Experiment No. 7

7. Implement Bankers Algorithm for Dead Lock Avoidance.

DESCRIPTION

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock. Deadlock avoidance is one of the techniques for handling deadlocks. This approach requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process. Banker's algorithm is a deadlock avoidance algorithm that is applicable to a system with multiple instances of each resource type.

Program:

```
// Banker's Algorithm
#include <stdio.h>
int main()
{
   // P0, P1, P2, P3, P4 are the Process names here

   int n, m, i, j, k;
   n = 5; // Number of processes
   m = 3; // Number of resources
   int alloc[5][3] = { { 0, 1, 0 }, // P0    // Allocation Matrix
                       { 2, 0, 0 }, // P1
                       { 3, 0, 2 }, // P2
                       { 2, 1, 1 }, // P3
                       { 0, 0, 2 } }; // P4

   int max[5][3] = { { 7, 5, 3 }, // P0    // MAX Matrix
                     { 3, 2, 2 }, // P1
                     { 9, 0, 2 }, // P2
                     { 2, 2, 2 }, // P3
                     { 4, 3, 3 } }; // P4

   int avail[3] = { 3, 3, 2 }; // Available Resources

   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;
                }
            }
        }
    }

    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]);

    return (0);
}
```

Output:

Following is the SAFE Sequence

P1 -> P3 -> P4 -> P0 -> P2

Experiment No. 8

8. Implement an Algorithm for Dead Lock Detection.

ALGORITHM:

Step 1: Start the program

Step 2: Declare the necessary variable

Step 3: Get no. Of .process, resources, max & need matrix

Step 4: Get the total resource and available resources

Step 5: Claim the deadlock occurred process

Step 6: Display the result

Step 7: Stop the program

Program:

```
#include <stdio.h>
main()
{
   int found,flag,l,p[4][5],tp,tr,c[4][5],i,j,k=1,m[5],r[5],a[5],temp[5],sum=0;
   printf("Enter total no of processes");
   scanf("%d",&tp);
   printf("Enter total no of resources");
   scanf("%d",&tr);
   printf("Enter claim (Max. Need) matrix\n");
   for(i=1;i<=tp;i++)
   {
      printf("process %d:\n",i);
      for(j=1;j<=tr;j++)
      scanf("%d",&c[i][j]);
   }
   printf("Enter allocation matrix\n");
   for(i=1;i<=tp;i++)
   {
      printf("process %d:\n",i);
      for(j=1;j<=tr;j++)
      scanf("%d",&p[i][j]);
   }
   printf("Enter resource vector (Total resources):\n");
   for(i=1;i<=tr;i++)
   {
      scanf("%d",&r[i]);
   }
   printf("Enter availability vector (available resources):\n");
   for(i=1;i<=tr;i++)
   {
```

```
   scanf("%d",&a[i]);
   temp[i]=a[i];
}
for(i=1;i<=tp;i++)
{
sum=0;
for(j=1;j<=tr;j++)
   {
      sum+=p[i][j];
   }
if(sum==0)
{
   m[k]=i;
   k++;
}
}
for(i=1;i<=tp;i++)
{
   for(l=1;l<k;l++)
   if(i!=m[l])
   {
      flag=1;
      for(j=1;j<=tr;j++)
      if(c[i][j]<temp[j])
         {
         flag=0;
         break;
         }
   }
   if(flag==1)
   {
   m[k]=i;
   k++;
   for(j=1;j<=tr;j++)
      temp[j]+=p[i][j];
   }
}
printf("deadlock causing processes are:");
for(j=1;j<=tp;j++)
{
found=0;
for(i=1;i<k;i++)
   {
   if(j==m[i])
   found=1;
   }
```

```
    if(found==0)
        printf("%d\t",j);
    }
}
```

Output:

```
Enter total no. of processes : 4
Enter total no. of resources : 5
Enter claim (Max. Need) matrix :
0 1 0 0 1
0 0 1 0 1
0 0 0 0 1
1 0 1 0 1
Enter allocation matrix :
1 0 1 1 0
1 1 0 0 0
0 0 0 1 0
0 0 0 0 0
Enter resource vector (Total resources) :
2 1 1 2 1
Enter availability vector (available resources) :
0 0 0 0 1
deadlock causing processes are : 2 3
```