Experiment No.7
Implement deadlock detection and avoidance using Banker's algorithm.
Date of Performance:
Date of Submission:

Aim: To study and implement deadlock detection and avoidance using Banker's algorithm.

Objective: The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Theory:

Data Structures for the Banker's Algorithm.

Let n = number of processes, and m = number of resources types.

1. Available: Vector of length m. If available [j] = k, there are k instances of resource type Rj available

2.Max: n x m matrix.

If Max [i,j] = k, then process Pi may request at most k instances of resource type Rj

3. Allocation: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj

4.Need: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rj to complete its task

Need [i,j] = Max[i,j] – Allocation [i,j]

Safety Algorithm

1. Let Work and Finish be vectors of length m and n, respectively.
   Initialize:

   Work = Available

   Finish [i] = false for i = 0, 1, …, n- 1
2. Find an i such that both:

   (a) Finish [i] = false

   (b) Need ≤ Work

If no such i exists, go to step 4

3. Work = Work + Allocation

   Finish[i] = true

   go to step 2

4. If Finish [i] == true for all i, then the system is in a safe state.

Resource-Request Algorithm for Process Pi

Request = request vector for process Pi. If Request [j] = k then process Pi wants k instances of resource type R

1. If Request ≤ Need go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If Request ≤ Available, go to step 3. Otherwise Pi must wait, since resources are not available 3. Pretend to allocate requested resources to Pi by modifying the state as follows:

Available = Available – Request;

Allocation = Allocation + Request;

Need = Need – Request ;

1. If safe ⇒ the resources are allocated to Pi
2. If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored.

Result:

CODE:-

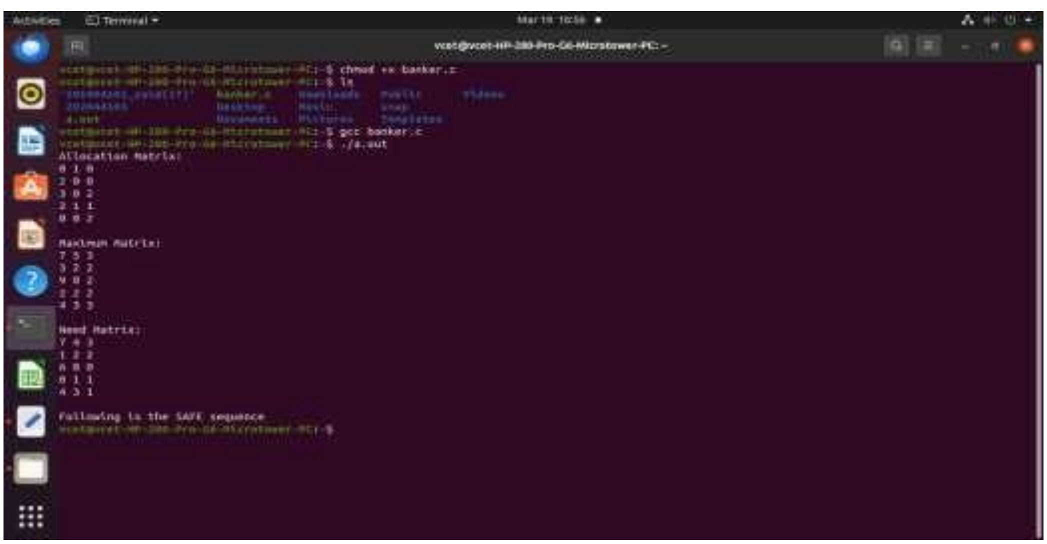
```
#include <stdio.h>

int main()

{

  int n,m,i,j,k;

  n=5;

  m=3;

  int alloc[5][3]={{0,1,0},{2,0,0},{3,0,2},{2,1,1},{0,0,2}};
```

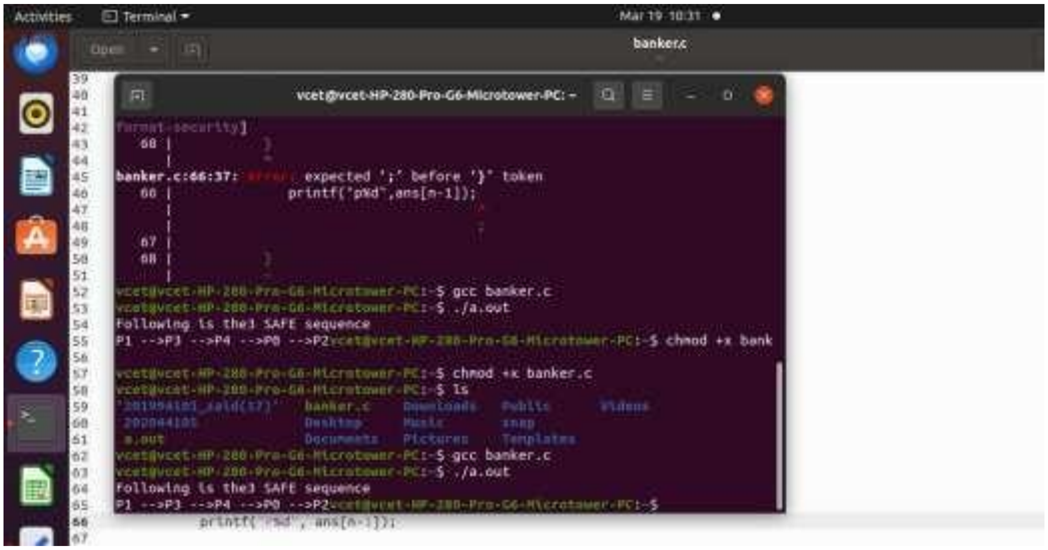
```
int max[5][3]={{7,5,3},{3,2,2},{9,0,2},{2,2,2},{4,3,3}};
int avail[3]={3,3,2};
int f[n], ans[n],ind=0;
for(k=0;k<n;k++)
{
 f[k]=0;
 }
 int need[n][m];
 for(i=0;i<n;i++)
 {
   for(j=0;j<m;j++)
   {
      need[i][j]=max[i][j]-alloc[i][j];
   }
 }
 int y=0;
 for(k=0;k<5;k++)
    {
       for(i=0;i<n;i++)
       {
       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;
             }
         }
      }
    }
  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 the3 SAFE sequence \n");
            for(i=0;i<n-1;i++)
                printf("P%d -->", ans[i]);
            printf("P%d", ans[n-1])
         }
         return 0;
}
```

OUTPUT:-





Conclusion: Within the Banker's algorithm architecture for deadlock avoidance, the safety algorithm is essential. It successfully avoids deadlock situations by carefully evaluating the system's condition and making sure that resources are distributed in a way that maintains safety. In addition to providing protection against system instability, this proactive strategy encourages effective resource usage and flexibility in response to changing resource requirements. In the context of operating systems and concurrent programming, the safety algorithm therefore serves as a vital method for preserving system stability and dependability