**Deadlock Avoidance Algorithm**

**Experiment No: 5 Date: 27/09/23**

**Aim:** To implement Banker’s Algorithm for Deadlock Avoidance.

**Theory:**

Bankers’ algorithm is the deadlock avoidance algorithm used to a resource allocation system

with multiple instances of each resource type. The name was chosen because the algorithm

could be used in a banking system to ensure that the bank never allocated its available cash

in such a way that it could no longer satisfy.

When a new process enters the system, it must declare the maximum number of instances

of each resource type that it may need. This number may not exceed the total number of

resources in the system. When a user requests a set of resources, the system must

determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated, otherwise the process must wait until some other process releases enough resources.

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 R j available.

1. Max: n x m matrix. If Max [i,j] = k, then process P i may request at most

k instances of resource type R j .

1. Allocation: n x m matrix. If Allocation[i,j] = k then P i is currently allocated k instances

of R j .

1. Need: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of R j 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

1. Finish [ i ] = false for i - 1,3, …, n.
2. Find and i such that both:

(a) Finish [ i ] = false

(b) Need i ≤ Work

1. If no such i exists, go to step 4.
2. Work := Work + Allocation i
3. Finish [ i ] := true
4. go to step 2.
5. If Finish [ i ] = true for all i , then the system is in a safe state.

**Resource-Request Algorithm for Process *P i***

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

1. If *Request i* ≤ *Need i* go to step 2. Otherwise, raise error condition, since process

has exceeded its maximum claim.

1. If *Request i* ≤ *Available* , go to step 3. Otherwise, *P i* must wait, since resources

are not available.

1. Pretend to allocate requested resources to *Pi* by modifying the state as follows: Available := Available = Request i ;

Allocation i := Allocation i + Request i ;

Need i := Need i – Request i ;

• If safe ⇒ the resources are allocated to Pi.

• If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored

Example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Process | Allocation  R1 R2 R3 | Maximum  R1 R2 R3 | Available  R1 R2 R3 | Need  R1 R2 R3 |
| A | 2 2 3 | 3 6 8 | 7 7 10 | 1 4 5 |
| B | 2 0 3 | 4 3 3 | 2 2 3 | 2 3 0 |
| C | 1 2 4 | 3 4 4 | 2 2 3 | 2 2 0 |

The system is in safe sequence.

Safe Sequence – A => B => C

Current available resources – 12 10 20

**CODE:**

// Bankers Algorithm

#include <bits/stdc++.h>

using namespace std;

#define N 30

// Safety algo begins here

bool isSafe(int available[], int n, int m, int allocation[][N], int need[][N], int safeSeq[])

{

// Mark all processes as unfinished

bool fin[N];

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

fin[i] = false, safeSeq[i] = 0;

// Initializing work equal to available

int work[N];

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

work[i] = available[i];

// Find an index i such that the process is not finished and needs less than or equal to work

int count = 0;

while (count < n)

{

bool found = false;

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

{

if (!fin[i])

{

int j;

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

{

if (need[i][j] > work[j])

break;

}

if (j == m)

{

for (int k = 0; k < m; k++)

work[k] += allocation[i][k];

safeSeq[count++] = i;

fin[i] = true;

found = true;

}

}

}

// If no process is found, the system is not in a safe state

if (!found)

{

cout << "The system is not in a safe state\n";

return false;

}

}

// If all processes are finished, the system is in a safe state

cout << "The system is in a safe state\n";

return true;

}

// resource request algo begins here

// here i is the process number

// request is the request vector

bool resourceRequest(int i, int m, int n, int request[], int need[][N], int safeSeq[], int available[], int allocation[][N])

{

// request vector has a size m

// reqi[j]=k means process i wants k instances of resource j

bool tempSafe = true; // Initialize a flag for temporary safety

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

{

// if request is greater than need, return false

if (request[j] > need[i][j])

{

tempSafe = false;

break;

}

// if request is greater than available, return false

if (request[j] > available[j])

{

tempSafe = false;

break;

}

}

if (tempSafe)

{

// pretend to allocate the resources

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

{

available[j] -= request[j];

allocation[i][j] += request[j];

need[i][j] -= request[j];

}

// check if the temp state is safe

tempSafe = isSafe(available, n, m, allocation, need, safeSeq);

// revert back to original state if not safe

if (!tempSafe)

{

for (int x = 0; x < m; x++)

{

available[x] += request[x];

allocation[i][x] -= request[x];

need[i][x] += request[x];

}

}

}

return tempSafe;

}

// Utility Functions

void printMat(int x, int y, int mat[][N])

{

for (int i = 0; i < x; i++)

{

for (int j = 0; j < y; j++)

cout << mat[i][j] << " ";

cout << endl;

}

cout << endl;

}

void printDetails(int n, int m, int allocation[][N], int maxm[][N], int need[][N], int available[])

{

cout << "Allocation\n";

printMat(n, m, allocation);

cout << "Max\n";

printMat(n, m, maxm);

cout << "Need\n";

printMat(n, m, need);

cout << "Available\n";

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

cout << available[i] << " ";

cout << endl;

}

// print safe sequence

void printSafeSeq(int safeSeq[], int n)

{

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

cout << safeSeq[i] << (i == n - 1 ? "\n" : " -> ");

cout << endl;

}

int main(int argc, char const \*argv[])

{

int need[N][N], safeSeq[N];

int m, n, available[N], maxm[N][N], allocation[N][N];

cout << "Enter the number of processes: ", cin >> n;

cout << "Enter the number of resources: ", cin >> m;

cout << "How many instances of each resource are available?\n";

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

cout << "R" << i << ": ", cin >> available[i];

cout << "Enter the maximum resources required for each process\n";

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

{

cout << "P" << i << ": ";

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

cin >> maxm[i][j];

}

cout<<"Enter the number of resources allocated to each process\n";

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

{

cout << "P" << i << ": ";

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

cin >> allocation[i][j];

}

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

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

need[i][j] = maxm[i][j] - allocation[i][j];

printDetails(n, m, allocation, maxm, need, available);

if (isSafe(available, n, m, allocation, need, safeSeq))

{

cout << "The safe sequence is: ";

printSafeSeq(safeSeq, n);

}

int choice, requesting\_process, request[N];

while (1)

{

cout << "Enter choice\n1. Enter request for a new process\n2. Exit\n", cin >> choice;

switch (choice)

{

case 1:

cout << "Which Process?\n", cin >> requesting\_process;

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

cout << "R" << i << ": ", cin >> request[i];

if (resourceRequest(requesting\_process, m, n, request, need, safeSeq, available, allocation))

cout << "The request has been granted successfully Leaving the system in a Safe State\nSafe Sequence: ", printSafeSeq(safeSeq, n), printDetails(n, m, allocation, maxm, need, available);

else

cout << "The request cannot be granted. The Process needs to wait for the resources to free up!\n";

break;

case 2:

exit(EXIT\_SUCCESS);

default:

cout << "Invalid choice\n";

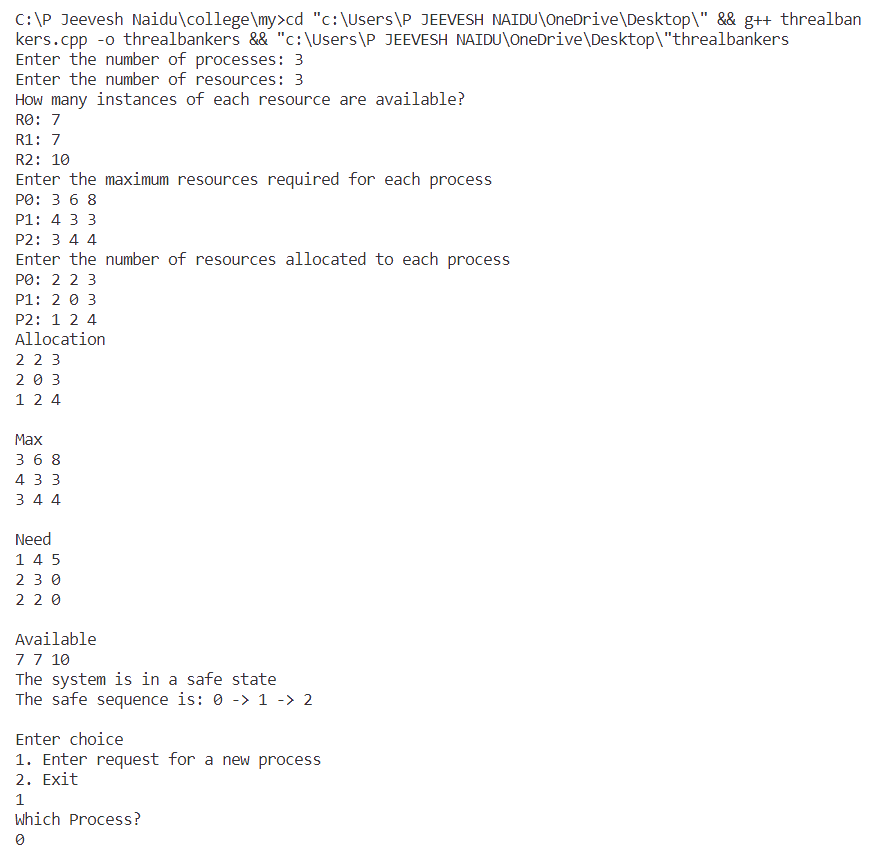
break;

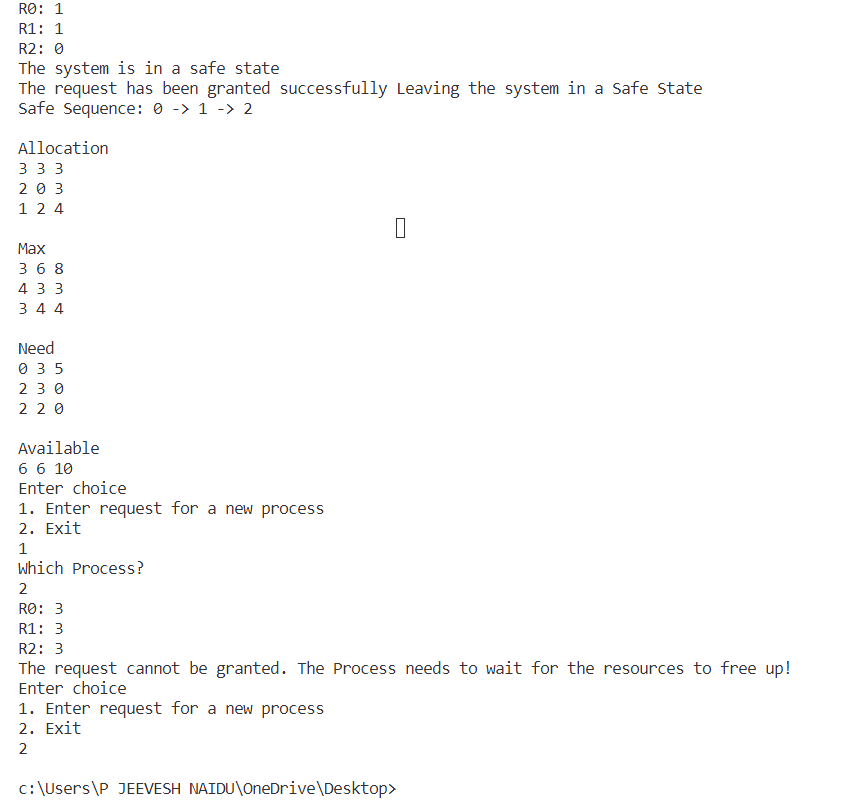
}

}

return 0;

**OUTPUT:**

****

****

**Conclusion:**

The Deadlock Avoidance algorithm – Banker’s Algorithm was successfully implemented in this experiment.