**WEEK -7**

**AIM:-**

To implement Bankers algorithm for deadlock avoidance.

**Description:-**

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.

**Safety Algorithm:-**

*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:-**

*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*

**Code :-**

#include <stdio.h>

int main()

{

  int n,m,i, j, k;

  printf("enter the number of process\n");

scanf("%d",&n);

printf("enter the number of resources\n");

scanf("%d",&m);

int alloc[n][m];

int max[n][m];

int avail[m];

printf("Enter allocation matrix\n");

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

{

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

{

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

}

}

printf("Enter Max matrix\n");

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

{

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

{

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

}

}

printf("Enter available\n");

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

{

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

}

  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++)

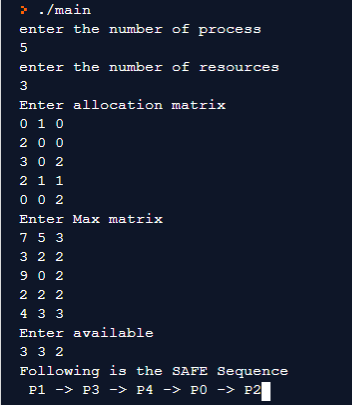
    printf(" P%d ->", ans[i]);

  printf(" P%d", ans[n - 1]);

  return (0);

}

**Output:-**



**Advantages:-**

* Avoids deadlock and it is less restrictive than deadlock prevention
* Allows mutual exclusion, wait and hold, no pre-emption.
* Prevents circular wait.

**Disadvantages:**

* It requires the number of processes to be fixed; no additional processes can start while it is executing.
* It requires that the number of resources remain fixed; no resource may go down for any reason without the possibility of deadlock occurring.
* It allows all requests to be granted in finite time, but one year is a finite amount of time.

**Observations:-**

* All processes must know and state their maximum resource need in advance.
* all of the processes guarantee that the resources loaned to them will be repaid in a finite amount of time. While this prevents absolute starvation, some pretty hungry processes might develop.
* Resources may be allocated to a process only if the amount of resources requested is less than or equal to the amount available; otherwise, the process waits until resources are available.Some of the resources that are tracked in real systems are [memory](https://en.wikipedia.org/wiki/Memory_(computers)), [semaphores](https://en.wikipedia.org/wiki/Semaphore_(programming)) and [interface](https://en.wikipedia.org/wiki/Interface_(computer_science)) access.