**Experiment 5**

**Title:** Deadlock Management: Bankers Algorithm

Write a C program to demonstrate the concept of deadlock avoidance using Bankers algortihm

**Estimated time to complete this experiment:** 2 hours

**Objective:** Learning about deadlock avoidance. Implementing a programfor avoiding deadlock by keeping system in safe state using Bankers algorithm.

**Expected Outcome of Experiment:** To create a temporary state of the system for each request and perform safety check. To determine whether request can be granted or not based on result of safety check.

**Books/ Journals/ Websites referred:**

1. William Stallings, Operating System: Internals and Design Principles, Prentice Hall, 8thEdition, 2014, ISBN-10: 0133805913 • ISBN-13: 9780133805918.
2. Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, Operating System Concepts, John Wiley &Sons, Inc., 9thEdition, 2016, ISBN 978-81-265-5427-0

**Pre Lab/ Prior Concepts:** Any Programming platform, Matrix operations.

**Brief description:**

When a set of processes wait in a cycle for resources to be released by other process, the system is said to be in deadlock. Deadlock can be prevented by negating one of the 4 conditions for deadlock, namely: Mutual Exclusion, Hold and Wait, No Pre-emption, Circular Wait. If none of the 4 conditions are negated then system may enter in deadlock state. In order to avoid the state of deadlock, Bankers algorithm ensures that if a request from a process i is granted then there is at-least one sequence of execution by which all processes can complete. In other words, granting the request will maintain the system in safe state. The requirement of Bankers algorithm is to obtain details of maximum requirement of resources from each process (which cannot be modified later)

**New Concepts to be learned:** Deadlock Avoidance techniques

**Requirements:** PC with any programming platform.

**Theory:**

Bankers Algorithm for deadlock avoidance works as follows:

* + Tentatively grant each resource request
  + Analyze resulting system state to see if it is “safe”.
  + If safe, grant the request
  + if unsafe refuse the request (undo the tentative grant)
  + block the requesting process until it is safe to grant it.

Data Structures used by Bankers algorithm are:

Let n = number of processes,

m = number of resource types

* Available: Vector of length *m*. If Available [*j*] = *k*, there are *k* instances of resource type *Rj* currently available
* *Max: n x m* matrix. If *Max* [*i,j*] = *k*, then process *Pi* will request at most *k* instances of resource type *Rj*.
* *Alloc: n x m* matrix. If Alloc[*i,j*] = *k* then *Pi* is currently allocated (i.e. holding) *k* instances of *Rj.*
* *Need: n x m* matrix. If *Need*[*i,j*] = *k*, then *Pi* may need *k* more instances of *Rj*to complete its task.

*NOTE: Need* [*i,j]* = *Max*[*i,j*] – *Alloc* [*i,j*].

Steps for Safety Algorithm are:

1. Let Work and Finish be vectors of length m and n, respectively.

Initialize: Work := Available.

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

1. Find an i such that both:

Finish [i] == false. Needi ≤ Work.

If no such i exists, go to step 4.

1. Work := Work + Allocationi (Resources freed when process completes!)

Finish[i] := true.

go to step 2.

1. If Finish [i] = true for all i, then the system is in a safe state.

Steps for Resource-Request Algorithm are:

Requesti = request vector for Pi .

Requesti [j] = k means process Pi wants k instances of resource type Rj.

1. If Requesti ≤ Needi go to step 2. Otherwise, error ( process exceeded its maximum claim).
2. If Requesti ≤ Available, go to step 3. Otherwise Pi must wait, (resources not available).
3. “Allocate” requested resources to Pi as follows:

Available := Available - Requesti

Alloci := Alloci + Requesti

Needi := Needi – Requesti

**If safe** ⇒ **the resources are allocated to Pi.**

**If unsafe ⇒ restore the old resource-allocation state and block Pi**

**Program:**

#include <iostream>

#include <vector>

using namespace std;

int NoOfProcess, NoOfResources;

vector<vector<int>> Allocation;

vector<vector<int>> MaxNeed;

vector<vector<int>> Need;

vector<int> MaxInstances;

vector<int> CurrentlyAvailable;

vector<int> AvailvleAfterGiving;

void SimpleFill(vector<vector<int>> &matrix, int rows, int columns)

{

    for (int i = 0; i < rows; i++)

    {

        vector<int> temp(0);

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

        {

            int temp1;

            cin >> temp1;

            temp.push\_back(temp1);

        }

        matrix.push\_back(temp);

    }

}

void CalNeed(vector<vector<int>> &matrix, int rows, int columns)

{

    for (int i = 0; i < rows; i++)

    {

        vector<int> temp(0);

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

        {

            int temp1;

            temp1 = MaxNeed[i][j] - Allocation[i][j];

            temp.push\_back(temp1);

        }

        matrix.push\_back(temp);

    }

}

void maxFill(vector<int> &matrix, int columns)

{

    for (int i = 0; i < columns; i++)

    {

        int temp;

        cin >> temp;

        matrix.push\_back(temp);

    }

}

void print2D(vector<vector<int>> &matrix, int rows, int columns)

{

    for (int i = 0; i < rows; i++)

    {

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

        {

            cout << matrix[i][j] << "\t";

        }

        cout << endl;

    }

}

void print1D(vector<int> &matrix, int columns)

{

    for (int j = 0; j < columns; j++)

    {

        cout << matrix[j] << "\t";

    }

    cout<<endl;

}

void CalCA()

{

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

    {

        int temp = 0;

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

        {

            temp += Allocation[i][j];

        }

        CurrentlyAvailable.push\_back(MaxInstances[j] - temp);

    }

}

void Banker()

{

*// Step 1: Calculate the currently available resources*

    CalCA();

*// Step 2: Initialize the visited and answer vectors*

    vector<bool> visited(NoOfProcess, false);

    vector<int> ans;

*// Step 3: Repeat until all processes have been visited*

    while (ans.size() != NoOfProcess)

    {

*// Step 4: Find a process that has not been visited and whose needs can be satisfied*

        int i;

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

        {

            if (!visited[i])

            {

                bool check = true;

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

                {

                    if (Need[i][j] > CurrentlyAvailable[j])

                    {

                        check = false;

                        break;

                    }

                }

                if (check)

                {

                    break;

                }

            }

        }

*// Step 5: If no such process is found, the system is in an unsafe state*

        if (i == (NoOfProcess-1))

        {

            cout << "The system is in an unsafe state." << endl;

            return;

        }

*// Step 6: Mark the process as visited and add it to the answer vector*

        visited[i] = true;

        ans.push\_back(i);

*// Step 7: Update the currently available resources*

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

        {

            CurrentlyAvailable[j] += Allocation[i][j];

        }

    }

*// Step 8: If all processes have been visited, the system is in a safe state*

    cout << "The system is in a safe state." << endl;

    cout << "Safe sequence: ";

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

    {

        cout << ans[i] << " ";

    }

    cout << endl;

}

int main()

{

    cout << "Enter No of process" << endl;

    cin >> NoOfProcess;

    cout << "Enter No of Resources " << endl;

    cin >> NoOfResources;

    cout << "Enter the max No. of instence for each resources" << endl;

    maxFill(MaxInstances, NoOfResources);

    cout << "Enter resource allocation for each of the process" << endl;

    SimpleFill(Allocation, NoOfProcess, NoOfResources);

    cout << "Enter  max resource needed for each of the process" << endl;

    SimpleFill(MaxNeed, NoOfProcess, NoOfResources);

    CalNeed(Need, NoOfProcess, NoOfResources);

    cout << "Allocation Matrix" << endl;

    print2D(Allocation, NoOfProcess, NoOfResources);

    cout << "Max need Matrix" << endl;

    print2D(MaxNeed, NoOfProcess, NoOfResources);

    cout << "Max avilable Instances" << endl;

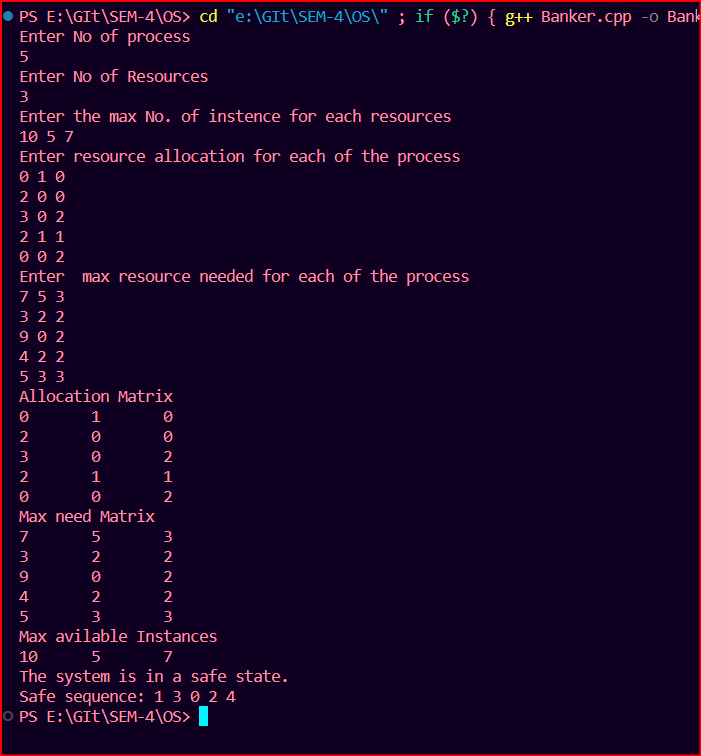
    print1D(MaxInstances, NoOfResources);

    Banker();

    return 0;

}

**Output:**

****

**Conclusion:** Hence we have understoodthe method to avoid deadlock and ensure that system is always in safe state.

**Real Life Application:** To avoid situation of deadlock in any operating system.