S.No: 13	Exp. Name: ***Implementation of Banker's algorithm.***	**Date:2025-04-12**

Aim:

Aim: Implementation of Banker's algorithm.

Description: The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for the 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.
The following **Data structures** are used to implement the Banker‟s Algorithm:
Let **'n'** be the number of processes in the system and **'m'** be the number of resource types.

Available:
It is a 1-d array of size **'m'** indicating the number of available resources of each type.
Available[j] = k means there are **'k'** instances of resource type **Rj**

Max:
- It is a 2-d array of size „**n*m'** that defines the maximum demand of each process in a system.
- Max[i, j] = k means process **Pi** may request at most **'k'** instances of resource type **Rj.**

Allocation:
- It is a 2-d array of size **'n*m'** that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process **Pi** is currently allocated **'k'** instances of resource type **Rj**

Need:
- It is a 2-d array of size **'n*m'** that indicates the remaining resource need of each process.
- Need [i, j] = k means process **Pi** currently need **'k'** instances of resource type **Rj** for its execution.
- Need [i, j] = Max [i, j] – Allocation [i, j]
- Allocationi specifies the resources currently allocated to process **Pi** and Needi specifies the additional resources that process **Pi** may still request to complete its task.

Banker's algorithm consists 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[i] 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** by modifying the state as follows:
Available = Available – Requesti Allocationi = Allocationi + Requesti Needi = Needi– Requesti

Source Code:

OptimalPage.c

```
#include<stdio.h>
int main()
{
   int count=0,m,n,process,temp,resource;
   int allocation_table[5]={0,0,0,0,0};
   int available[5],current[5][5],maximum_clAim[5][5];
   int maximum_resource[5],running[5],safe_state=0;
   printf("Enter The Total Number Of Processes: ");
   scanf("%d",&process);
   for(m=0;m<process;m++)
      {
         running[m]=1;
         count++;
      }
   printf("Enter The Total Number Of Resources To Allocate: ");
   scanf("%d",&resource);
   printf("Enter The Claim Vector: ");
   for(m=0;m<resource;m++)
      {
         scanf("%d",&maximum_resource[m]);
      }
   printf("Enter Allocated Resource Table: ");
   for(m=0;m<process;m++)
      {
         for(n=0;n<resource;n++)
            {
               scanf("%d",&current[m][n]);
            }
      }
   printf("Enter The Maximum Claim Table: ");
   for(m=0;m<process;m++)
      {
         for(n=0;n<resource;n++)
            {
               scanf("%d",&maximum_clAim[m][n]);
            }
      }
   printf("The Claim Vector: ");
   for(m=0;m<resource;m++)
      {
         printf("\t%d ",maximum_resource[m]);
      }
   printf("\nThe Allocated Resource Table\n");
   for(m=0;m<process; m++)
      {
         for(n=0;n<resource;n++)
            {
               printf("\t%d",current[m][n]);
            }
         printf("\n");
      }
   printf("The Maximum Claim Table\n");
   for(m=0; m<process; m++)
```

```
    {
       for(n=0; n<resource; n++)
          {
             printf("\t%d",maximum_clAim[m][n]);
          }
       printf("\n");
    }
 for(m=0; m<process; m++)
    {
       for(n=0; n<resource; n++)
          {
             allocation_table[n]+=current[m][n];
          }
    }
 printf("Allocated Resources");
 for(m=0;m<resource; m++)
    {
       printf("\t%d",allocation_table[m]);
    }
 for(m=0; m<resource;m++)
    {
       available[m]=maximum_resource[m]-allocation_table[m];
    }
 printf("\nAvailable Resources:");
 for(m=0;m<resource;m++)
    {
       printf("\t%d",available[m]);
    }
 printf("\n");
 while(count!=0)
    {
       safe_state=0;
       for(m=0;m<process;m++)
          {
             if(running[m])
             {
                temp=1;
                for(n=0; n<resource;n++)
                   {
                      if(maximum_clAim[m][n]-current[m][n]>available[n])
                      {
                         temp=0;
                         break;
                      }
                   }
                if(temp)
                {
                   printf("\nProcess %d Is In Execution \n",m+1);
                   running[m]=0;
                   count--;
                   safe_state=1;
                   for(n=0; n<resource;n++)
                      {
                         available[n] += current[m][n];
                      }
                   break;
```

```
                    }
                }
            }
            if(!safe_state)
            {
                printf("The Processes Are In An Unsafe State\n");
                break;
            }
            else
            {
                printf("The Process Is In A Safe State \n");
                printf("\nAvailable Vector\n");
                for(m=0;m<resource;m++)
                    {
                        printf("\t%d",available[m]);
                    }
                printf("\n");
            }
        }
    return 0;
}
```

Execution Results - All test cases have succeeded!

Test Case - 1
User Output
`Enter The Total Number Of Processes:  2`
`Enter The Total Number Of Resources To Allocate:  2`
`Enter The Claim Vector:  1 2`
`Enter Allocated Resource Table:  1 2 3 4 5 6`
`Enter The Maximum Claim Table:  2 3 4 5 6 7`
`The Claim Vector:       1       2`
`The Allocated Resource Table`
`        1       2`
`        3       4`
`The Maximum Claim Table`
`        5       6`
`        2       3`
`Allocated Resources     4       6`
`Available Resources:    -3      -4`
`The Processes Are In An Unsafe State`