OPERATING SYSTEM LABORATORY MANUAL

UNIVERSITY OF THE PUNJAB

FACULTY OF COMPUTING & INFORMATION TECHNOLOGY, LAHORE

DEPARTMENT OF COMPUTER SCIENCE

Course:	**Operating System Lab**	**Date:**	
Course Code:	**CC-217-3L**	**Max Marks: 40**	
Faculty/Instructor's Name & Email:	**Dr. Ahmad Hassan Butt (ahmad.hassan@pucit.edu.pk)**		

LAB MANUAL # 14
(SPRING 2023)

Name:____________________________________ Enroll No: ____________________________

Objective(s) :

To understand concepts of Deadlocks and Banker's Algorithm.

Lab Tasks :

Task 1: What is a deadlock and Banker's Algorithm?
Task 2: Which data structures is being used in Banker's Algorithm?
Task 3 & 4: Write and analyze the program to illustrate the Banker's Algorithm.

Lab Grading Sheet :

Task	Max Marks	Obtained Marks	Comments(*if any*)
1.	10		
2.	10		
3.	10		
4.	10		
Total	**40**		**Signature**

Note : Attempt all tasks and get them checked by your Instructor

Lab 14: Deadlocks and Banker's Algorithm

Objective(s):

To understand concepts of Deadlocks and Banker's Algorithm.

Task 1: What is a deadlock and Banker's Algorithm?

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.

Task 2: Which data structures is being used in Banker's Algorithm?
Following **Data structures** are used to implement the Banker's Algorithm:
Let **'n'** be the number of processes in the system and **'m'** be the number of resources types.

Available:

- It is a 1-d array of size **'m'** indicating the number of available resources of each type.
- Available [j] = k means there are **'k'** instances of resource type $\mathbf{R_j}$

Max:

- It is a 2-d array of size '**n*m'** that defines the maximum demand of each process in a system.
- Max[i, j] = k means process $\mathbf{P_i}$ may request at most **'k'** instances of resource type $\mathbf{R_j}$.

Allocation:

- It is a 2-d array of size **'n*m'** that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process $\mathbf{P_i}$ is currently allocated **'k'** instances of resource type $\mathbf{R_j}$

Need:

- It is a 2-d array of size **'n*m'** that indicates the remaining resource need of each process.
- Need [i, j] = k means process $\mathbf{P_i}$ currently allocated **'k'** instances of resource type $\mathbf{R_j}$
- Need [i, j] = Max [i, j] – Allocation [i, j]

Allocation$_i$ specifies the resources currently allocated to process P$_i$ and Need$_i$ specifies the additional resources that process P$_i$ may still request to complete its task.

Task 3 & 4: Write and analyze the program to illustrate the Banker's Algorithm.

PROGRAM:

```
// C++ program to illustrate Banker's Algorithm
#include <iostream>
using namespace std;

// Number of processes
const int P = 5;

// Number of resources
const int R = 3;

// Function to find the need of each process
void calculateNeed(int need[P][R], int maxm[P][R],
                   int allot[P][R])
{
    // Calculating Need of each P
    for (int i = 0 ; i < P ; i++)
        for (int j = 0 ; j < R ; j++)

            // Need of instance = maxm instance -
            //                    allocated instance
            need[i][j] = maxm[i][j] - allot[i][j];
```

```
}


// Function to find the system is in safe state or not
bool isSafe(int processes[], int avail[], int maxm[][R],
            int allot[][R])
{
    int need[P][R];
    // Function to calculate need matrix
    calculateNeed(need, maxm, allot);

    // Mark all processes as infinish
    bool finish[P] = {0};

    // To store safe sequence
    int safeSeq[P];
    // Make a copy of available resources
    int work[R];
    for (int i = 0; i < R ; i++)
        work[i] = avail[i];

    // While all processes are not finished
    // or system is not in safe state.
    int count = 0;
```

```
    while (count < P)
    {
        // Find a process which is not finish and
        // whose needs can be satisfied with current
        // work[] resources.
        bool found = false;
        for (int p = 0; p < P; p++)
        {
            // First check if a process is finished,
            // if no, go for next condition
            if (finish[p] == 0)
            {
                // Check if for all resources of
                // current P need is less
                // than work
                int j;
                for (j = 0; j < R; j++)
                    if (need[p][j] > work[j])
                        break;

                // If all needs of p were satisfied.
                if (j == R)
                {
                    // Add the allocated resources of
```

```
                    // current P to the available/work
                    // resources i.e.free the resources
                    for (int k = 0 ; k < R ; k++)
                        work[k] += allot[p][k];

                    // Add this process to safe sequence.
                    safeSeq[count++] = p;

                    // Mark this p as finished
                    finish[p] = 1;

                    found = true;
                }
            }
        }
        // If we could not find a next process in safe
        // sequence.
        if (found == false)
        {
            cout << "System is not in safe state";
            return false;
        }
    }
```

```
    // If system is in safe state then
    // safe sequence will be as below
    cout << "System is in safe state.\nSafe"
         " sequence is: ";
    for (int i = 0; i < P ; i++)
        cout << safeSeq[i] << " ";

    return true;
}
// Driver code
int main()
{
    int processes[] = {0, 1, 2, 3, 4};

    // Available instances of resources
    int avail[] = {3, 3, 2};

    // Maximum R that can be allocated
    // to processes
    int maxm[][R] = {{7, 5, 3},
                     {3, 2, 2},
                     {9, 0, 2},
                     {2, 2, 2},
```

```
                      {4, 3, 3}};


    // Resources allocated to processes
    int allot[][R] = {{0, 1, 0},
                       {2, 0, 0},
                       {3, 0, 2},
                       {2, 1, 1},
                       {0, 0, 2}};


    // Check system is in safe state or not
    isSafe(processes, avail, maxm, allot);


    return 0;
}
```