## AIM: Implementation of Bankers algorithm

**Objective:** To study and implement bankers algorithms.

#### Theory:

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that simulates resource allocation for predetermined maximum possible amounts of all resources before performing an "s-state" check to look for potential activities and determining whether allocation should be permitted to continue.

**Utilizing the following data structures, the Banker's Algorithm is implemented:**

Assume that there are n processes in the system and m different resource kinds.

**Available:**

* It is a 1-d array of size "m" that lists the total amount of resources of each type that are accessible.
* There are 'k' instances of the resource type if Available[j] = k. Rj

**Max:**

* The maximum demand of each process in a system is specified by a 2-d array of size 'n\*m'.
* Process Pi may request a maximum of 'k' instances of resource type Rj if

Max[i, j] = k.

**Allocation:**

* The number of resources of each type currently allocated to each process is specified by a 2-d array of size 'n\*m'.
* Process is indicated by Allocation[i, j] = k. Pi has been given 'k' instances of the resource type at this time. Rj

**Need:**

* It is a 2-d array of size 'n\*m' that shows how much more each process needs in terms of resources.
* Need I j] = k denotes that process Pi need 'k' instances of the resource type at this time. Rj
* Allocation [i, j] - Maximum [i, j] = Need [i, j]

The resources that are now allotted to process Pi are identified by Allocation, while the extra resources that process Pi could yet need in order to do its work are identified by Need.

The safety algorithm **and the resource request algorithm make up the banker's algorithm.**

**Algorithm**:

**Safety Algorithm**

1) Let us assume Work and Finish be vectors of size ‘m’ and ‘n’.

Initialize: Work = Available

Finish[i] = false; for i=1, 2,...n

2) Find i such that

a) Finish[i] = false

b) Need [i] <= 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

the system is in a safe state

**Resource-Request Algorithm**

Let us consider a vector Request for Process Pi.

Requesti[j]= k means process Pi requires k instances of Resource Rj.

1) If Request[i] <= Need[i]

Goto step (2) ;

else

raise error condition as the process exceeds its maximum claim.

2) If Request[i] <= Available

Goto step (3);

else wait as the resources are not available.

3)Pretend to have allocated the resources that are requested to process Pi by modifying the state as

Available = Available – Request[i]

Allocation[i] = Allocation[i] + Request[i]

Need[i] = Need[i]– Request[i]

**Source code:**

// 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;

}

}

}

}

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 the SAFE Sequence\n");

for (i = 0; i < n - 1; i++)

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

printf(" P%d", ans[n - 1]);

}

return (0);

}