MEDICAPS UNIVERSITY, INDORE

Department of Computer Science & Engineering
FACULTY OF ENGINEERING

Lab Manual

Operating System [CS3CO36]

Medi-Caps University, Indore

Department of Computer Science & Engineering

INDEX

S.no.	List of Practical
1.	Study of latest Operating System and their Key features
2.	WAP to implement FCFS Scheduling Algorithm and Shortest Job First Scheduling Algorithm
3.	WAP to implement Round Robin Scheduling Algorithm
4.	WAP to implement Priority Scheduling Algorithm
5.	WAP to implement the Producer Consumer Problem
6.	WAP to implement the Dining Philosopher’s Problem
7.	WAP to implement the Banker’s Algorithm
8.	WAP to implement the concept of First Fit memory allocation and Best fit memory allocation.
9.	WAP to implement the concept of Worst Fit memory allocation.
10.	WAP a program to implement FIFO Page Replacement algorithm.
11.	WAP a program to implement LRU Page Replacement algorithms
12.	WAP a program to implement Optimal Page Replacement algorithm.
13.	WAP to implement FCFS Disk Scheduling
14.	WAP to implement SCAN Disk Scheduling and C-SCAN Disk Scheduling
15.	WAP to implement and LOOK Disk Scheduling and C-LOOK Disk Scheduling

Practical – 1

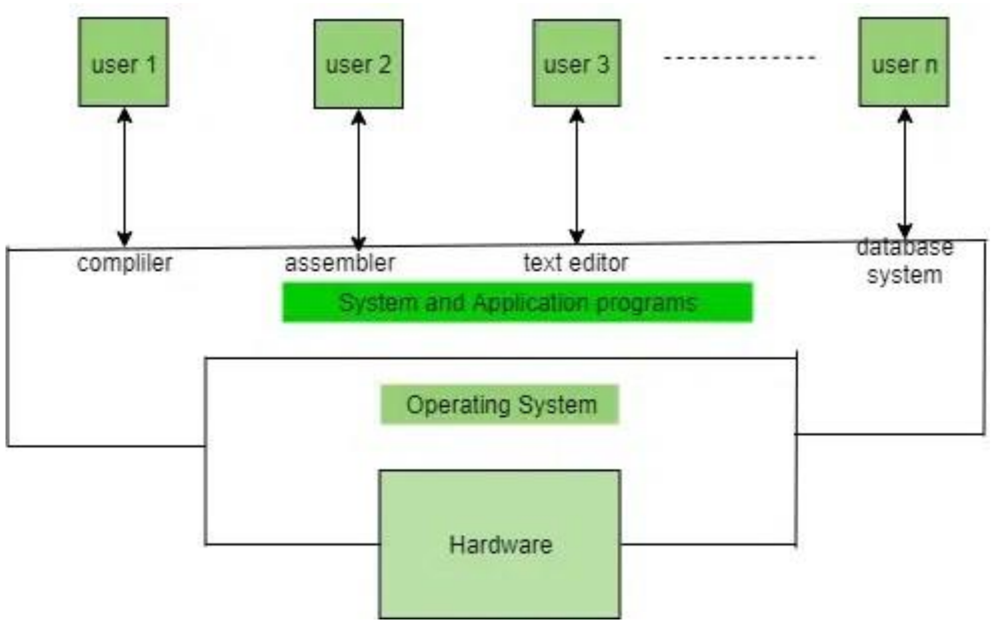
Objective: Study of latest Operating System and their latest features

Theory: An operating system (OS) is software that manages computer hardware and provides services for computer programs. It serves as an intermediary between the hardware and the applications, ensuring that the computer's resources are utilized efficiently and that different software can run smoothly on the same hardware.

Key functions of an operating system include:

1. **Process Management:** It manages processes, which are instances of executing programs. This involves tasks such as process scheduling, creation, and termination.
2. **Memory Management:** The OS allocates and deallocates memory space as needed by different programs, ensuring that each program has the necessary resources to run.
3. **File System Management:** It oversees the organization, storage, and retrieval of data on storage devices, such as hard drives or solid-state drives.
4. **Device Management:** The OS controls and coordinates the use of hardware devices, such as printers, disk drives, and input/output devices.
5. **Security and Protection:** It enforces security policies, controls access to resources, and protects the system and its data from unauthorized access and malicious software.
6. **User Interface:** The OS provides a user interface, which can be command-line based or graphical, allowing users to interact with the computer.

Examples of popular operating systems include Microsoft Windows, macOS, Linux, and Unix. Each operating system has its own set of features, functionalities, and user interfaces, catering to different user preferences and requirements.



Serial No	Name of Operating System	Launching Date	Key Features
1	Windows 10 20H2	20 Oct 2020	Cortana Desktop, Xbox App, Universal App, Multiple Desktop
2	Windows 8.1	8 Oct 2014	Metro Apps, Tablet Ready, SkyDrive Connectivity, Touch Interface
3	Windows 7	19 March 2019	Touch, Speech and Handwriting Recognition, Support for additional file format
4	Ubuntu 20.10	22 Oct 2020	Muted Mic Indicator, App Recorder, Improves Fingerprint login support
5	Mac OS 11 Big Sur	12 Nov 2020	Design refresh, Control centre, Maps overhaul, Messaging Apps
6	Fedora 33	29 Sep 2020	Animated Background, BIOS support, NANO (a terminal text based editor)
7	Solaris 11.4	28 Aug 2018	Top level device Removal, Scheduled Scrub, Send Streams
8	Free BSD 11.4	23 June 2020	More security, Firewalls
9	Chrome OS	7 Jan 2021	File App Upgrade, Tab grouping improvements, Better pdf viewing
10	CentOS Linux	12 Nov 2020	Extended Device Support, New boxes features, New on-screen keyboard

11	Debian 10.7	5 Dec 2020	Security updates, Bugfixes, Removes packages
12	Deepin	30 Dec 2020	New enchanting graphical interface, updated deepin installer, App store update
13	Red Hat Linux	7 May 2019	Cloud containers, storage application development, virtualization
14	Oracle Linux	13 Nov 2020	Leading security, high Performance
15	IOS	June 2020	Picture-in-picture support, New app library, translate app, app clips

Practical – 2

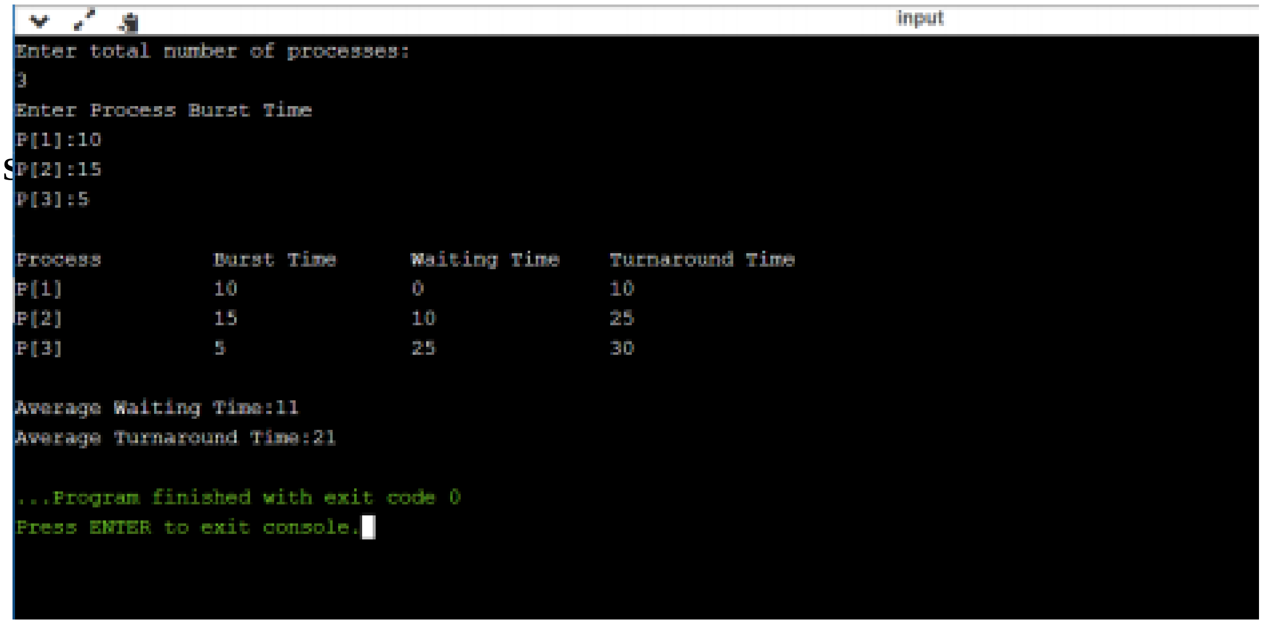
Objective: Write a Program to implement FCFS Scheduling Algorithm and Shortest Job First Scheduling Algorithm.

Theory: First-Come, First-Served (FCFS) CPU scheduling is a simple scheduling algorithm in operating systems where processes are executed in the order they arrive in the ready queue. The first process that enters the queue is the first one to be executed, and subsequent processes are scheduled in the order of their arrival. While FCFS is easy to understand and implement, it may lead to the "convoy effect," where short processes get delayed behind long processes, impacting overall system efficiency. This scheduling method does not consider factors like the burst time or priority of processes and is often used in scenarios where simplicity is prioritized over optimizing resource utilization. Shortest Job First (SJF) is a CPU scheduling algorithm in operating systems that prioritizes the execution of processes based on their burst time or execution time. The process with the shortest burst time is selected for execution first, aiming to minimize the overall waiting time and turnaround time. SJF can be preemptive or non-preemptive; in the preemptive version, if a new process with a shorter burst time arrives while another process is executing, the currently running process may be interrupted. This algorithm is efficient in reducing average turnaround time and improving system throughput by favoring shorter tasks, but it can pose challenges in predicting the exact burst time, leading to issues like starvation for longer processes if not implemented carefully.

Program of FCFS Scheduling Algorithm:

```
#include<iostream>

 using namespace std;

 int main()

{

 int n,bt[20],wt[20],tat[20],avwt=0,avtat=0,i,j;

 cout<<"Enter total number of processes:\n";

 cin>>n;

 cout<<"Enter Process Burst Time\n";

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

 {

 cout<<"P["<<i+1<<"]:";

 cin>>bt[i];
 }
 wt[0]=0;
for(i=1;i<n;i++)
 {
 wt[i]=0;
 for(j=0;j<i;j++)
 wt[i]+=bt[j];
 }
 cout<<"\nProcess\t\tBurst Time\tWaiting Time\tTurnaround
Time";  for(i=0;i<n;i++)
 {
tat[i]=bt[i]+wt[i];
 avwt+=wt[i];
 avtat+=tat[i];
 cout<<"\nP["<<i+1<<"]"<<"\t\t"<<bt[i]<<"\t\t"<<wt[i]<<"\t\t"<<tat[i];
 }

 avwt/=i;
 avtat/=i;
 cout<<"\n\nAverage Waiting Time:"<<avwt;
 cout<<"\nAverage Turnaround Time:"<<avtat;

 return 0;
}
```

Output of FCFS Scheduling:



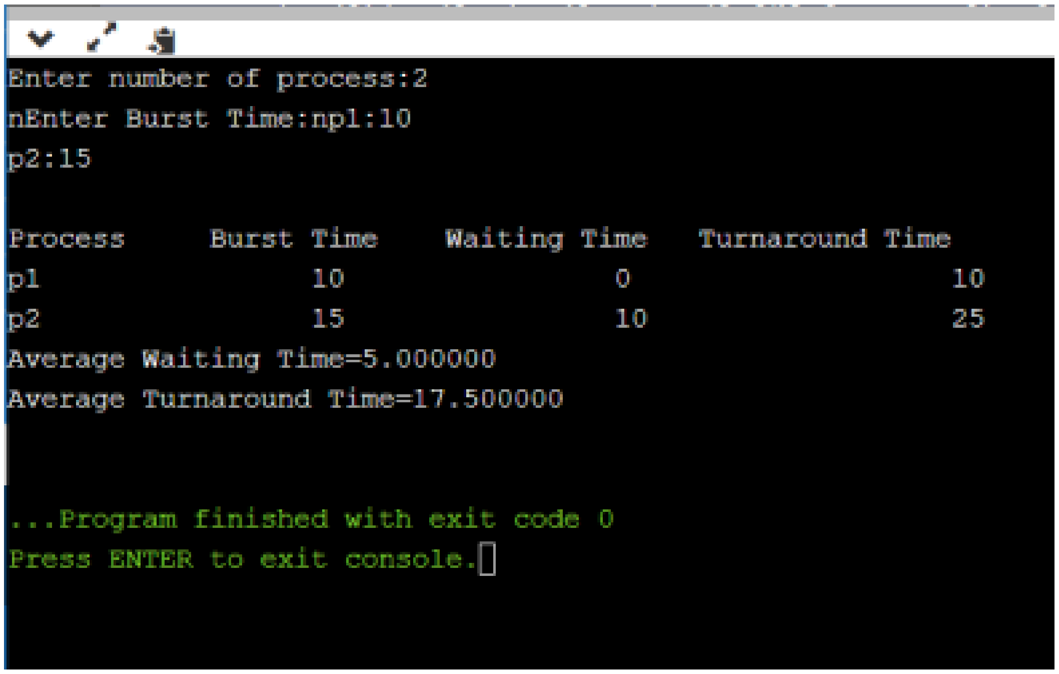
```
Enter total number of processes:
3
Enter Process Burst Time
P[1]:10
P[2]:15
P[3]:5

Process         Burst Time      Waiting Time    Turnaround Time
P[1]            10              0               10
P[2]            15              10              25
P[3]            5               25              30

Average Waiting Time:11
Average Turnaround Time:21

...Program finished with exit code 0
Press ENTER to exit console.
```

Program of Shortest Job first Scheduling Algortihm:

```
#include<stdio.h>
 int main()
{
 int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;
 float avg_wt,avg_tat;
 printf("Enter number of process:");
  scanf("%d",&n);
 printf("nEnter Burst Time:n");
 for(i=0;i<n;i++)
 {
 printf("p%d:",i+1);
 scanf("%d",&bt[i]);
 p[i]=i+1;
 }
 for(i=0;i<n;i++)
 {
 pos=i;
 for(j=i+1;j<n;j++)
 {
 if(bt[j]<bt[pos])
 pos=j;
 }
temp=bt[i];
 bt[i]=bt[pos];
 bt[pos]=temp;


 temp=p[i];
 p[i]=p[pos];
 p[pos]=temp;
 }
 wt[0]=0;
 for(i=1;i<n;i++)
 {
 wt[i]=0;
 for(j=0;j<i;j++)
 wt[i]+=bt[j];
 total+=wt[i];
 }
```

```
avg_wt=(float)total/n;
 total=0;


 printf("\nProcess\t Burst Time Waiting Time Turnaround
Time");  for(i=0;i<n;i++)
 {
 tat[i]=bt[i]+wt[i];
 total+=tat[i];
 printf("\np%d\t\t %d\t\t %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
avg_tat=(float)total/n;
 printf("\nAverage Waiting Time=%f",avg_wt);
 printf("\nAverage Turnaround Time=%f\n",avg_tat);
}
```

Output of SJF Algorithm:



```
Enter number of process:2
nEnter Burst Time:np1:10
p2:15

Process     Burst Time    Waiting Time   Turnaround Time
p1                10                  0                       10
p2                15                  10                      25
Average Waiting Time=5.000000
Average Turnaround Time=17.500000


...Program finished with exit code 0
Press ENTER to exit console.
```

Sample Questions:

1. Can you explain the concept of the ready queue in the context of FCFS scheduling?
2. What is the main advantage of FCFS scheduling?
3. Discuss a scenario where FCFS scheduling might lead to poor system performance.
4. How does FCFS handle the waiting time and turnaround time of processes?
5. Can you explain the concept of burst time in the context of SJF scheduling?
6. What is the primary objective of SJF scheduling, and how does it contribute to system performance?
7. Distinguish between preemptive and non-preemptive SJF scheduling.

Practical – 3

Objective: Write a Program to implement Round Robin Scheduling Algorithm

Theory: Round Robin (RR) scheduling is a preemptive CPU scheduling algorithm in operating systems that allocates fixed time slices, often called time quanta or time slots, to each process in a cyclic manner. Processes are arranged in a circular queue, and the scheduler assigns each process a time quantum during which it can execute. Once a process exhausts its time quantum, it is moved to the back of the queue, and the next process in line is given the CPU. This continues until all processes complete their execution. Round Robin is fair and ensures that each process gets an equal share of the CPU, preventing any single process from monopolizing the resources for an extended period. While it may introduce some overhead due to frequent context switching, Round Robin is widely used in time-sharing systems and environments where responsiveness and fairness are essential.

Program:

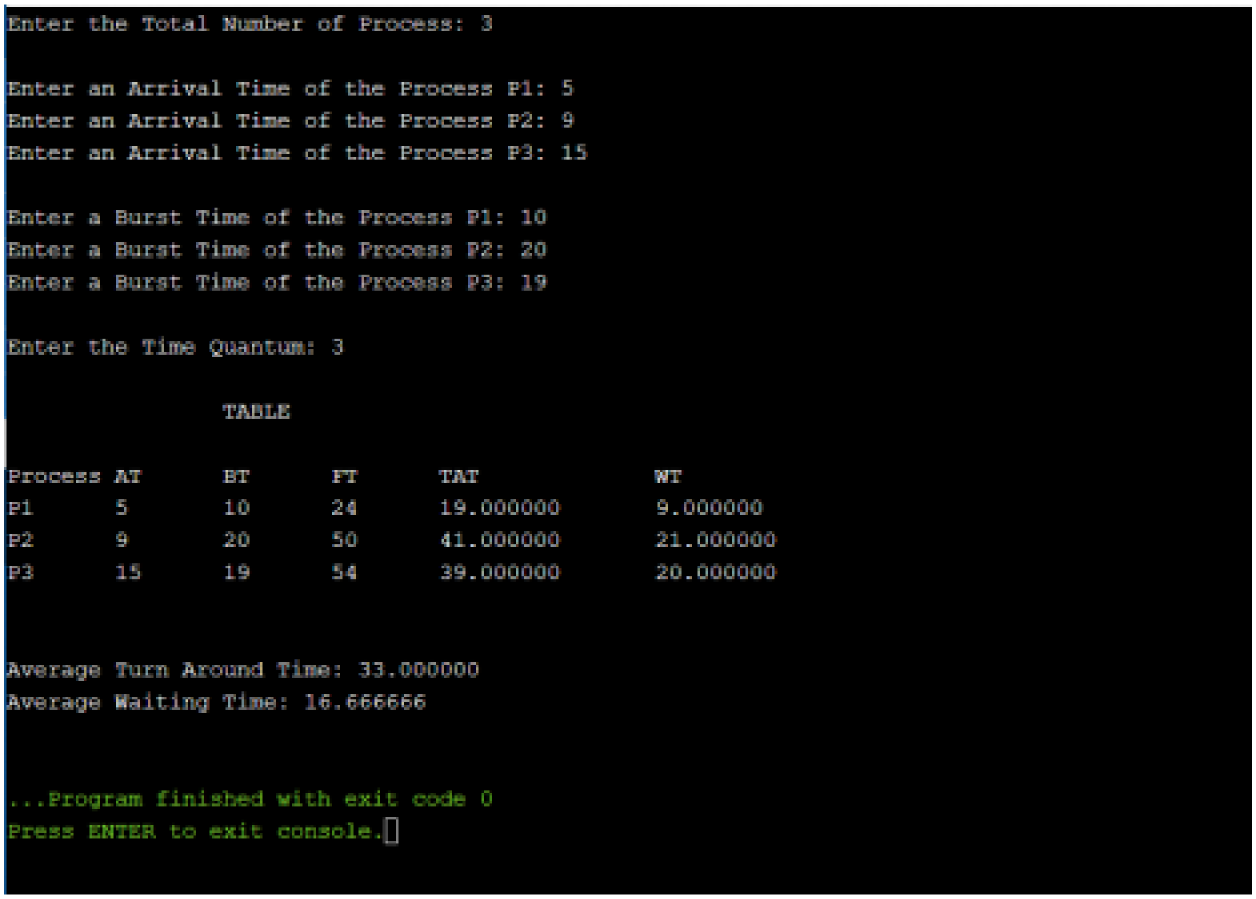
```
#include<iostream>

#include<cstdlib>

#include<queue>

#include<cstdio>

using namespace std;

typedef struct process

{

 int id,at,bt,st,ft,pr;

 float wt,tat;

 }process;

 process p[10],p1[10],temp;

 queue<int> q1;

int accept(int ch);

void turnwait(int n);

void display(int n);

void ganttrr(int n);
```

```
int main()

{

 int i,n,ts,ch,j,x;

 p[0].tat=0;

 p[0].wt=0;


 n=accept(ch);

 ganttrr(n);
turnwait(n);

 display(n);

 return 0;

 }

int accept(int ch)

{

 int i,n;

 printf("Enter the Total Number of Process:

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

 if(n==0)

 {

 printf("Invalid");

 exit(1);

 }

 cout<<endl;

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

 {

 printf("Enter an Arrival Time of the Process P%d:

",i);  scanf("%d",&p[i].at);
```

```
 p[i].id=i;
 }
 cout<<endl;
 for(i=1;i<=n;i++)
 {
 printf("Enter a Burst Time of the Process P%d:
",i);  scanf("%d",&p[i].bt);
 }
{
 p1[i]=p[i];
 }
 return n;
}
void ganttrr(int n)
{
 int i,ts,m,nextval,nextarr;
 nextval=p1[1].at;
 i=1;
 cout<<"\nEnter the Time
Quantum: ";  cin>>ts;
 for(i=1;i<=n &&
p1[i].at<=nextval;i++)  {
 q1.push(p1[i].id);
 }
 while(!q1.empty())
 {
 m=q1.front();
 q1.pop();

 if(p1[m].bt>=ts)

 {

 nextval=nextval+ts;

 }

 else

 {
nextval=nextval+p1[m].bt;

}

 if(p1[m].bt>=ts)

 {

 p1[m].bt=p1[m].bt-ts;

 }

 else

 {

 p1[m].bt=0;

 }


while(i<=n&&p1[i].at<=nextv

al)  {

 q1.push(p1[i].id);

 i++;

 }

  if(p1[m].bt>0)

 {

 q1.push(m);

 }
 if(p1[m].bt<=0)
 {
 p[m].ft=nextval;
 }
 }
}
void turnwait(int n)
{
 int i;
 for(i=1;i<=n;i++)
 {
 p[i].tat=p[i].ft-p[i].at;
 p[i].wt=p[i].tat-p[i].bt;
 p[0].tat=p[0].tat+p[i].tat;
 p[0].wt=p[0].wt+p[i].wt;
 }
 p[0].tat=p[0].tat/n;
 p[0].wt=p[0].wt/n;
}
void display(int n)
{
  int i;
  cout<<"\n TABLE \n";
printf("\nProcess\tAT\tBT\tFT\tTAT\t\tWT");
 for(i=1;i<=n;i++)
 {
 printf("\nP%d\t%d\t%d\t%d\t%f\t%f",p[i].id,p[i].at,p[i].bt,p[i].ft,p[i].tat,p
 [i].wt); }
```

```
cout<<"\n \n";
 printf("\nAverage Turn Around Time: %f",p[0].tat);
 printf("\nAverage Waiting Time: %f\n",p[0].wt);
}
```

Output :



```
Enter the Total Number of Process: 3

Enter an Arrival Time of the Process P1: 5
Enter an Arrival Time of the Process P2: 9
Enter an Arrival Time of the Process P3: 15

Enter a Burst Time of the Process P1: 10
Enter a Burst Time of the Process P2: 20
Enter a Burst Time of the Process P3: 19

Enter the Time Quantum: 3

                TABLE

Process AT      BT      FT      TAT             WT
P1      5       10      24      19.000000       9.000000
P2      9       20      50      41.000000       21.000000
P3      15      19      54      39.000000       20.000000


Average Turn Around Time: 33.000000
Average Waiting Time: 16.666666


...Program finished with exit code 0
Press ENTER to exit console.
```

Sample Questions:

1. What is the significance of the circular queue in the Round Robin algorithm?
2. Discuss the impact of the time quantum size on the performance of Round Robin scheduling.
3. Explain the term "context switching" and how it relates to Round Robin scheduling.
4. What is the main advantage of Round Robin in a time-sharing system?

Practical – 4

Objective: Write a Program to implement Priority Scheduling Algorithm

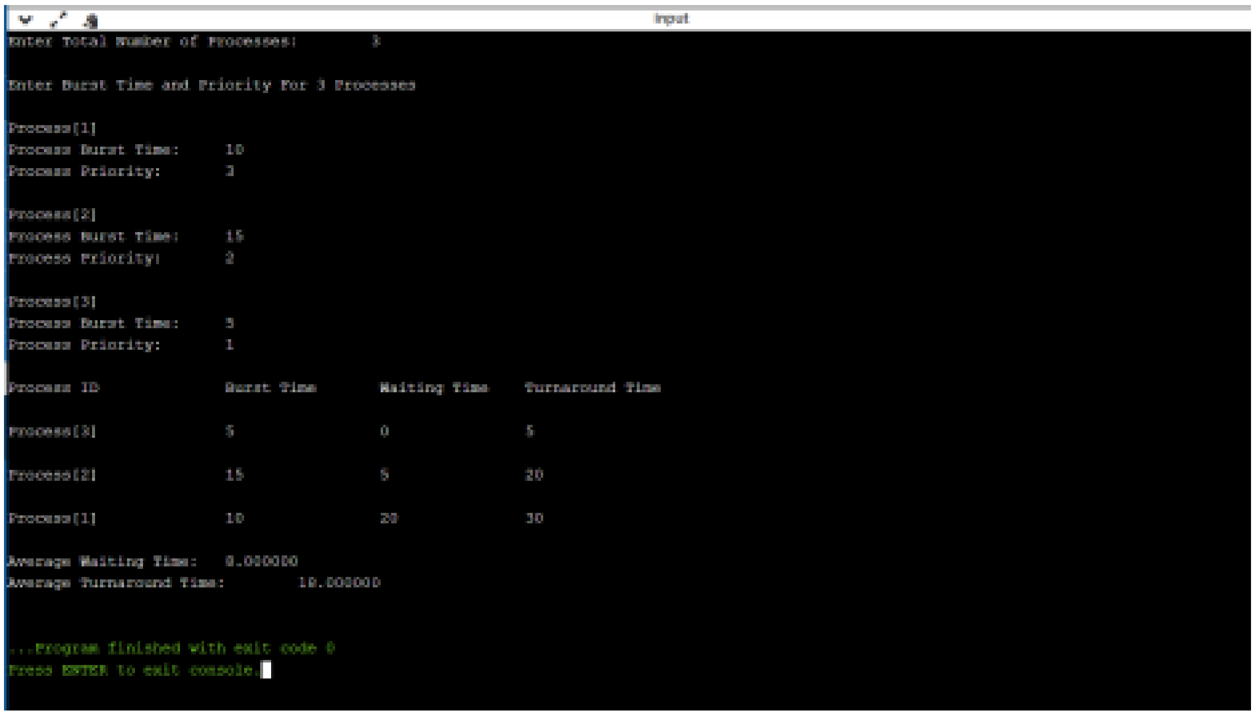
Theory : Priority scheduling is a process scheduling algorithm in computer systems where each process is assigned a priority, and the one with the highest priority is scheduled for execution first. It can be implemented in both preemptive and non-preemptive manners. In non-preemptive priority scheduling, a process continues to run until it completes or voluntarily gives up the CPU, while in preemptive priority scheduling, the scheduler may interrupt a running process to allow a higher-priority process to execute. The priority values are typically based on factors such as system requirements, task importance, or deadlines. One challenge is preventing starvation, where low-priority processes might not get a chance to execute; this can be mitigated through aging mechanisms that gradually increase the priority of waiting processes. Priority scheduling is commonly used in real-time systems and situations where certain tasks require immediate attention and higher priority than others.

Program :

```
#include<stdio.h>

 int main()

 {

 int burst_time[20], process[20], waiting_time[20], turnaround_time[20],

priority[20];  int i, j, limit, sum = 0, position, temp;

 float average_wait_time, average_turnaround_time;

 printf("Enter Total Number of Processes:\t");

 scanf("%d", &limit);

 printf("\nEnter Burst Time and Priority For %d Processes\n",

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

 {

 printf("\nProcess[%d]\n", i + 1);

 printf("Process Burst Time:\t");

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

 printf("Process Priority:\t");

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

```
process[i] = i + 1;
}
for(i = 0; i < limit; i++)
{
position = i;
for(j = i + 1; j < limit; j++)
{
if(priority[j] < priority[position])
{
position = j;
}
}
 temp = priority[i];
priority[i] = priority[position];
priority[position] = temp;
temp = burst_time[i];
burst_time[i] = burst_time[position];
burst_time[position] = temp;
temp = process[i];
process[i] = process[position];
process[position] = temp;
}
waiting_time[0] = 0;
for(i = 1; i < limit; i++)
{
waiting_time[i] = 0;
for(j = 0; j < i; j++)
{
waiting_time[i] = waiting_time[i] + burst_time[j];  }

sum = sum + waiting_time[i];

}

average_wait_time = sum / limit;
sum = 0;

printf("\nProcess ID\t\tBurst Time\t Waiting Time\t Turnaround Time\n");

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

{

turnaround_time[i] = burst_time[i] + waiting_time[i];

sum = sum + turnaround_time[i];

printf("\nProcess[%d]\t\t%d\t\t %d\t\t %d\n", process[i], burst_time[i], waiting_time[i],
turnaround_time[i]);

}

average_turnaround_time = sum / limit;

printf("\nAverage Waiting Time:\t%f", average_wait_time);

printf("\nAverage Turnaround Time:\t%f\n", average_turnaround_time);

return 0;

}
```

Output :



```
Enter Total Number of Processes:        3

Enter Burst Time and Priority For 3 Processes

Process[1]
Process Burst Time:     10
Process Priority:       3

Process[2]
Process Burst Time:     15
Process Priority:       2

Process[3]
Process Burst Time:     5
Process Priority:       1

Process ID              Burst Time      Waiting Time    Turnaround Time

Process[3]              5               0               5

Process[2]              15              5               20

Process[1]              10              20              30

Average Waiting Time:   8.000000
Average Turnaround Time:        18.000000


...Program finished with exit code 0
Press ENTER to exit console.
```

Sample Questions:

1. What is priority scheduling, and how does it differ from other scheduling algorithms?
2. Explain the concept of priority in priority scheduling. How are priorities assigned to processes?
3. What are the advantages of using priority scheduling in a computer system?
4. Can you describe the difference between preemptive and non-preemptive priority scheduling?
5. How does priority scheduling address the issue of starvation?

Practical – 5

Objective: WAP to implement the Producer Consumer Problem

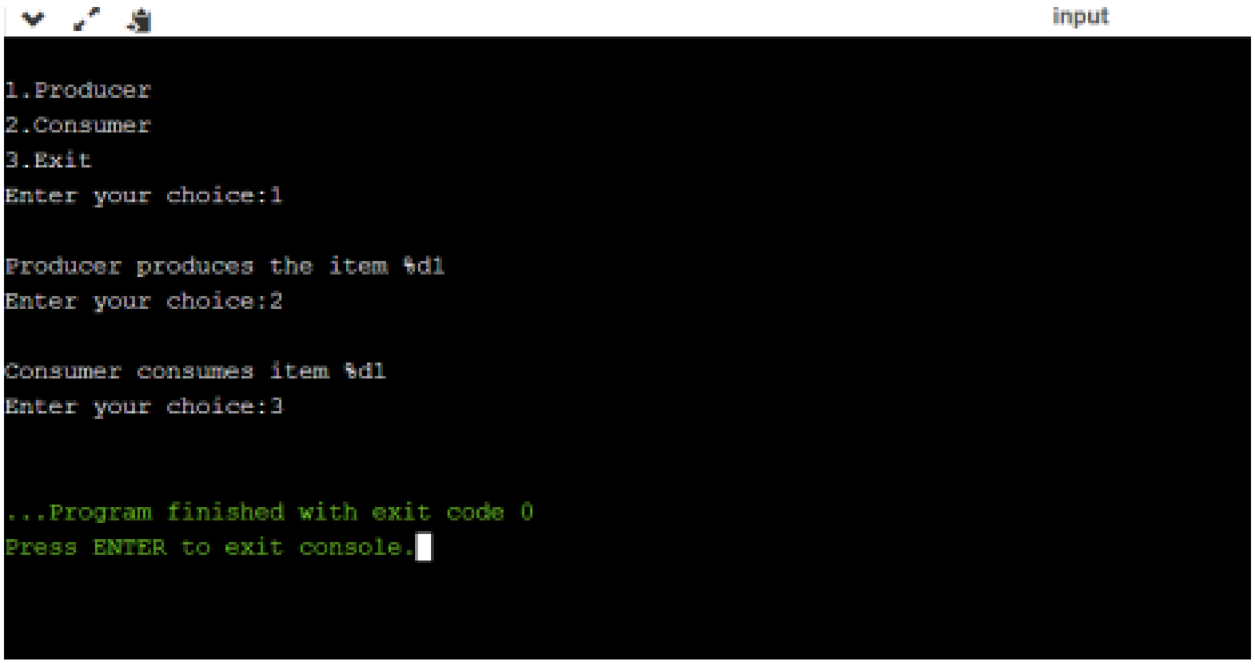
Theory : The producer-consumer problem is a classic synchronization issue in concurrent computing, where there are two types of processes, producers and consumers, sharing a common, fixed-size buffer or queue. Producers generate data items and place them in the buffer, while consumers retrieve and process these items. The challenge lies in ensuring that producers do not attempt to add items to a full buffer, and consumers do not try to retrieve items from an empty buffer simultaneously. Solutions typically involve using semaphores or other synchronization mechanisms to coordinate access to the shared buffer, preventing race conditions and ensuring proper communication and coordination between producers and consumers.

Program :

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int mutex=1,full=0,empty=3,x=0;
 int main()
{
        int n;
        void producer();
        void consumer();
        int wait(int);
        int signal(int);
        cout<<"\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
                                    cout<<"Buffer is full!!";
                                break;
                        case 2: if((mutex==1)&&(full!=0))
                                        consumer();

                                else
                                    cout<<"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++;

cout<<"\nProducer produces the item %d"<<x;

mutex=signal(mutex);

}

void consumer()

{

mutex=wait(mutex);

full=wait(full);

empty=signal(empty);

cout<<"\nConsumer consumes item %d"<<x;

x--;

mutex=signal(mutex);

}
```

Output :



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

Producer produces the item %d1
Enter your choice:2

Consumer consumes item %d1
Enter your choice:3


...Program finished with exit code 0
Press ENTER to exit console.
```

Sample Questions:

1. What is the shared resource in the producer-consumer problem, and why is proper synchronization necessary when accessing it?
2. How do race conditions occur in the context of the producer-consumer problem, and why are they problematic?
3. What are the key challenges in designing a solution for the producer-consumer problem?
4. What are the common synchronization mechanisms used to address the producer-consumer problem, and how do they work?
5. Can you describe the role of semaphores in solving the producer-consumer problem?

Practical – 6

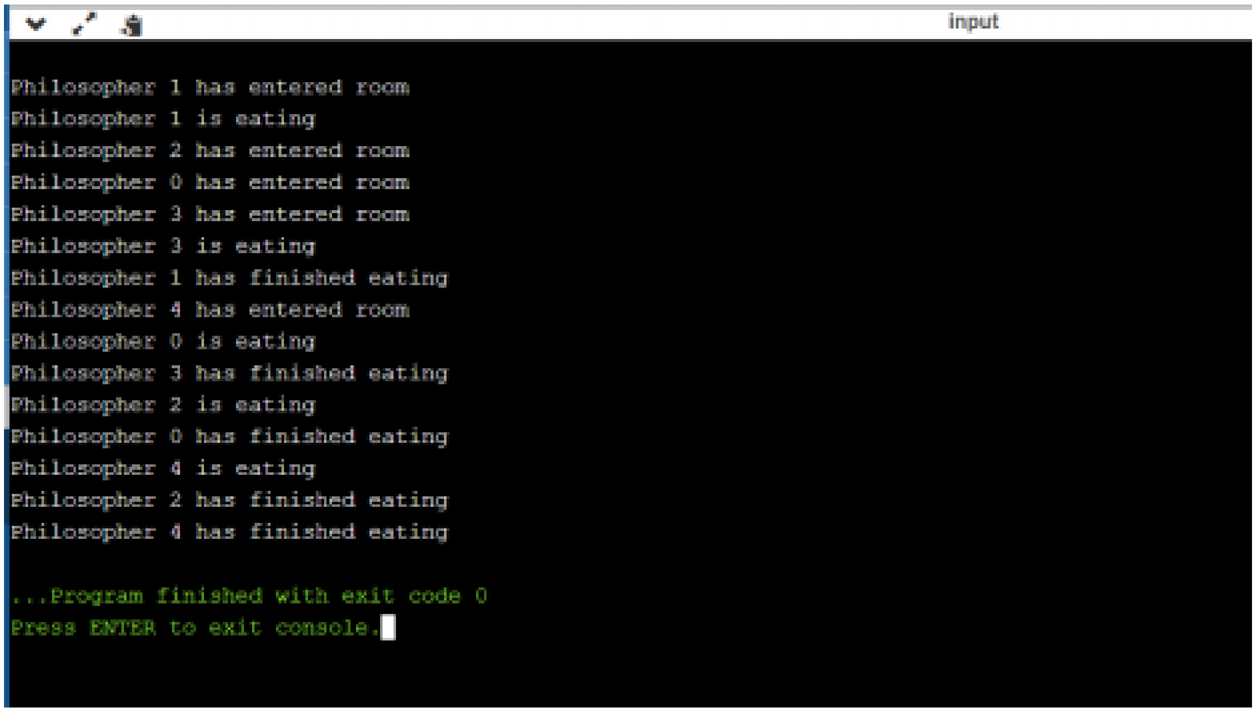
Objective: WAP to implement the Dining Philosopher's Problem

Theory: The Dining Philosophers problem is a classic synchronization and concurrency challenge in computer science, depicting a scenario where five philosophers sit around a dining table, and each must alternately think and eat. The philosophers share a finite number of chopsticks, with one placed between each pair of adjacent philosophers. To avoid deadlock and contention, a philosopher can only eat if they can successfully acquire both adjacent chopsticks. The problem illustrates the need for proper synchronization mechanisms, such as semaphores or mutexes, to ensure that philosophers can access chopsticks without conflicting and that they release resources after use, preventing deadlock and enabling fair access to the dining table. Various solutions exist, ranging from assigning a unique order for philosophers to pick up chopsticks to more sophisticated algorithms that prevent deadlock and promote efficient resource utilization.

Program :

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

sem_t room;
sem_t chopstick [5];

void * philosopher (void *);
void eat(int);
int main()
{
        int i,a[5];

        pthread_t tid[5];


        sem_init(&room,0,4);


        for(i=0;i<5;i++)

                sem_init(&chopstick[i],0,1);


        for(i=0;i<5;i++){

                a[i]=i;

                pthread_create(&tid[i],NULL,philosopher,(void *)&a[i]);

        }
        for(i=0;i<5;i++)

                pthread_join(tid[i],NULL);

}


void * philosopher(void * num)

{

        int phil=*(int *)num;


        sem_wait(&room);

        printf("\nPhilosopher %d has entered

        room",phil); sem_wait(&chopstick[phil]);

        sem_wait(&chopstick[(phil+1)%5]);


        eat(phil);

        sleep(2);

        printf("\nPhilosopher %d has finished eating",phil);
        sem_post(&chopstick[(phil+1)%5]);

        sem_post(&chopstick[phil]);

        sem_post(&room);

}


void eat(int phil)

{

        printf("\nPhilosopher %d is eating",phil);

}
```

Output :



```
Philosopher 1 has entered room
Philosopher 1 is eating
Philosopher 2 has entered room
Philosopher 0 has entered room
Philosopher 3 has entered room
Philosopher 3 is eating
Philosopher 1 has finished eating
Philosopher 4 has entered room
Philosopher 0 is eating
Philosopher 3 has finished eating
Philosopher 2 is eating
Philosopher 0 has finished eating
Philosopher 4 is eating
Philosopher 2 has finished eating
Philosopher 4 has finished eating

...Program finished with exit code 0
Press ENTER to exit console.
```

Sample Questions:

1. What is the Dining Philosophers problem, and why is it considered a classic synchronization challenge in concurrent computing?
2. Explain the concept of deadlock in the context of the Dining Philosophers problem. How does the scenario of philosophers holding chopsticks lead to potential deadlock, and what measures are taken to prevent it?

Practical – 7

Objective: WAP to implement the Banker's Algorithm

Theory: The Banker's algorithm is a resource allocation and deadlock avoidance algorithm used in operating systems. It is designed to prevent deadlock by determining whether granting a resource request would lead to an unsafe state. The algorithm works by maintaining information about the maximum demand of each process, the currently allocated resources, and the available resources in the system. Before granting a resource request, the Banker's algorithm simulates the allocation and checks if it would still be possible to satisfy the maximum demands of all processes. If the system remains in a safe state, the resource request is granted; otherwise, it is denied. This proactive approach helps ensure that resources are allocated in a way that avoids potential deadlocks, contributing to the overall stability and reliability of the system.

Program:

```
#include <stdio.h>
int current[5][5], maximum_claim[5][5], available[5];

int allocation[5] = {0, 0, 0, 0, 0};

int maxres[5], running[5], safe = 0;

int counter = 0, i, j, exec, resources, processes, k = 1;


int main()

{

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

 scanf("%d", &processes);


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

        {

 running[i] = 1;

 counter++;

 }
 printf("\nEnter number of resources: ");
 scanf("%d", &resources);

 printf("\nEnter Claim Vector:");
 for (i = 0; i < resources; i++)
            {
             scanf("%d", &maxres[i]);
}

 printf("\nEnter Allocated Resource Table:\n");
for (i = 0; i < processes; i++)
            {
             for(j = 0; j < resources; j++)
                        {
 scanf("%d", &current[i][j]);  }
 }

 printf("\nEnter Maximum Claim Table:\n");
for (i = 0; i < processes; i++)
            {
 for(j = 0; j < resources; j++)
                        {
 scanf("%d", &maximum_claim[i][j]);  }
 }

           printf("\nThe Claim Vector is: ");
 for (i = 0; i < resources; i++)
```

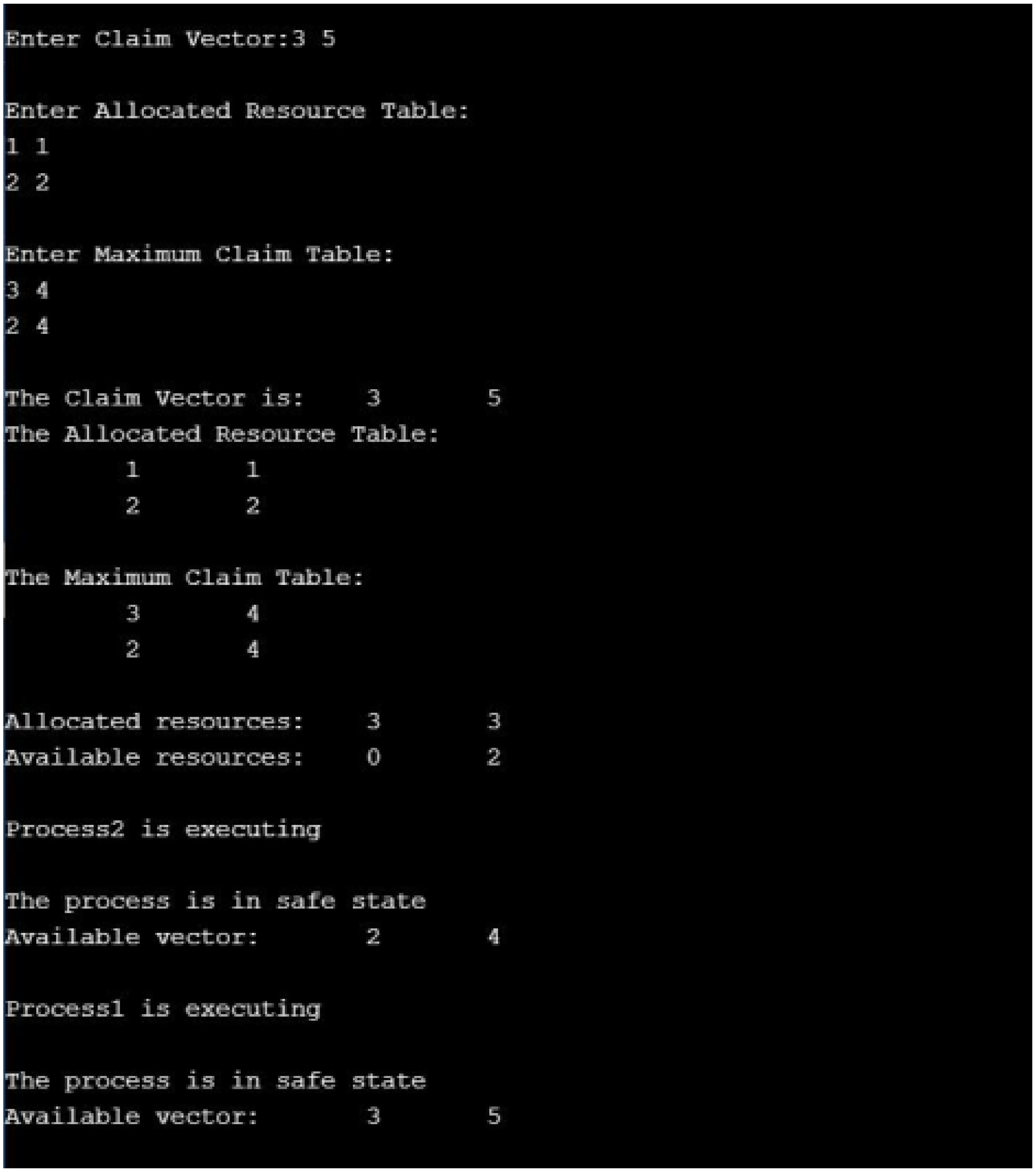
```
            {
             printf("\t%d", maxres[i]);
            }

printf("\nThe Allocated Resource Table:\n");


for (i = 0; i < processes; i++)
            {
             for (j = 0; j < resources; j++)
                        {
printf("\t%d", current[i][j]);
}
                        printf("\n");
}

printf("\nThe Maximum Claim Table:\n");
for (i = 0; i < processes; i++)
            {
for (j = 0; j < resources; j++)
                        {
                         printf("\t%d", maximum_claim[i][j]);
}
printf("\n");
}

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

```
 for (j = 0; j < resources; j++)
                        {
 allocation[j] += current[i][j];  }
 }
printf("\nAllocated resources:");  for
(i = 0; i < resources; i++)
            {
 printf("\t%d", allocation[i]);
 }

 for (i = 0; i < resources; i++)
            {
             available[i] = maxres[i] - allocation[i];
            }

 printf("\nAvailable resources:");  for
(i = 0; i < resources; i++)
            {
 printf("\t%d", available[i]);
 }
 printf("\n");

 while (counter != 0)
            {
 safe = 0;
 for (i = 0; i < processes; i++)
                        {
```

```
if (running[i])
                                {
exec = 1;
for (j = 0; j < resources; j++)
                                          {
                if (maximum_claim[i][j] - current[i][j] > available[j])  {
exec = 0;
break;
}
}
if (exec)
                                          {
printf("\nProcess%d is executing\n", i + 1);  running[i] = 0;
counter--;
safe = 1;

for (j = 0; j < resources; j++)
                                                     {
available[j] += current[i][j];
}
                                break;
}
}
}
if (!safe)
                     {
printf("\nThe processes are in unsafe state.\n");
```

```
 break;
 }


                         else
                         {
 printf("\nThe process is in safe state");
printf("\nAvailable vector:");


          for (i = 0; i < resources; i++)  {
 printf("\t%d", available[i]);  }


                          printf("\n");
 }
 }
 return 0;
}
```

Output:



```
Enter Claim Vector:3 5

Enter Allocated Resource Table:
1 1
2 2

Enter Maximum Claim Table:
3 4
2 4

The Claim Vector is:    3       5
The Allocated Resource Table:
        1       1
        2       2

The Maximum Claim Table:
        3       4
        2       4

Allocated resources:    3       3
Available resources:    0       2

Process2 is executing

The process is in safe state
Available vector:       2       4

Process1 is executing

The process is in safe state
Available vector:       3       5
```

Sample Questions:

1. Explain the key data structures used in the Banker's algorithm. How does it keep track of the maximum demand, currently allocated resources, and available resources for each process?
2. Walk through the steps of the Banker's algorithm when a process requests additional resources. How does the algorithm determine whether granting the request will lead to a safe state?

Practical – 8

Objective: WAP to implement the concept of First Fit memory allocation and Best fit memory allocation.

Theory: The first fit and best fit algorithms are memory allocation techniques used in operating systems for managing the allocation of memory blocks. In the first fit algorithm, when a process requests memory, the system allocates the first available block of memory that is large enough to accommodate the process's requirements. This method is simple and efficient but can lead to fragmentation. On the other hand, the best fit algorithm allocates the smallest available block of memory that meets the process's needs, minimizing fragmentation. While best fit can potentially result in more efficient memory utilization, it tends to require more extensive searching, impacting performance. Both algorithms aim to optimize memory usage by efficiently allocating and managing available memory blocks based on the specific needs of processes in the system.

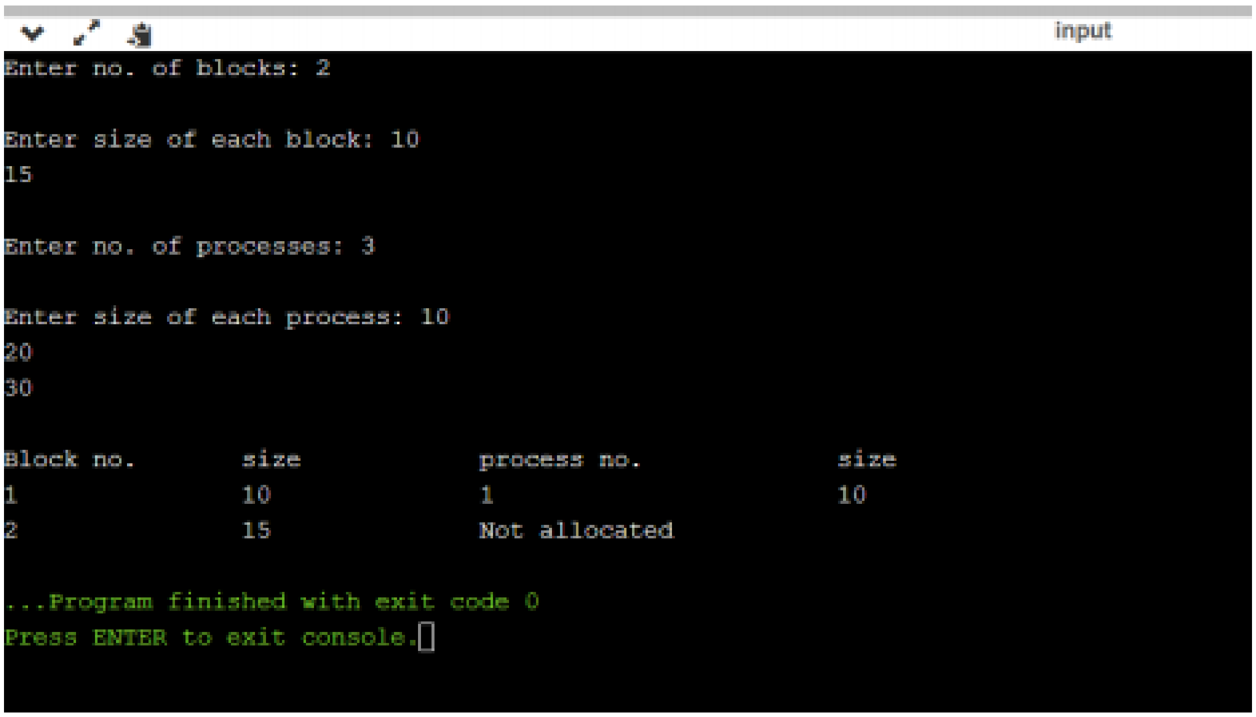
Program of First Fit Algorithm:

```
#include<iostream>

using namespace std;


int main()

{

        int bsize[10], psize[10], bno, pno, flags[10], allocation[10], i, j;


        for(i = 0; i < 10; i++)

        {

                flags[i] = 0;

                allocation[i] = -1;

        }


        cout<<"Enter no. of blocks: ";

        cin>>bno;
```

```
cout<<"\nEnter size of each block: ";

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

        cin>>bsize[i];


cout<<"\nEnter no. of processes: ";

cin>>pno;


cout<<"\nEnter size of each process: ";
for(i = 0; i < pno; i++)

        cin>>psize[i];

for(i = 0; i < pno; i++) //allocation as per first fit

        for(j = 0; j < bno; j++)

                if(flags[j] == 0 && bsize[j] >= psize[i])

                {

                        allocation[j] = i;

                        flags[j] = 1;

                        break;

                }


//display allocation details

cout<<"\nBlock no.\tsize\t\tprocess no.\t\tsize";

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

{

        cout<<"\n"<< i+1<<"\t\t"<<bsize[i]<<"\t\t";

        if(flags[i] == 1)

                cout<<allocation[i]+1<<"\t\t\t"<<psize[allocation[i]];

        else

                cout<<"Not allocated";
```

```
            }


            return 0;
}
```

Output of First Fit Algorithm:



```
Enter no. of blocks: 2

Enter size of each block: 10
15

Enter no. of processes: 3

Enter size of each process: 10
20
30

Block no.       size            process no.             size
1               10              1                       10
2               15              Not allocated

...Program finished with exit code 0
Press ENTER to exit console.
```

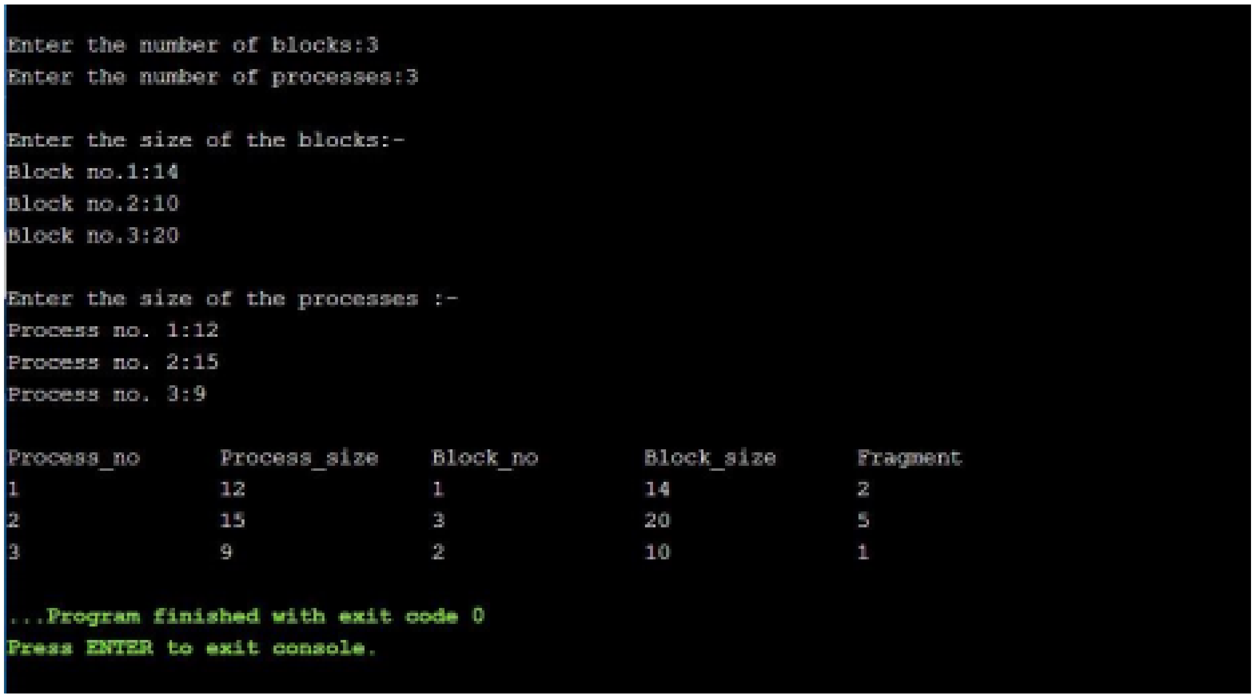
Program of Best fit memory allocation:

```
#include<iostream>

using namespace std;


int main()
{
            int fragment[20],b[20],p[20],i,j,nb,np,temp,lowest=9999;

            static int barray[20],parray[20];
```

```
        cout<<"\nEnter the number of blocks:";
        cin>>nb;
        cout<<"Enter the number of processes:";
        cin>>np;

        cout<<"\nEnter the size of the blocks:-\n";
        for(i=1;i<=nb;i++)
 {
 cout<<"Block no."<<i<<":";
 cin>>b[i];
 }

        cout<<"\nEnter the size of the processes :-\n";
        for(i=1;i<=np;i++)
 {
 cout<<"Process no. "<<i<<":";
 cin>>p[i];
}

        for(i=1;i<=np;i++)
        {
                for(j=1;j<=nb;j++)
                {
                        if(barray[j]!=1)
                        {
                                temp=b[j]-p[i];
                                if(temp>=0)
                                        if(lowest>temp)
```

```
                    {
                        parray[i]=j;
                        lowest=temp;
                    }
              }
          }

          fragment[i]=lowest;
          barray[parray[i]]=1;
          lowest=10000;
      }

      cout<<"\nProcess_no\tProcess_size\tBlock_no\tBlock_size\tFragment";
      for(i=1;i<=np && parray[i]!=0;i++)

  cout<<"\n"<<i<<"\t\t"<<p[i]<<"\t\t"<<parray[i]<<"\t\t"<<b[parray[i]]<<"\t\t"<<fragment[i]
                                                                                     ;
  return 0;
  }
```

Output of Best Fit Algorithm:



```
Enter the number of blocks:3
Enter the number of processes:3

Enter the size of the blocks:-
Block no.1:14
Block no.2:10
Block no.3:20

Enter the size of the processes :-
Process no. 1:12
Process no. 2:15
Process no. 3:9

Process_no      Process_size    Block_no        Block_size      Fragment
1               12              1               14              2
2               15              3               20              5
3               9               2               10              1

...Program finished with exit code 0
Press ENTER to exit console.
```

Sample Questions:

1. What is the primary objective of memory allocation algorithms, and how do First Fit and Best Fit contribute to achieving this goal?
2. Explain the key idea behind the First Fit algorithm. How does it allocate memory to a process when a request is made, and what are its advantages and drawbacks?
3. Describe the Best Fit memory allocation algorithm. How does it differ from First Fit, and what are the considerations when selecting the smallest available block for a process?

Practical – 9

Objective: WAP to implement the concept of Worst Fit memory allocation.

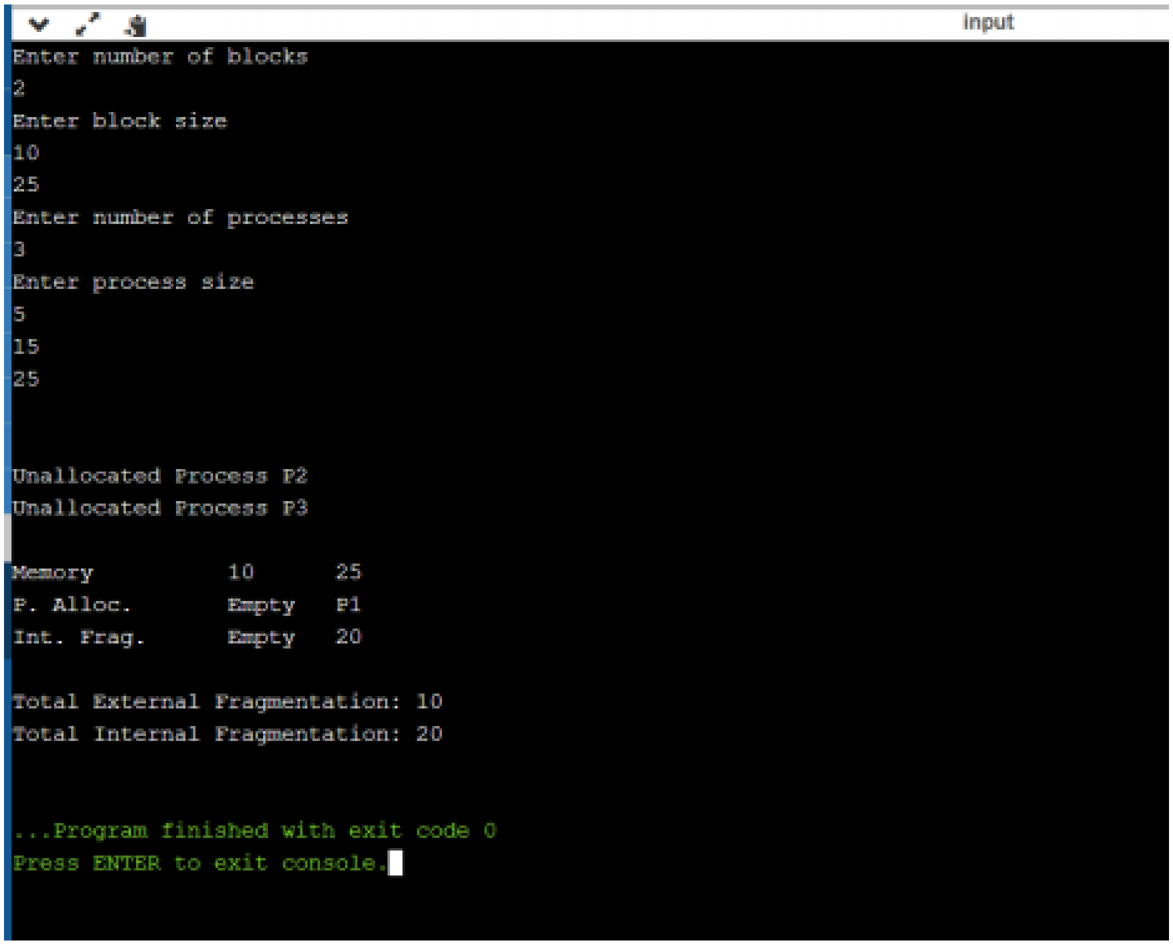
Theory: The Worst Fit algorithm is a memory allocation strategy used in operating systems to manage the assignment of memory blocks to processes. In this approach, when a process requests memory, the system allocates the largest available block of memory, aiming to minimize the remaining free space. This strategy is designed to reduce internal fragmentation by allocating processes into the largest possible blocks. However, the Worst Fit algorithm can result in inefficient use of memory over time, as it may leave behind smaller, less usable memory segments. Despite its simplicity, Worst Fit is generally considered less effective than other allocation algorithms like First Fit or Best Fit, which often lead to better overall memory utilization and fragmentation management.

Program:

```
#include<iostream>
#include<algorithm>
using namespace std;
struct node{
 int memsize;
 int allocp=-1;
 int pos;
 int allocSize;
 }m[200];
bool posSort(node a,node b){
 return a.pos < b.pos;
 }
bool memSort(node a,node b){
 return a.memsize > b.memsize;
 }
int main(){
 int nm,np,choice, i, j, p[200];
 cout<<"Enter number of blocks\n";
 cin>>nm;
 cout<<"Enter block size\n";
 for(i=0;i<nm;i++){
 cin>>m[i].memsize;
 m[i].pos=i;
 }
 cout<<"Enter number of processes\n";
 cin>>np;

 cout<<"Enter process size\n";
 for(i=0;i<np;i++){
 cin>>p[i];
 }
 cout<<"\n\n";
 sort(m,m+nm,memSort);
 int globalFlag=0;
 for(i=0;i<np;i++){
 int flag=0;
 for(j=0;j<nm;j++){
 if(p[i]<=m[j].memsize && m[j].allocp==-1){
m[j].allocp=i;
 m[j].allocSize=p[i];
 flag=1;
 break;
 }
 }
 if(flag==0){

 cout<<"Unallocated Process P"<<i+1<<"\n";

globalFlag=1;

 }

 }

 sort(m,m+nm,posSort);

 cout<<"\n";

 int intFrag=0,extFrag=0;

 cout<<"Memory\t\t";

 for(i=0;i<nm;i++){
cout<<m[i].memsize<<"\t";

 }

 cout<<"\n";

 cout<<"P. Alloc.\t";

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

 if(m[i].allocp!=-1){

 cout<<"P"<<m[i].allocp+1<<"\t";  }

 else{

 cout<<"Empty\t";

 }

 }

 cout<<"\n";

 cout<<"Int. Frag.\t";

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

 if(m[i].allocp!=-1){

 cout<<m[i].memsize-m[i].allocSize<<"\t";

intFrag+=m[i].memsize-m[i].allocSize;  }
```

```
 else{
 extFrag+=m[i].memsize;
 cout<<"Empty\t";
 }
 }
 cout<<"\n";
 cout<<"\n";
 if(globalFlag==1)
cout<<"Total External Fragmentation: "<<extFrag<<"\n";
 else{
 cout<<"Available Memory: "<<extFrag<<"\n";
 }
 cout<<"Total Internal Fragmentation: "<<intFrag<<"\n";
 return 0;
}
```

Output:



```
Enter number of blocks
2
Enter block size
10
25
Enter number of processes
3
Enter process size
5
15
25


Unallocated Process P2
Unallocated Process P3

Memory          10      25
P. Alloc.       Empty   P1
Int. Frag.      Empty   20

Total External Fragmentation: 10
Total Internal Fragmentation: 20


...Program finished with exit code 0
Press ENTER to exit console.
```

Sample Questions:

1. Discuss the concept of internal fragmentation in the context of the Worst Fit algorithm. How does it aim to minimize internal fragmentation, and what are the potential drawbacks associated with this strategy?
2. Compare the Worst Fit algorithm with other memory allocation algorithms in terms of efficiency and impact on system performance. In what scenarios might Worst Fit be preferred, and when could it be less suitable?
3. How does the Worst Fit algorithm contribute to memory utilization and fragmentation management in an operating system? Can you highlight any trade-offs or challenges associated with using this algorithm in practical applications?

Practical – 10

Objective: WAP a program to implement FIFO Page Replacement algorithm.

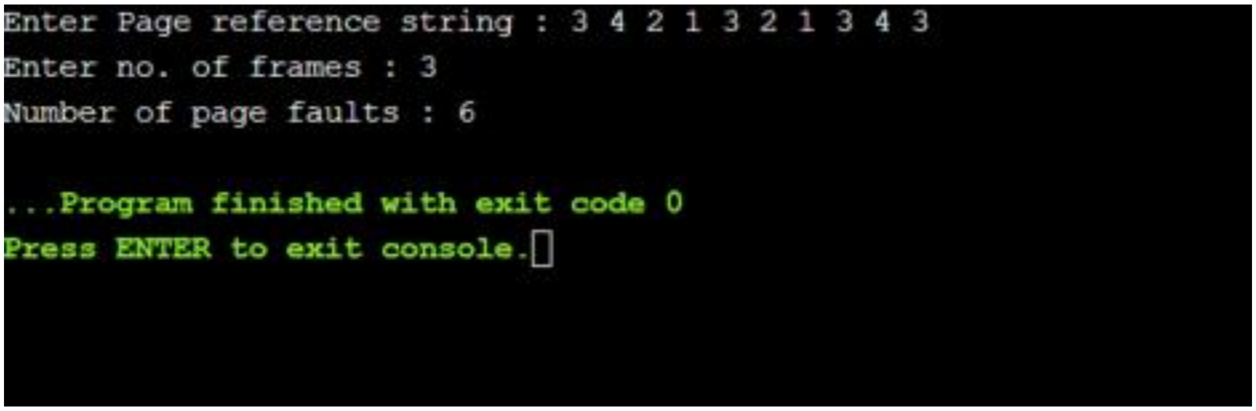
Theory: First-In-First-Out (FIFO) is a page replacement algorithm employed in operating systems to manage memory. In this scheme, the page that has been in the memory the longest is the first to be replaced when a new page needs to be loaded. FIFO operates like a queue, where pages are brought into memory in a sequential manner, and the oldest page is removed when the memory is full. While easy to implement, FIFO may suffer from the "Belady's Anomaly," where increasing the number of frames does not guarantee a reduction in page faults, and it may not always provide optimal performance compared to more advanced page replacement algorithms.

Program :

```
#include<iostream>
using namespace std;
class fifo_alg
{
 char ref[100];
 int frame,fault,front,rear;
 char *cir_que;
public:
 fifo_alg()
 {
 front=rear=-1;
 fault=0;
 }
 void getdata();
 void page_fault();
 };
```

```
void fifo_alg::getdata()
{
 cout<<"Enter Page reference string : ";
  cin.getline(ref,50);
 cout<<"Enter no. of frames : ";
 cin>>frame;
}
void fifo_alg::page_fault()
{
 cir_que=new char[frame];
int flag=0;
 for(int i=0;ref[i]!=0;i++)
 {
 if(ref[i]==' ')
 continue;
 if(front==-1)
 {
 cir_que[0]=ref[i];
 front=rear=0;
 fault++;
 }
 else
 {
 for(int y=front%frame;y!=rear;y=(y+1)%frame)
if(cir_que[y]==ref[i])
 {
 flag=1;
 break;
 }
 if(cir_que[rear]==ref[i])
 flag=1;
 if(flag==0)
 {
 if((rear+1)%frame==front)
 front=(front+1)%frame;
 rear=(rear+1)%frame;
 cir_que[rear]=ref[i];
fault++;
 }
 flag=0;
 }
 }
 cout<<"Number of page faults : "<<fault;
}
int main()
{
 fifo_alg page;
 page.getdata();
 page.page_fault();
 return 0;
}
```

Output:



```
Enter Page reference string : 3 4 2 1 3 2 1 3 4 3
Enter no. of frames : 3
Number of page faults : 6

...Program finished with exit code 0
Press ENTER to exit console.
```

Sample Questions:

1. Explain the fundamental concept of the FIFO (First-In-First-Out) page replacement algorithm. How does it determine which page to replace when a page fault occurs, and what is the underlying principle behind its operation?
2. Discuss the potential drawback of FIFO known as "Belady's Anomaly." What is this anomaly, and how does it impact the performance of the FIFO algorithm in certain scenarios?
3. Compare FIFO with other page replacement algorithms, such as LRU (Least Recently Used) or Optimal. What are the advantages and limitations of FIFO in terms of simplicity, implementation, and overall efficiency, especially in the context of varying workload patterns?

Practical – 11

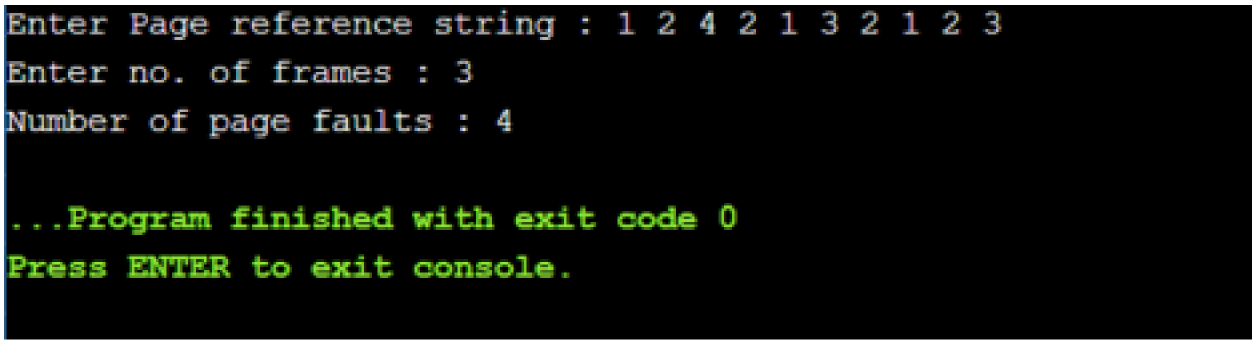
Objective: WAP a program to implement LRU Page Replacement algorithms

Theory: The Least Recently Used (LRU) page replacement algorithm is a method employed in operating systems to manage memory and determine which page to replace when a page fault occurs. LRU operates on the principle of discarding the page that has not been accessed for the longest period. It maintains a record of the order in which pages are accessed and, when a new page needs to be loaded into memory, it selects the page that has not been accessed for the longest time. LRU aims to minimize page faults by retaining pages that are most likely to be referenced in the near future. However, its practical implementation may involve additional complexity, such as maintaining access timestamps or using special data structures like counters or stacks to track page access history.

Program:

```
#include<iostream>
using namespace std;

struct node

{

 char data;

 node *next;

};

class lru_alg

{

 char ref[100];

 int frame,count,fault;

 node *front,*rear;

public:

 lru_alg()

 {

 front=rear=NULL;

 fault=count=0;

 }
 void getdata();
 void page_fault();
 };
void lru_alg::getdata()
{
 cout<<"Enter Page reference string : ";
 cin.getline(ref,50);


 cout<<"Enter no. of frames :
";  cin>>frame;
 }
void lru_alg::page_fault()
{
 int flag=0;
 for(int i=0;ref[i]!=0;i++)
 {
 if(ref[i]==' ')
 continue;
 if(front==NULL)
 {
 front=rear=new node;
 front->data=ref[i];
 front->next=NULL;
 fault=count=1;
 }
 else
 {
 node *temp=front,*prev=NULL;
while(temp!=NULL)
 {
 if(temp->data==ref[i])  {
 flag=1;
 if(prev==NULL)  {
rear=rear->next=front;  front=front-
>next;  rear->next=NULL;  }
 else if(temp!=rear)  {
 prev->next=temp->next;  rear=rear-
>next=temp;  temp->next=NULL;  }
 break;
 }
 prev=temp;
 temp=temp->next;
 }
 if(flag==0)
 {
 if(count==frame)
 {
 rear=rear->next=front;
front=front->next;
 rear->data=ref[i];
 rear->next=NULL;  }
 else
 {
 rear=rear->next=new node;
rear->data=ref[i];
 rear->next=NULL;
 count++;
 }
 fault++;
 }
 flag=0;
 }
 }
 cout<<"Number of page faults : "<<fault;
}
int main()
{
 lru_alg page;
 page.getdata();
 page.page_fault();
 return 0;
}
```

Output:



```
Enter Page reference string : 1 2 4 2 1 3 2 1 2 3
Enter no. of frames : 3
Number of page faults : 4

...Program finished with exit code 0
Press ENTER to exit console.
```

Sample Questions:

1. Can you explain the basic idea behind the Least Recently Used (LRU) page replacement algorithm and how it decides which page to replace when a page fault occurs?

Practical – 12

Objective: WAP a program to implement Optimal Page Replacement algorithm.

Theory: The Optimal page replacement algorithm is a theoretical approach to managing memory in operating systems. It aims to minimize page faults by selecting the page that will not be used for the longest period in the future. In this algorithm, when a page fault occurs and a new page needs to be brought into memory, the Optimal algorithm identifies the page that will have the farthest next reference in the execution of the program and replaces it. While Optimal offers the best possible page replacement strategy in terms of minimizing page faults, it is impractical for real-world implementation because it requires knowledge of future page accesses, which is generally not available. Despite its theoretical optimality, more practical algorithms like FIFO, LRU, or clock-based approaches are often used in real operating systems.

Program:

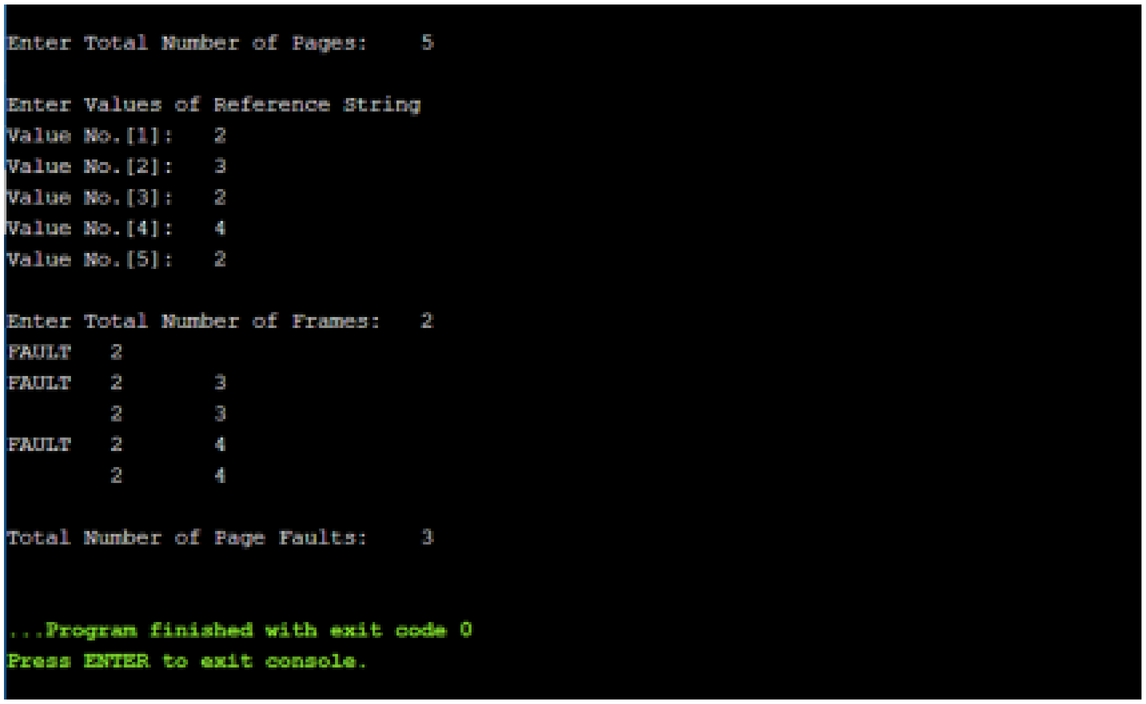
```
#include<stdio.h>
int main()
{
 int reference_string[25], frames[25], interval[25];
 int pages, total_frames, page_faults = 0;
 int m, n, temp, flag, found;
 int position, maximum_interval, previous_frame = -1;
 printf("\nEnter Total Number of Pages:\t");
 scanf("%d", &pages);
 printf("\nEnter Values of Reference String\n");
 for(m = 0; m < pages; m++)
 {
 printf("Value No.[%d]:\t", m + 1);
 scanf("%d", &reference_string[m]);
 }
 printf("\nEnter Total Number of Frames:\t");
 scanf("%d", &total_frames);
   for(m = 0; m < total_frames; m++)
```

```
{
frames[m] = -1;
}
for(m = 0; m < pages; m++)
{
flag = 0;

for(n = 0; n < total_frames; n++)
{
if(frames[n] == reference_string[m])  {
flag = 1;
printf("\t");
break;
}
}
if(flag == 0)
{
if (previous_frame == total_frames - 1)  {
for(n = 0; n < total_frames; n++)  {
for(temp = m + 1; temp < pages; temp++)  {
interval[n] = 0;
if (frames[n] == reference_string[temp])  {
interval[n] = temp - m;  break;
}
}
}
```

```
 found = 0;

 for(n = 0; n < total_frames; n++)
{

 if(interval[n] == 0)

 {

 position = n;

 found = 1;

 break;

 }

 }

 }

 else

 {

 position = ++previous_frame;  found =

1;

 }

 if(found == 0)

 {

 maximum_interval = interval[0];  position

= 0;

 for(n = 1; n < total_frames; n++)  {

 if(maximum_interval < interval[n])  {

 maximum_interval = interval[n];  position = n;

 }

 }

 }
frames[position] = reference_string[m];

 printf("FAULT\t");
```

```
page_faults++;
}
for(n = 0; n < total_frames; n++)
{
if(frames[n] != -1)
{
printf("%d\t", frames[n]);
}
}
printf("\n");
}
printf("\nTotal Number of Page Faults:\t%d\n", page_faults);
return 0;
}
```

Output:



```
Enter Total Number of Pages:    5

Enter Values of Reference String
Value No.[1]:   2
Value No.[2]:   3
Value No.[3]:   2
Value No.[4]:   4
Value No.[5]:   2

Enter Total Number of Frames:   2
FAULT   2
FAULT   2       3
        2       3
FAULT   2       4
        2       4

Total Number of Page Faults:    3


...Program finished with exit code 0
Press ENTER to exit console.
```

Sample Questions:

1.Can you explain the basic idea behind the Optimal page replacement algorithm ?

Practical – 13

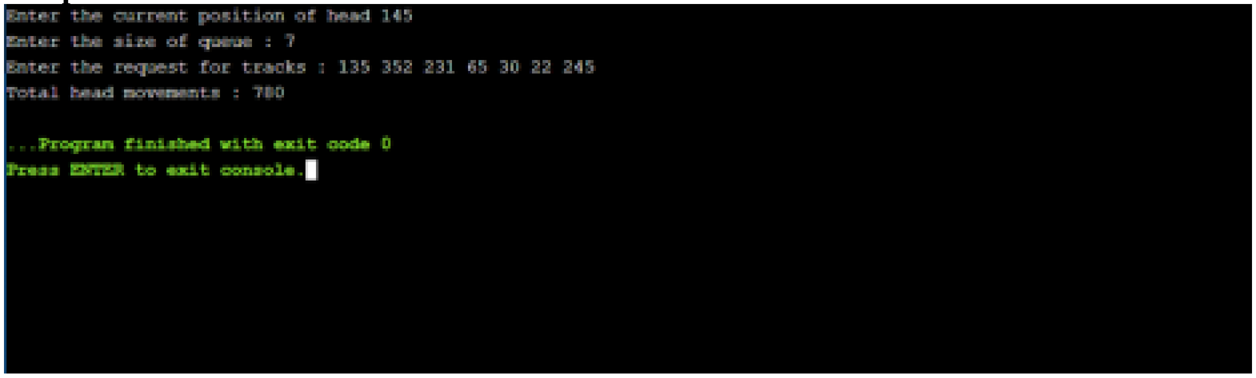
Objective: WAP to implement FCFS Disk Scheduling

Theory: First-Come-First-Serve (FCFS) is a disk scheduling algorithm that determines the order in which pending disk I/O requests are serviced. In FCFS, the requests are serviced in the order they arrive, forming a queue. The first request that arrives is the first to be executed, followed by subsequent requests in the same order. While FCFS is simple to implement and ensures fairness by processing requests in the order they are received, it may lead to inefficiencies, such as the "head of the disk" constantly moving back and forth, causing increased seek times and reduced overall disk performance. This can result in suboptimal disk scheduling in terms of minimizing access time and improving throughput compared to more sophisticated scheduling algorithms.

Program:

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

using namespace std;

class fcfs_disk

{

 int ref[100];

 int pos,size;

public:

 void getdata();

 void total_move();

 };
void fcfs_disk::getdata()

{

 cout<<"Enter the current position of head

";  cin>>pos;

 cout<<"Enter the size of queue :

";  cin>>size;

 cout<<"Enter the request for tracks :

";  for(int i=0;i<size;i++)

 cin>>ref[i];
}
void fcfs_disk::total_move()
{
 int num=pos,move=0;
 for(int i=0;i<size;i++)
 {
 move+=abs(num-ref[i]);
 num=ref[i];
 }
 cout<<"Total head movements :
"<<move; }
int main()
{
 fcfs_disk fcfs;
 fcfs.getdata();
 fcfs.total_move();
 return 0;
}
```

Output:



```
Enter the current position of head 145
Enter the size of queue : 7
Enter the request for tracks : 135 352 231 65 30 22 245
Total head movements : 780

...Program finished with exit code 0
Press ENTER to exit console.
```

Sample Questions:

1. Explain the working principle of the First-Come-First-Serve (FCFS) disk scheduling algorithm?

Practical – 14

Objective: WAP to implement SCAN and C SCAN Disk Scheduling

Theory: The SCAN (Elevator) and C-SCAN (Circular SCAN) are disk scheduling algorithms used to enhance the efficiency of accessing data on a disk. SCAN operates by servicing requests in one direction until the end of the disk is reached and then reversing direction. This back-and-forth movement minimizes the total seek time and avoids starvation of requests towards the ends of the disk. C-SCAN, on the other hand, functions similarly but restricts its servicing to one direction only, servicing requests in a circular manner. Both algorithms are designed to reduce the average seek time compared to simpler algorithms like FCFS, particularly in scenarios where requests are distributed across the disk, promoting more balanced and efficient disk access.

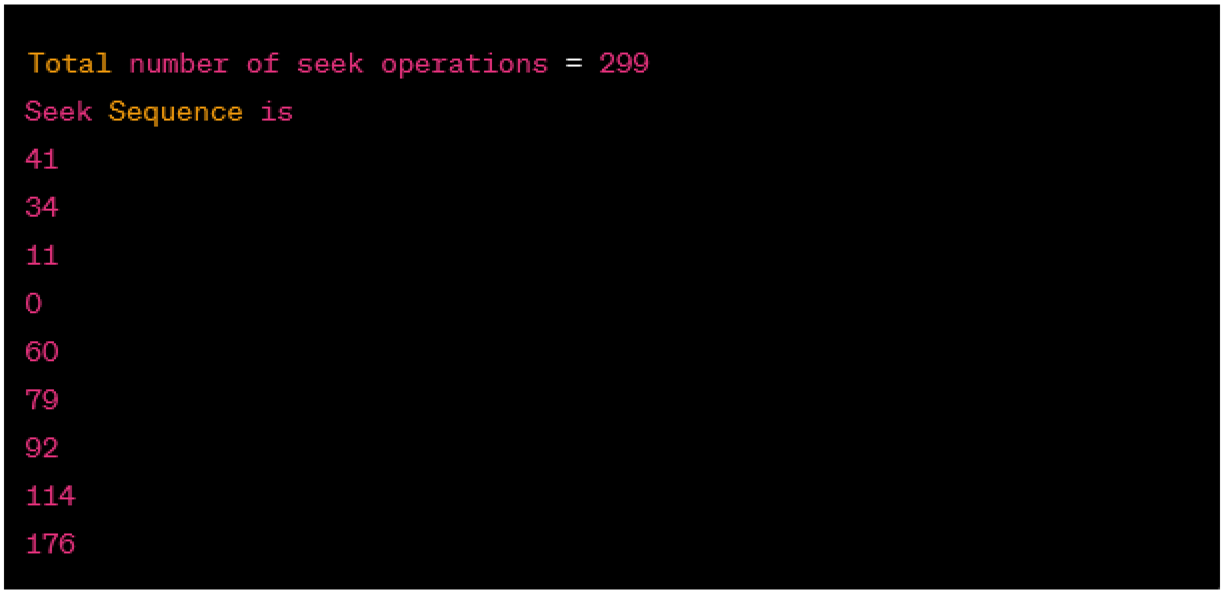
Program of SCAN Disk Scheduling:

```
#include <bits/stdc++.h>

using namespace std;

int size = 8;

int disk_size = 200;

void SCAN(int arr[], int head, string direction)

{

        int seek_count = 0;

        int distance, cur_track;

        vector<int> left, right;

        vector<int> seek_sequence;

        if (direction == "left")

                left.push_back(0);
        else if (direction == "right")

                right.push_back(disk_size - 1);

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

                if (arr[i] < head)

                        left.push_back(arr[i]);

                if (arr[i] > head)

                        right.push_back(arr[i]);

        }
```

```
std::sort(left.begin(), left.end());

std::sort(right.begin(), right.end());

int run = 2;

while (run--) {

        if (direction == "left") {

                for (int i = left.size() - 1; i >= 0; i--) {

                        cur_track = left[i];

                        seek_sequence.push_back(cur_track);

                        distance = abs(cur_track - head);

                        seek_count += distance;

                        head = cur_track;

                }

                direction = "right";

        }

        else if (direction == "right") {

                for (int i = 0; i < right.size(); i++) {

                        cur_track = right[i];

                        seek_sequence.push_back(cur_track);

                        distance = abs(cur_track - head);
                        seek_count += distance;

                        head = cur_track;

                }

                direction = "left";

        }

}

      cout << "Total number of seek

  operations = " << seek_count << endl;

cout << "Seek Sequence is" << endl;
```

```
        for (int i = 0; i < seek_sequence.size();
                i++) { cout << seek_sequence[i]
                << endl;
        }
}

int main()
{
        int arr[size] = { 176, 79, 34, 60,92, 11, 41,
        114 }; int head = 50;
        string direction = "left";
        SCAN(arr, head, direction);
        return 0;
}
```

Output:



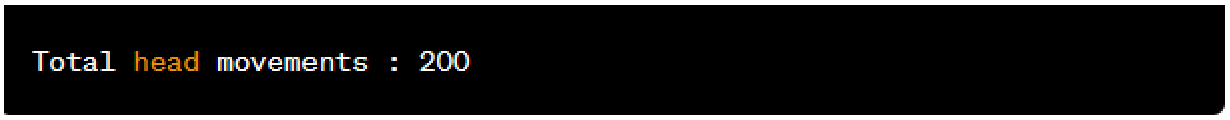
```
Total number of seek operations = 299
Seek Sequence is
41
34
11
0
60
79
92
114
176
```

Program of C-SCAN Disk Scheduling:

```
#include<iostream>

#include<stdlib.h>

using namespace std;

class cscan_disk

{

 int ref[100];

 int ttrk,cur,size,prev;
int sort();

public:

 void getdata();

 void total_move();

 };

int cscan_disk::sort()

{

 int temp;

 for(int i=0;i<size-1;i++)

 for(int y=0;y<size-1;y++)

 if(ref[y]>ref[y+1])

 {

 temp=ref[y];

 ref[y]=ref[y+1];

 ref[y+1]=temp;

 }

 for(int i=0;i<size;i++)

 if(ref[i]>cur)

 return i;
 return size;

 }

void cscan_disk::getdata()

{

 cout<<"Enter total number of tracks :

";  cin>>ttrk;

 ttrk--;

 cout<<"Enter the current position of head : ";
cin>>cur;

 cout<<"Enter previous position of head :

";  cin>>prev;

 cout<<"Enter the size of queue :

";  cin>>size;

 cout<<"Enter the request for tracks :

";  for(int i=0;i<size;i++)

 cin>>ref[i];

 }

void cscan_disk::total_move()

{

 int num=cur,move=0,ind,dir=cur-

 prev;  ind=sort();

 if(dir>0)

 {

 for(int i=ind;i<size;i++)

 {

 move+=ref[i]-num;

 num=ref[i];
 }
 if(ind!=0)
 {
 if(ind==size)
 move+=ttrk-cur;
 else
 move+=ttrk-ref[size-1];
 num=0;

 for(int i=0;i<=ind-1;i++)
 {
 move+=ref[i]-num;
 num=ref[i];
 }
 }
 }
 else
 {
 for(int i=ind-1;i>=0;i--)
 {
  move+=num-ref[i];
 num=ref[i];
 }
 if(ind==0)
 move+=cur+ref[0];
 else if(ind!=size)
 move+=ref[0];
```

```
 num=ttrk;
 for(int i=size-1;i>=ind;i--)
 {
 move+=num-ref[i];
 num=ref[i];
 }
 }
 cout<<"Total head movements :
"<<move; }
int main()
{
 cscan_disk cscan;
 cscan.getdata();
 cscan.total_move();
 return 0;
}
```

Output :



```
Total head movements : 200
```

Sample Questions:

1. Explain the significance of the "left" and "right" directions in the SCAN algorithm.
2. What issue does the SCAN algorithm aim to address in disk scheduling?
3. How does the SCAN algorithm determine the order in which it services disk requests?
4. What is the role of the seek_count variable in the SCAN algorithm, and how is it calculated?

Practical – 15

Objective: WAP to implement LOOK and C-LOOK Disk Scheduling

Theory: LOOK and C-LOOK are disk scheduling algorithms that aim to optimize the movement of the disk arm and reduce seek time when accessing data on a disk. LOOK moves the disk arm only as far as the last request in the current direction, reversing direction once it reaches the end of the disk. This prevents unnecessary movement to the disk's extreme positions, minimizing seek time. C-LOOK, an improvement over LOOK, operates in a similar fashion but scans only in one direction and ignores requests in the opposite direction, resulting in a more efficient and predictable disk arm movement. Both algorithms are designed to enhance disk access performance compared to simpler scheduling strategies like FCFS by considering the distribution of requests on the disk.

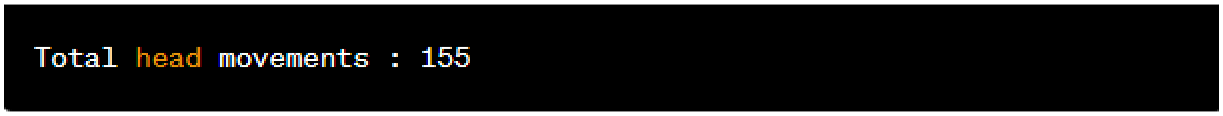
Program of LOOK Disk Scheduling:

```
#include<iostream>

#include<stdlib.h>

using namespace std;

class look_disk

{

 int ref[100];

 int ttrk,cur,size,prev;

 int sort();
public:

 void getdata();

 void total_move();

 };

int look_disk::sort()

{

 int temp;

 for(int i=0;i<size-1;i++)

 for(int y=0;y<size-1;y++)

 if(ref[y]>ref[y+1])

 {
```

```
 temp=ref[y+1];
 ref[y+1]=ref[y];
 ref[y]=temp;
 }
 for(int i=0;i<size;i++)
 if(ref[i]>cur)
 return i;
 return size;
 }
void look_disk::getdata()
{
 cout<<"Enter total number of tracks :
";  cin>>ttrk;
 ttrk--;
 cout<<"Enter the current position of head :
";  cin>>cur;
cout<<"Enter previous position of head :
";  cin>>prev;
 cout<<"Enter the size of queue :
";  cin>>size;
 cout<<"Enter the request for tracks :
";  for(int i=0;i<size;i++)
 cin>>ref[i];
 }
void look_disk::total_move()
{
 int num=cur,move=0,ind,dir=cur-
```

```
prev;  ind=sort();
 if(dir>0)
 {
 for(int i=ind;i<size;i++)
 {
 move+=ref[i]-num;
 num=ref[i];
 }
 if(ind!=0)
 {
 if(ind==size)
 move+=num-ref[ind-1];
 else
 move+=ref[size-1]-ref[ind-1];
num=ref[ind-1];
 for(int i=ind-2;i>=0;i--)
{
 move+=num-ref[i];
 num=ref[i];
 }
 }
 }
 else
 {
 for(int i=ind-1;i>=0;i--)
 {
 move+=num-ref[i];
 num=ref[i];

 }

 if(ind==0)

 move+=ref[ind]-num;

 else if(ind!=size)

 move+=ref[ind]-ref[0];

 num=ref[ind];

 for(int i=ind+1;i<size;i++)

 {

 move+=ref[i]-num;

 num=ref[i];

 }

 }

 cout<<"Total head movements :

"<<move; }

int main()
{

 look_disk look;

 look.getdata();

 look.total_move();

 return 0;

}
```

Output :



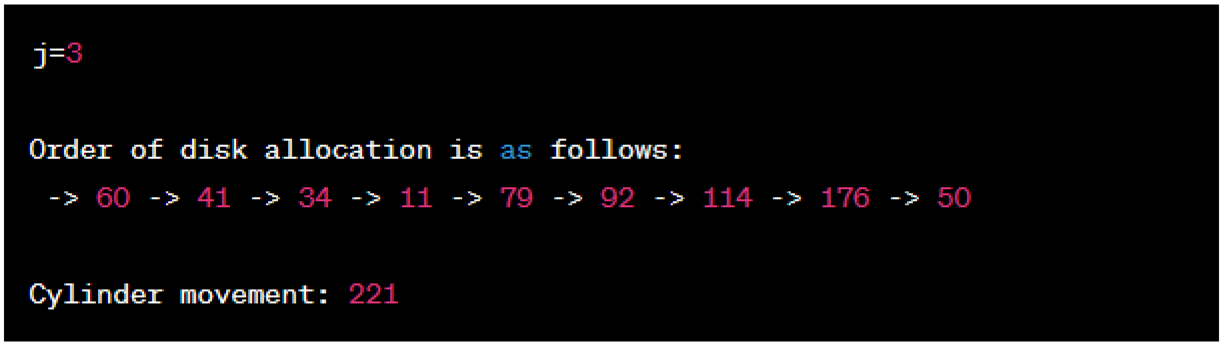
```
Total head movements : 155
```

Program of C-LOOK Disk Scheduling :

```
#include<stdio.h>

#include<stdlib.h>


int main()

{

 int n, i, j, head, item[20], dst[20];

 int cylinders=0;

 printf("Enter no. of locations:");
scanf("%d",&n);

 printf("Enter position of

head:");  scanf("%d",&head);

 printf("Enter elements of disk

queue:");  for(i=0;i<n;i++)

 {

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

 dst[i]=(head-item[i]);

 }

 //Selection Sort

 for(i=0;i<n-1;i++)

 {

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

 {

 if(dst[j]>dst[i])

 {

 int temp=dst[j];

 dst[j]=dst[i];
 dst[i]=temp;

 temp=item[i];
 item[i]=item[j];
 item[j]=temp;
 }
 }
 }



 for(i=0;i<n;i++)
 {
 if(item[i]>=head)
 {
 j=i;
 break;
 }
 }

 printf("j=%d", j);
 printf("\n\nOrder of disk allocation is as
follows:\n");  for(i=j;i<n;i++)
 {
 printf(" -> %d", item[i]);
 cylinders+= abs(head-item[i]);
 head=item[i];
```

```
}
for(i=0;i<j;i++)
{
printf(" -> %d", item[i]);
cylinders+= abs(head-item[i]);
head=item[i];


}


printf("\n\nCylinder movement: %d\n\n", cylinders );
}
```

Output:



```
j=3

Order of disk allocation is as follows:
 -> 60 -> 41 -> 34 -> 11 -> 79 -> 92 -> 114 -> 176 -> 50

Cylinder movement: 221
```

Sample Questions :

1. Can you explain how the SCAN disk scheduling algorithm is implemented in the provided C code?
2. Describe the key steps and the logic behind sorting the disk queue elements based on their distances from the disk head.