j<n-i-1;j++)
{
if(a[j]>a[j+1])
{
temp=a[j];
a[j]=a[j+1];
a[j+1]=temp;
}
```

```
}
}
scan();
cscan();
}
```

SAMPLE OUTPUT



11. Write a C program to simulate disk scheduling algorithms

a) SSTF
b) LOOK
c) c-LOOK

```
#include<stdio.h>
#include<conio.h>
int head,a[20],range,n;
void sstf()
{
int c=0,i,j,headm=0,k,t,temp,b[20];
for(i=0;i<n;i++)
{
   b[i]=a[i];
}
b[n]=head;
for(i=0;i<n;i++)
{
for(j=0;j<n-i;j++)
{
if(b[j]>b[j+1])
{
temp=b[j];
b[j]=b[j+1];
b[j+1]=temp;
}
}
}
```

```
for(i=0;i<n;i++)
{
if(b[i]==head)
break;
else
c++;
}
j=c;
k=c;
t=j;
for(i=0;i<n;i++)
{
if((b[k+1]-b[t])<(b[t]-b[j-1]) && j>0)
{
headm+=(b[k+1]-b[t]);
k++;
t=k;
}
else if(j==0)
{
headm+=(b[k+1]-b[t]);
k++;
t=k;
}
else
{
headm+=(b[t]-b[j-1]);
j--;
```

```
t=j;
}
}
printf("SSTF-Total head movement=%d\n",headm);
}
void look()
{
int headm=0,i,dir,temp,cnt=0;
printf("Enter the direction, upward/right=1, downward/left=-1:\n");
scanf("%d",&dir);
if(dir==1)
{
for(i=0;i<n;i++)
{
if(a[i]<head)
{
cnt++;
continue;
}
else if(i==cnt)
headm=headm+(a[i]-head);
else
headm=headm+(a[i]-a[i-1]);
}
headm+=a[n-1]-a[cnt-1];
for(i=cnt-1;i>0;i--)
{
headm+=(a[i]-a[i-1]);
```

```
}
}
else
{
for(i=0;i<n;i++)
{
if(a[i]>head)
break;
else
cnt++;
}
headm+=(head-a[cnt-1]);
for(i=cnt-1;i>0;i--)
{
headm+=(a[i]-a[i-1]);
}
headm+=(a[cnt]-a[0]);
for(i=cnt;i<n-1;i++)
{
headm+=(a[i+1]-a[i]);
}
}
printf("LOOK-Total head movement=%d\n",headm);

}
void clook()
{
int headm=0,i,dir,temp,cnt=0;
```

```
printf("Enter the direction, upward/right=1, downward/left=-1:\n");
scanf("%d",&dir);
if(dir==1)
{
for(i=0;i<n;i++)
{
if(a[i]<head)
{
cnt++;
continue;
}
else if(i==cnt)
headm=headm+(a[i]-head);
else
headm=headm+(a[i]-a[i-1]);
}
for(i=cnt-1;i>0;i--)
{
headm+=(a[i]-a[i-1]);
}
}
else
{
for(i=0;i<n;i++)
{
if(a[i]>head)
break;
else
```

```
cnt++;
}
headm+=(head-a[cnt-1]);
for(i=cnt-1;i>0;i--)
{
headm+=(a[i]-a[i-1]);
}
for(i=cnt;i<n-1;i++)
{
headm+=(a[i+1]-a[i]);
}

}
printf("\nCLOOK-Total head movement=%d\n",headm);
}
void main()
{
int i,j,temp;
printf("\nEnter the total range of cylinders:");
scanf("%d",&range);
printf("\nEnter the number of cylinders:");
scanf("%d",&n);
printf("\nEnter the header:");
scanf("%d",&head);
printf("\nEnter the cylinder numbers:");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
```

```
}
for(i=0;i<n-1;i++)
{
for(j=0;j<n-i-1;j++)
{
if(a[j]>a[j+1])
{
temp=a[j];
a[j]=a[j+1];
a[j+1]=temp;
}
}
}
sstf();
look();
clook();
}
```

SAMPLE OUTPUT

```
C:\Academics\OS\disk2.exe

Enter the total range of cylinders:199

Enter the number of cylinders:8

Enter the header:53

Enter the cylinder numbers:98 183 37 122 14 124 65 67
SSTF-Total head movement=236
Enter the direction, upward/right=1, downward/left=-1:
1
LOOK-Total head movement=299
Enter the direction, upward/right=1, downward/left=-1:
-1

CLOOK-Total head movement=157

--------------------------------
Process exited after 62.31 seconds with return value 31
Press any key to continue . . .
```

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

```
#include<stdio.h>

#include<conio.h>

int n,m,a[20],p[10];

void fifo()

{

        int i,j,flag,cnt=0,k=0;

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

```
        {
                flag=1;
                for(j=0;j<m;j++)
                {
                        if(a[i]==p[j])
                        {
                                flag=0;
                                break;
                        }
                }
                if(flag==1)
                {
                cnt++;
                p[k]=a[i];
                k=(k+1)%m;
                }
        }
        printf("\nFIFO-Page faults=%d",cnt);
}
void optimal()
{
        int i,j,flag,cnt=0,k=0,t,temp,f,help[10],ct;
        for(i=0;i<n;i++)
        {
                flag=1,f=1,ct=0;
                for(j=0;j<m;j++)
                {
                        help[j]=0;
```

```
        if(a[i]==p[j])
        {
                flag=0;
                break;
        }
}
if(flag==1)
{
        cnt++;
        for(j=0;j<m;j++)
        {
                if(p[j]==-1)
                {
                        p[j]=a[i];
                        f=0;
                        break;
                }
        }
        if(f==1)
        {
                for(k=i+1;k<n;k++)
                {
                        for(j=0;j<m;j++)
                        {
                                if(p[j]==a[k]&&help[j]==0)
                                {
                                        temp=j;
                                        help[j]=1;
```

```
                    }
                }
            }
            for(j=0;j<m;j++)
            {
                if(help[j]==0)
                temp=j;
            }
        p[temp]=a[i];
        }
    }
}
printf("\nOPTIMAL-Page faults=%d",cnt);
}
void lru()
{
    int flag,f,k,cnt=0,i,j,temp,ct,help[10];
    for(i=0;i<n;i++)
    {
        flag=1,f=1,ct=0;
        for(j=0;j<m;j++)
        {
            help[j]=0;
            if(p[j]==a[i])
            {
                flag=0;
                break;
            }
```

```
}
if(flag==1)
{
        cnt++;
for(j=0;j<m;j++)
{
        if(p[j]==-1)
        {
                p[j]=a[i];
                {
                        f=0;
                        break;
                }
        }
}
if(f==1)
{
        for(k=i-1;k>=0;k--)
        {
                for(j=0;j<m;j++)
                {
                        if(p[j]==a[k]&& help[j]==0)
                        {
                                temp=j;
                                help[j]=1;
                        }
                }
        }
```

```
                    p[temp]=a[i];
                }

        }

    }
    printf("\nLRU-Page faults=%d",cnt);
}
void main()
{
    int i;
    printf("Enter the number of pages:");
    scanf("%d",&n);
    printf("\nEnter the page numbers:");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    printf("\nEnter the number of frames:");
    scanf("%d",&m);
    for(i=0;i<m;i++)
    p[i]=-1;
    fifo();
    for(i=0;i<m;i++)
    p[i]=-1;
    optimal();
    for(i=0;i<m;i++)
    p[i]=-1;
```

```
    lru();
}
```

SAMPLE OUTPUT

```
Enter the number of pages:14

Enter the page numbers:0 4 3 2 1 4 6 3 0 8 9 3 8 5

Enter the number of frames:3

FIFO-Page faults=13
OPTIMAL-Page faults=10
LRU-Page faults=13
--------------------------------
Process exited after 30.75 seconds with return value 19
Press any key to continue . . .
```

13. Write a C program to simulate paging technique of memory management. (create a logical memory space, physical memory space and page table, you should show the address translation entirely)

```
#include<stdio.h>

#include<conio.h>

main()

{

 int ms, ps, nop, np, rempages, i, j, x, y, pa, offset;

 int s[10], fno[10][20];
```

```
printf("\nEnter the memory size -- ");
scanf("%d",&ms);

printf("\nEnter the page size -- ");
scanf("%d",&ps);

nop = ms/ps;
printf("\nThe no. of pages available in memory are -- %d ",nop);

printf("\nEnter number of processes -- ");
 scanf("%d",&np);
rempages = nop;
for(i=1;i<=np;i++)

{

printf("\nEnter no. of pages required for p[%d]-- ",i);
 scanf("%d",&s[i]);

if(s[i] >rempages)
{

printf("\nMemory is Full");
break;
```

```
}
rempages = rempages - s[i];

printf("\nEnter pagetable for p[%d] --- ",i);
 for(j=0;j<s[i];j++)
scanf("%d",&fno[i][j]);
}

printf("\nEnter Logical Address to find Physical Address ");
printf("\nEnter process no. and pagenumber and offset -- ");

scanf("%d %d %d",&x,&y, &offset);



if(x>np || y>=s[i] || offset>=ps)
printf("\nInvalid Process or Page Number or offset");

else
{ pa=fno[x][y]*ps+offset;
printf("\nThe Physical Address is -- %d",pa);

}
getch();
```

}

SAMPLE OUTPUT

```
C:\Users\Admin\Desktop\1BM21CS253\paging_technique.exe

Enter the memory size -- 128

Enter the page size -- 8

The no. of pages available in memory are -- 16
Enter number of processes -- 3

Enter no. of pages required for p[1]-- 5

Enter pagetable for p[1] --- 2
5
7
9
11

Enter no. of pages required for p[2]-- 3

Enter pagetable for p[2] --- 3
4
8

Enter no. of pages required for p[3]-- 1

Enter pagetable for p[3] --- 13

Enter Logical Address to find Physical Address
Enter process no. and pagenumber and offset -- 1 5 4

The Physical Address is -- 4
```