## Lab 3: Implementation of Banker's Algorithm for avoiding Deadlock

**Theory:** The Banker's algorithm, developed by Edsger Dijkstra, is a resource allocation and deadlock avoidance algorithm.

When a new process enters a system, it must declare the maximum number of instances of each resource type that it may ever claim; clearly, that number may not exceed the total number of resources in the system. Also, when a process gets all its requested resources it must return them in a finite amount of time.

For the Banker's algorithm to work, it needs to know three things:

- How much of each resource each process could possibly request ("MAX")
- How much of each resource each process is currently holding ("ALLOCATED")
- How much of each resource the system currently has available ("AVAILABLE")

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.  Let **n** be the number of processes in the system and **m** be the number of resource types. Then we need the following data structures:

- **Available:** A vector of length **m** indicates the number of available resources of each type. If **Available[j] = k**, there are k instances of resource type $R_j$ available.

- **Max:** An **n x m** matrix defines the maximum demand of each process. If **Max[i, j] = k**, then $P_i$ may request at most **k** instances of resource type $R_j$.

- **Allocation:** An **n x m** matrix defines the number of resources of each type currently allocated to each process. If **Allocation[i, j] = k**, then process $P_i$ is currently allocated **k** instances of resource type $R_j$.

- **Need:** An **n x m** matrix indicates the remaining resource need of each process. If **Need[i, j] = k**, then $P_i$ may need **k** more instances of resource type $R_j$ to complete the task.

Also, **Need[i, j] = Max[i, j] – Allocation[i, j]**

**Idea:**

1. There are two vectors **copyAvailable** and **finished** of length m and n in a safety algorithm.

**Initialization**

      copyAvailable = available

      finished[i] = false; for i = 0, 1, 2, 3, 4… n-1.

2. Check the availability status for each process i for each type of resources j, such as:

      need[i][j] <= copyAvailable[j]

      finished[i] == false

      If the j does not exist, go to step 4.

3. Set,

      newAvailable[j] =  newAvailable[j] +allocation[i][j]

      finished[i] = true

      Go to step 2 to check the status of resource availability for the next process.

4. If finished[i] == true; it means that the system is safe for all processes.

**Note:** The Banker's algorithm has some limitations when implemented. Specifically, it needs to know how much of each resource a process could possibly request. In most systems, this information is unavailable, making it impossible to implement the Banker's algorithm. Also, it is unrealistic to assume that the number of processes is static since in most systems the number of processes varies dynamically. Moreover, the requirement that a process will eventually release all its resources (when the process terminates) is sufficient for the correctness of the algorithm, however it is not sufficient for a practical system. Waiting for hours (or even days) for resources to be released is usually not acceptable.

**Source Code:**

```cpp
#include <iostream>
#include <cstdlib>
#define n 3
#define m 4
using namespace std;
void computeNeed(int need[n][m], int maximum[n][m],
int allocated[n][m]);

bool isSystemSafe(int process[n], int available[m],
int maximum[n][m], int allocated[n][m]);
int main()
{
  // Total number of processes
  int process[n] = {0, 1, 2};

  // Available units of resources
  int available[m] = {3, 1, 1, 2};

  // Maximum units of resources that can be allocated to processes
  int maximum[n][m] = {{3, 3, 2, 2},
          {1, 2, 3, 4},
          {1, 3, 5, 0}};

  // Resources currently allocated to processes
  int allocated[n][m] = {{1, 2, 2, 1},
          {1, 0, 3, 3},
          {1, 2, 1, 0}};

  // Check whether the system is in safe state or not
  isSystemSafe(process, available, maximum, allocated);

  return 0;
}
void computeNeed(int need[n][m], int maximum[n][m],
int allocated[n][m])
{
```

```
  for (int i=0;i<n;i++)
    for (int j =0;j<m;j++)
      need[i][j] = maximum[i][j] - allocated[i][j];
}

bool isSystemSafe(int process[n], int available[m],
int maximum[n][m], int allocated[n][m])
{
  int i,j,k;
  bool flag;

  int need[n][m];
  // First we compute need
  computeNeed(need, maximum, allocated);

  // Initially all processes are marked as unfinished
  bool finished[n] = {false};

  // Array to store the safe sequence of execution
  int safeSequence[n];

  // Since we will be mutating available
  // array, we make its copy and work
  // on that copy
  int copyAvailable[m];
  for(i=0;i<m;i++)
    copyAvailable[i] = available[i];

  int count = 0;
  while(count<n)
  {
    // Find a process which is unfinished and whose need can be
   //satisfied from current available resources
    flag = false;
    for(i=0;i<n;i++)
    {
      if(!finished[i])
      {
```

```cpp
      // Check if all of the process need
      // is less than available
      for(j=0;j<m;j++)
        if(need[i][j]>copyAvailable[j])
          break;
      // If all of the process need are less
      // than what is avalable
      if(j == m)
      {
        // This means that the process can finish
        // its execution and when it finishes then
        // we can free the resources it was using
        for(k=0;k<m;k++)
          copyAvailable[k] += allocated[i][k];

        // This process can be executed safely
        safeSequence[count++] = i;

        // This process is finished
        finished[i] = true;

        flag = true;
      }
    }
  }
  if(flag == false)
  {
    cout<<"System is not in safe state "<<endl;
    return false;
  }
}
cout << "System is in safe state.\nSafe sequence is: ";
for (int i = 0;i<n;i++)
  cout << safeSequence[i] <<"\t";
return true;
}
```

## Task: Find the safe sequence of execution for the following.

| Process | Allocation | | | Max | | | Available | | |
|---------|---|---|---|---|---|---|---|---|---|
| | A | B | C | A | B | C | A | B | C |
| $P_0$ | 0 | 1 | 0 | 7 | 5 | 3 | 3 | 3 | 2 |
| $P_1$ | 2 | 0 | 0 | 3 | 2 | 2 | | | |
| $P_2$ | 3 | 0 | 2 | 9 | 0 | 2 | | | |
| $P_3$ | 2 | 1 | 1 | 2 | 2 | 2 | | | |
| $P_4$ | 0 | 0 | 2 | 4 | 3 | 3 | | | |