**EXPERIMENT-5**

**Aim:** Write a C program to simulate Bankers Algorithm for the purpose of Deadlock avoidance.

**Description:** The banker’s algorithm which is also known as avoidance algorithm is a deadlock detection algorithm. It was developed by Edsger Dijkstra. It is designed to check the safe state whenever a resource is requested. It takes analogy of bank, where customer request to withdraw cash. Based on some data the cash is lent to the customer. The banker can’t give more cash than what the customer has requested for, and the total available cash.  As this algorithm uses bank analogy so named as banker’s algorithm.

**Algorithm:**

Banker’s algorithm consist of Safety algorithm and Resource request algorithm.

**Safety Algorithm**

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

*1) Let Work and Finish be vectors of length ‘m’ and ‘n’ respectively.  
Initialize: Work = Available  
Finish[i] = false; for i=1, 2, 3, 4….n*

*2) Find an i such that both  
a) Finish[i] = false  
b) Needi <= Work if no such i exists goto step (4)*

*3) Work = Work + Allocation  
Finish[i] = true  
goto step (2)*

*4) if finish [i] = true for all i  
then the system is in a safe state*

**Resource-Request Algorithm**

Let Requesti be the request array for process Pi. Requesti[j] = k means process Pi wants k instances of resource type Rj. When a request for resources is made by process Pi, the following actions are taken:

*1)If Requesti<=Needi Goto step(2); otherwise, raise an error condition, since the process has exceeded its maximum claim.*

*2) If Requesti <= Available Goto step (3); otherwise, Pi must wait, since the resources are not available.*

*3) Have the system pretend to have allocated the requested resources to process Pi bymodifying the state as  
follows:  
Available=Available–Requesti  
Allocationi=Allocationi+Requesti  
Needi = Needi– Requesti*

**Code:**

#include<stdio.h>

#include<stdlib.h>

int avail[10],maxim[20][20],allo[20][20],m,n,need[20][20];

void accept()

{

    int i,j,val;

    for(i=0;i<n;i++)

    {

        for(j=0;j<m;j++)

        {

           printf("\nENTER THE MAX REQUIRED VALUE for P%d,R%d: ",i,j);

           loop: scanf("%d",&val);

           if(val>avail[j])

           {

               printf("ERROR!!THE ENTERED VALUE IS GREATER THAN THE AVAILABLE VALUE i.e %d .PLZ ENTER THE VALUE AGAIN: ",avail[j]);

               goto loop;

           }

           else

           maxim[i][j]=val;

        }

    }

}

void rollback(int roll[],int pro)

{

     int k;

    for(k=0;k<m;k++)

              {

              avail[k]=avail[k]+roll[k];

              allo[pro][k]=allo[pro][k]-roll[k];

              need[pro][k]=allo[pro][k]+roll[k];

              printf("\nWE ARE ROLLING BACK");

              }

}

int safety()

{

    int i,j,work[10],finish[10],flag=0,k,cnt=0,cn,flag1=0,ans;

    //char ans;

    printf("\nTHE ALTERED ALLOCATION TABLE IS:\n");

   printf("PROCESS");

   for(i=0;i<m;i++)

   {

       printf("\tR%d\t",i);

   }

   printf("\n");

   for(i=0;i<n;i++)

   {

       printf("\nP%d\t",i);

       for(j=0;j<m;j++)

       {

           printf("%d\t\t",allo[i][j]);

       }

   }

    j=0;

    for(i=0;i<m;i++)

    {

        work[j]=avail[i];

        j++;

    }

       for(i=0;i<n;i++)

      {

         finish[i]=0;

      }

   while(cnt<=10)

   {

    for(i=0;i<n;i++)

    {

        if(finish[i]==0)

        {

        for(j=0;j<m;j++)

        {

            if(need[i][j]<=work[j])

            {

                flag=1;

            }

            else

            {

                flag=0;

                break;

            }

         }

         if(flag==1)

         {

             printf("\nTHE PROCESS P%d RAN SUCCESSFULLY",i);

             finish[i]=1;

             for(k=0;k<m;k++)

             {

                 work[k]=work[k]+allo[i][k];

             }

         }

      }

    }

    for(cn=0;cn<n;cn++)

    {

        if(finish[cn]==1)

        {

            flag1=1;

        }

        else

        {

            flag1=0;

            break;

        }

    }

    cnt++;

    if(flag1==1)

    {

       printf("\nTHE SYSTEM IS IN SAFE STATE\n");

       ans=0;

       break;

     }

   }//end of while

    if(flag1==0)

    {

        printf("\nTHE SYSTEM IS NOT IN SAFE STATE\n");

        ans=1;

    }

    return ans;

}

void request()

{

    int pro,val,i,ans1,k,roll[10];

    char ans;

    while(1)

    {

        printf("\nDO YOU HAVE A REQUEST?(y/n)");

        scanf("%c",&ans);

        scanf("%c",&ans);

        k=0;

        if(ans=='y'||ans=='Y')

        {

            printf("ENTER THE PROCESS REQUESTING(0,1,2,3..): ");

            scanf("%d",&pro);

            for(i=0;i<m;i++)

            {

                printf("ENTER THE REQUIRED VALUE OF RESOURCE R%d: ",i);

               loop1: scanf("%d",&val);

                if(val>maxim[pro][i])

           {

               printf("ERROR!!THE ENTERED VALUE IS GREATER THAN THE MAX VALUE REQD. i.e %d .PLZ ENTER THE VALUE AGAIN: ",maxim[pro][i]);

               goto loop1;

           }

           else

               {

                allo[pro][i]=allo[pro][i]+val;

                avail[i]=avail[i]-val;

                need[pro][i]=need[pro][i]-val;

                roll[k]=val;

                k++;

               }

            }

            ans1=safety();

            if(ans1==1)

            {

               rollback(roll,pro);

             }

        }

        else

        {

            exit(0);

        } }}

int main()

{

   int i,val,j;

   printf("\nWELCOME TO BANKER'S ALGORITHM\n");

   printf("\nENTER THE NUMBER OF RESOURCES: ");

   scanf("%d",&m);

   printf("\nENTER THE NUMBER OF PROCESS(s): ");

   scanf("%d",&n);

   for(i=0;i<m;i++)

   {

       printf("\nENTER THE MAX AVAILABLE VALUE FOR RESOURCE R%d: ",i);

       scanf("%d",&val);

       avail[i]=val;

   }

   for(i=0;i<n;i++)

   {

       for(j=0;j<m;j++)

       {

           allo[i][j]=0;

       }

   }

   accept();

   printf("\nTHE TABLE ENTERED FOR MAX NEEDED IS:\n");

   printf("PROCESS");

   for(i=0;i<m;i++)

   {

       printf("\tR%d\t",i);

   }

   printf("\n");

   for(i=0;i<n;i++)

   {

       printf("\nP%d\t",i);

       for(j=0;j<m;j++)

       {

           printf("%d\t\t",maxim[i][j]);

       }

   }

      for(i=0;i<n;i++)

   {

       for(j=0;j<m;j++)

       {

           need[i][j]=maxim[i][j]-allo[i][j];

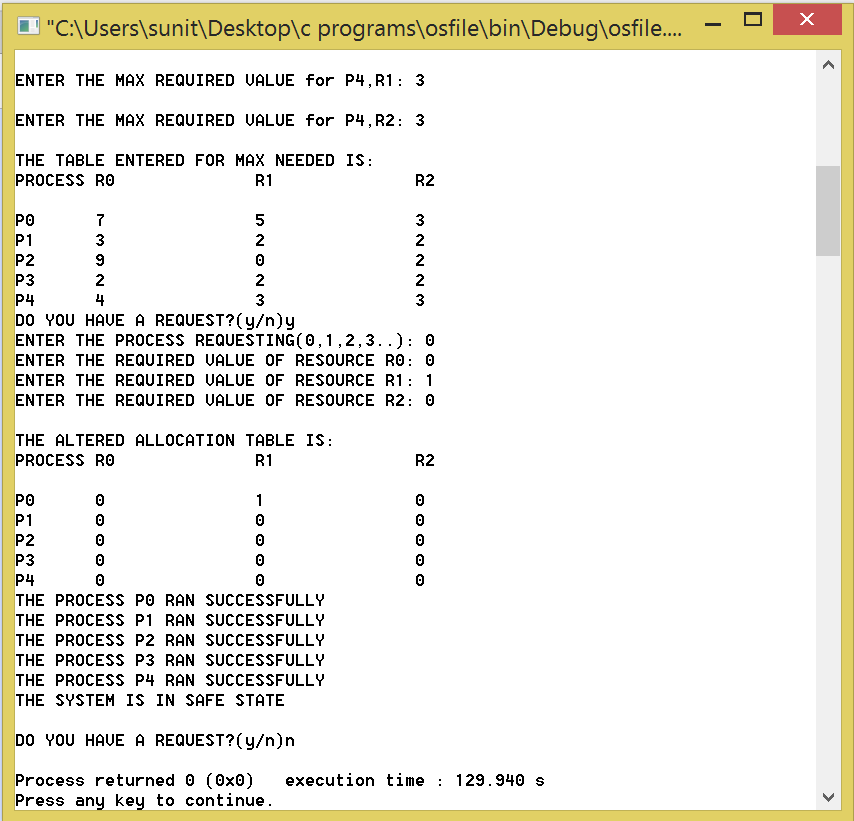
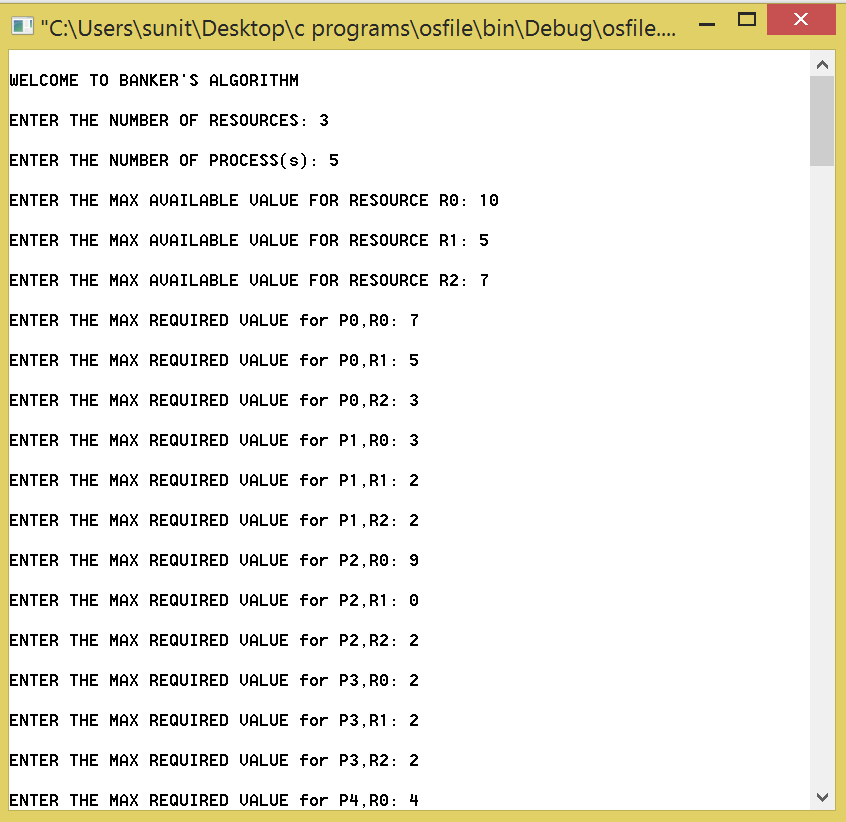
       }

   }

   request();

   return 0;

}

**Snapshot of O/P: **

**Conclusion:**  The Banker's algorithm has some limitations when implemented. Specifically, it needs to know how much of each resource a process could possibly request. In most systems, this information is unavailable, making it impossible to implement the Banker's algorithm. Also, it is unrealistic to assume that the number of processes is static since in most systems the number of processes varies dynamically.

**Learning outcome:**

**Advantages:** Avoids deadlock and it is less restrictive than deadlock prevention.

**Disadvantage:** Only works with fixed number of resources and processes.

**EXPERIMENT-6**

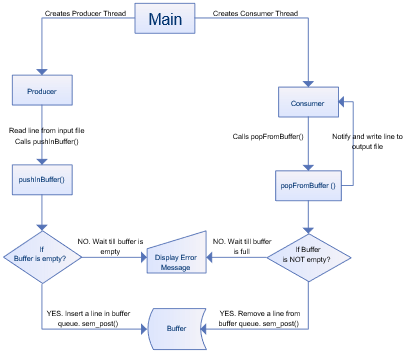
**Aim:** Write a C program to simulate producer-consumer problem.

**Description:** Producer–consumer problem. In computing, the producer–consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue.

**Algorithm:**

1. **Producer:**
2. Operate loop while counter is equal to buffer size.
3. Update the buffer array of next production.
4. Increment in and counter variables.
5. Repeat above steps while the condition is true.
6. **Consumer:**
7. Operate loop while counter is equal to 0.
8. Update the next consumer from buffer array.
9. Increment out and counter variables.
10. Repeat above steps while the condition is true.

**Flowchart:**

****

**Code:**

#include<stdio.h>

#include<stdlib.h>

int mutex=1,full=0,empty=3,x=0;

int main()

{

    int n;

    void producer();

    void consumer();

    int wait(int);

    int signal(int);

    printf("\n1.Producer\n2.Consumer\n3.Exit");

    while(1)

    {

        printf("\nEnter your choice:");

        scanf("%d",&n);

        switch(n)

        {

            case 1:    if((mutex==1)&&(empty!=0))

                        producer();

                    else

                        printf("Buffer is full!!");

                    break;

            case 2:    if((mutex==1)&&(full!=0))

                        consumer();

                    else

                        printf("Buffer is empty!!");

                    break;

            case 3:

                    exit(0);

                    break;

        }

    }

    return 0;

}

int wait(int s)

{

    return (--s);

}

int signal(int s)

{

    return(++s);

}

void producer()

{

    mutex=wait(mutex);

    full=signal(full);

    empty=wait(empty);

    x++;

    printf("\nProducer produces the item %d",x);

    mutex=signal(mutex);

}

void consumer()

{

    mutex=wait(mutex);

    full=wait(full);

    empty=signal(empty);

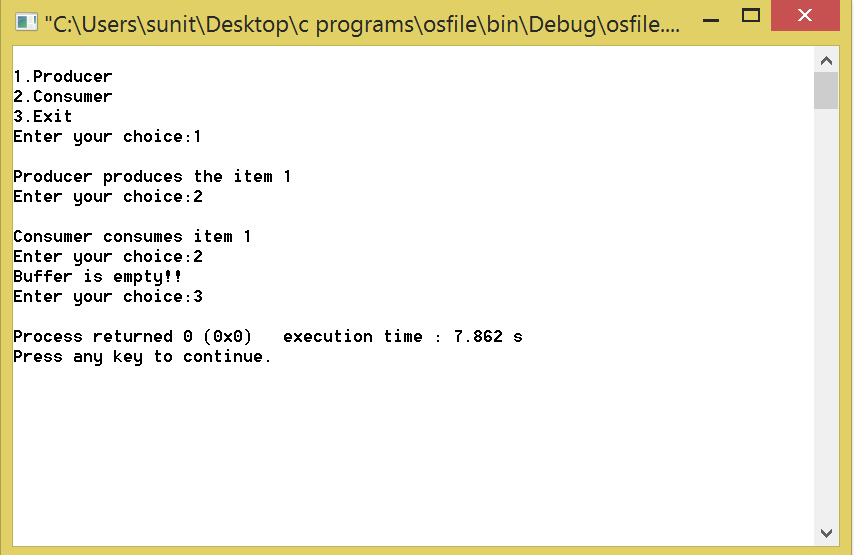
    printf("\nConsumer consumes item %d",x);

    x--;

    mutex=signal(mutex);

}

**Snapshot of O/P:**

****

**Conclusion:** The producer–consumer problem, particularly in the case of a single producer and single consumer, strongly relates to implementing a [FIFO](https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics)) or a [channel](https://en.wikipedia.org/wiki/Channel_(programming)). The producer–consumer pattern can provide highly efficient data communication without relying on semaphores, mutexes, or monitors for data transfer. Use of those primitives can give performance issues as they are expensive to implement. Channels and FIFOs are popular just because they avoid the need for end-to-end atomic synchronization.

**Learning outcome:**

**Advantages:** It gives the simpler solution of the producer consumer problem and gives us an overview of the chain cycle of the producer and the consumer alias in operating systems.

**Disadvantage:** We have to check for buffer overflow and buffer underflow while using shared region for producer consumer problem. Producer should stop producing if the buffer is full and consumer cannot consume anything if the buffer is empty.

**EXPERIMENT-7**

**Aim:** Write a C program to simulate Disk Scheduling Algorithms.

1. FCFS
2. SCAN

**Description: Disk scheduling**is done by operating systems to schedule I/O requests arriving for disk. Disk scheduling is also known as I/O scheduling.

Disk scheduling is important because:

* Multiple I/O requests may arrive by different processes and only one I/O request can be served at a time by disk controller. Thus other I/O requests need to wait in waiting queue and need to be scheduled.
* Two or more request may be far from each other so can result in greater disk arm movement.
* Hard drives are one of the slowest parts of computer system and thus need to be accessed in an efficient manner.

1. **FCFS:** FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue.
2. **SCAN:** In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works like an elevator and hence also known as **elevator algorithm.**As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

**Algorithm:**

1. **First Come First Serve Scheduling (FCFS) Algorithm:**
2. Input the maximum number of cylinders and work queue and its head starting position.
3. The operations are performed in order requested.
4. There is no reordering of work queue.
5. Every request is serviced, so there is no starvation.
6. The seek time is calculated.
7. **SCAN Scheduling Algorithm:**
8. The disk arm starts at one end of the disk, and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk.
9. At the other end, the direction of head movement is reversed, and servicing continues.
10. The head continuously scans back and forth across the disk.
11. The seek time is calculated.
12. Display the seek time and terminate the program

**Code:**

1. **FCFS:**

#include<conio.h>

#include<stdio.h>

#include<stdlib.h>

int main()

{

System(“color F0”);

int i,j,sum=0,n;

int ar[20],tm[20];

int disk;

clrscr();

printf("enter number of location\t");

scanf("%d",&n);

printf("enter position of head\t");

scanf("%d",&disk);

printf("enter elements of disk queue\n");

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

{

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

tm[i]=disk-ar[i];

if(tm[i]<0)

{

tm[i]=ar[i]-disk;

}

disk=ar[i];

sum=sum+tm[i];

}

/\*for(i=0;i<n;i++)

{

printf("\n%d",tm[i]);

} \*/

printf("\nmovement of total cylinders %d",sum);

getch();

return 0;

}

1. **SCAN:**

#include<conio.h>

#include<stdio.h>

#include<stdlib.h>

int main()

{

system("color F0");

int i,j,sum=0,n;

int d[20];

int disk; //loc of head

int temp,max;

int dloc; //loc of disk in array

printf("enter number of location\t");

scanf("%d",&n);

printf("enter position of head\t");

scanf("%d",&disk);

printf("enter elements of disk queue\n");

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

{

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

}

d[n]=disk;

n=n+1;

for(i=0;i<n;i++) // sorting disk locations

{

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

{

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

{

temp=d[i];

d[i]=d[j];

d[j]=temp;

}

}

}

max=d[n];

for(i=0;i<n;i++) // to find loc of disc in array

{

if(disk==d[i]) { dloc=i; break; }

}

for(i=dloc;i>=0;i--)

{

printf("%d -->",d[i]);

}

printf("0 -->");

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

{

printf("%d-->",d[i]);

}

sum=disk+max;

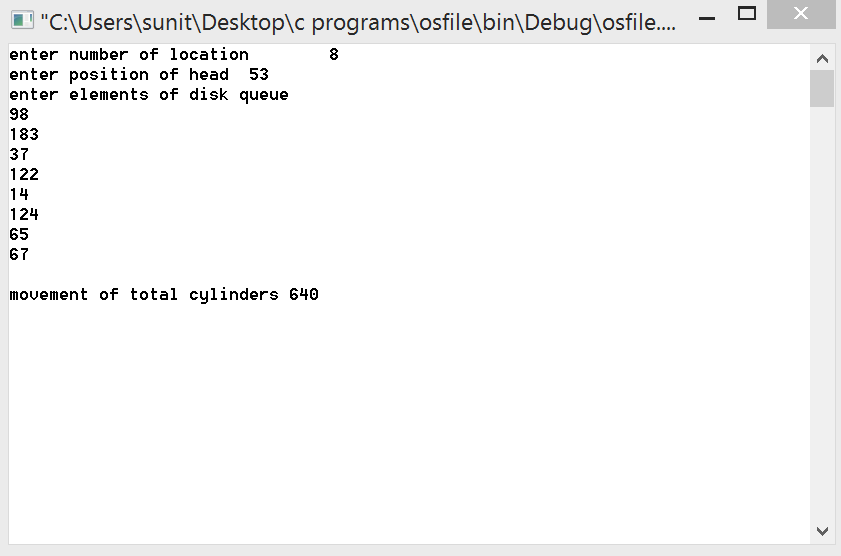
printf("\nmovement of total cylinders %d",sum);

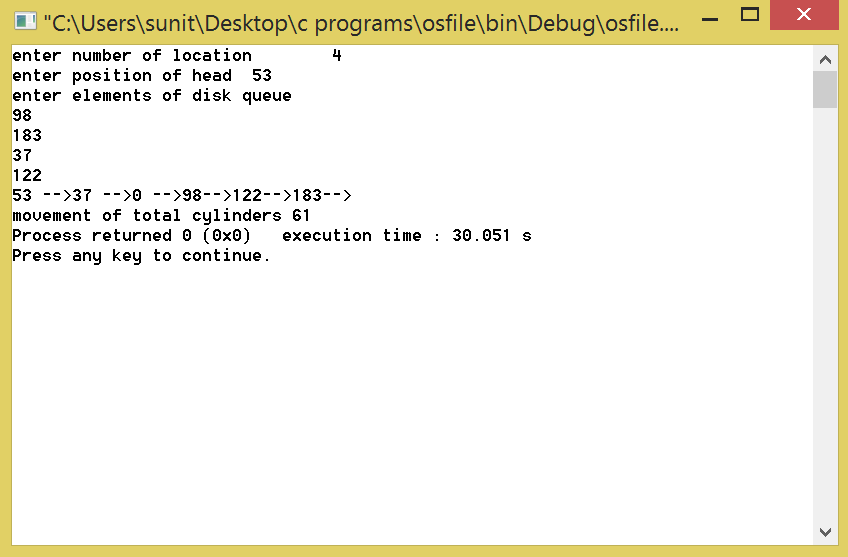
getch();

return 0;

}

**Snapshot of O/P:**

****



**Conclusion:**

**FCFS**: All incoming requests are placed at the end of the queue. Whatever number that is next in the queue will be the next number served. Using this algorithm doesn't provide the best results. To determine the number of head movements you would simply find the number of tracks it took to move from one request to the next. If you tally up the total number of tracks you will find how many tracks it had to go through before finishing the entire request.

**SCAN:** This approach works like an elevator does. It scans down towards the nearest end and then when it hits the bottom it scans up servicing the requests that it didn't get going down. If a request comes in after it has been scanned it will not be serviced until the process comes back down or moves back up.

**Learning outcome:**

1. **FCFS:**

**Advantages:**

* Every request gets a fair chance
* No indefinite postponement

**Disadvantages:**

* Does not try to optimize seek time

May not provide the best possible service

1. **SCAN:**

**Advantages:**

* High throughput
* Low variance of response time
* Average response time

**Disadvantages:**

Long waiting time for requests for locations just visited by disk