| | |
|---|---|
| **Batch: A1** | **Roll No.: 16010123012** |
| **Experiment No. 8** | |
| **Grade: AA / AB / BB / BC / CC / CD /DD** | |
| **Signature of the Staff In-charge with date** | |

**TITLE:** Implementation of Deadlock Avoidance Policy-Banker's Algorithm

---

**AIM:** Implementation of Deadlock Avoidance Policy-Banker's Algorithm

---

**Expected Outcome of Experiment:**

**CO3** Describe the problems related to process concurrency and the different synchronization mechanisms available to solve them.

---

**Books/ Journals/ Websites referred:**

1. **Silberschatz A., Galvin P., Gagne G. "Operating Systems Principles", Willey Eight edition.**

2. **Achyut S. Godbole , Atul Kahate "Operating Systems" McGraw Hill Third Edition.**

3. **William Stallings, "Operating System Internal & Design Principles", Pearson.**

4. **Andrew S. Tanenbaum, "Modern Operating System", Prentice Hall.**

---

**Pre Lab/ Prior Concepts:**

Banker's Algorithm is a resource allocation and deadlock avoidance algorithm used in operating systems. It ensures that a system remains in a safe state by carefully allocating resources to processes while avoiding unsafe states that could lead to deadlocks.

The Banker's Algorithm is a smart way for computer systems to manage how programs use resources, like memory or CPU time.

It helps prevent situations where programs get stuck and can not finish their tasks. This condition is known as deadlock.

By keeping track of what resources each program needs and what's available, the banker algorithm makes sure that programs only get what they need in a safe order.

## Components of the Banker's Algorithm

The 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 resource types.

### 1. 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 Rj

### 2. 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 Pi may request at most 'k' instances of resource type Rj.

### 3. 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 Pi is currently allocated 'k' instances of resource type Rj

### 4. 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 Pi currently needs 'k' instances of resource type Rj

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

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

Banker's algorithm consists of a Safety algorithm and a Resource request algorithm.

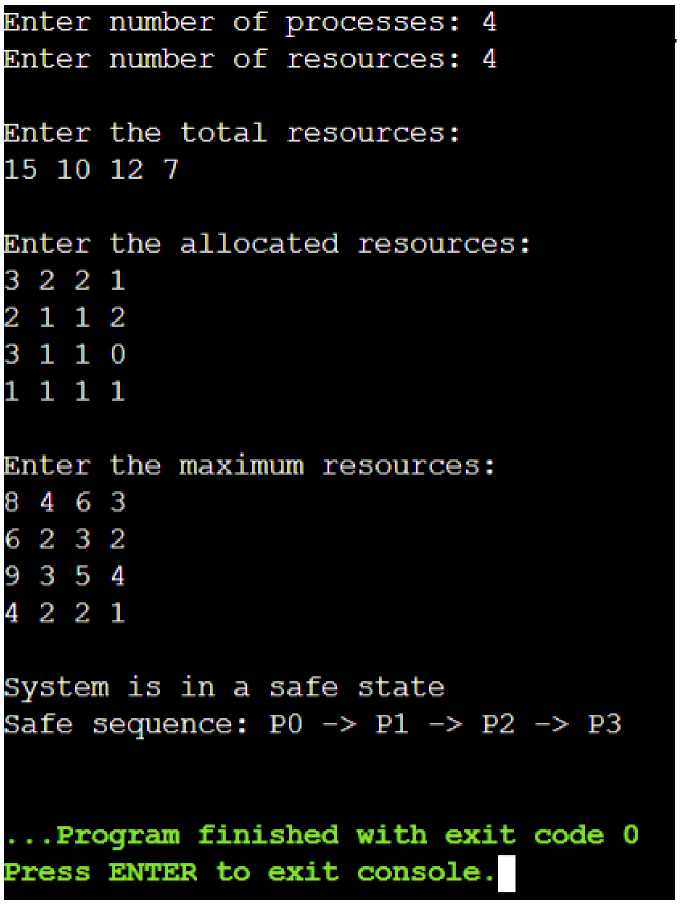
## Key Concepts in Banker's Algorithm

**Safe State:** There exists at least one sequence of processes such that each process can obtain the needed resources, complete its execution, release its resources, and thus allow other processes to eventually complete without entering a deadlock.

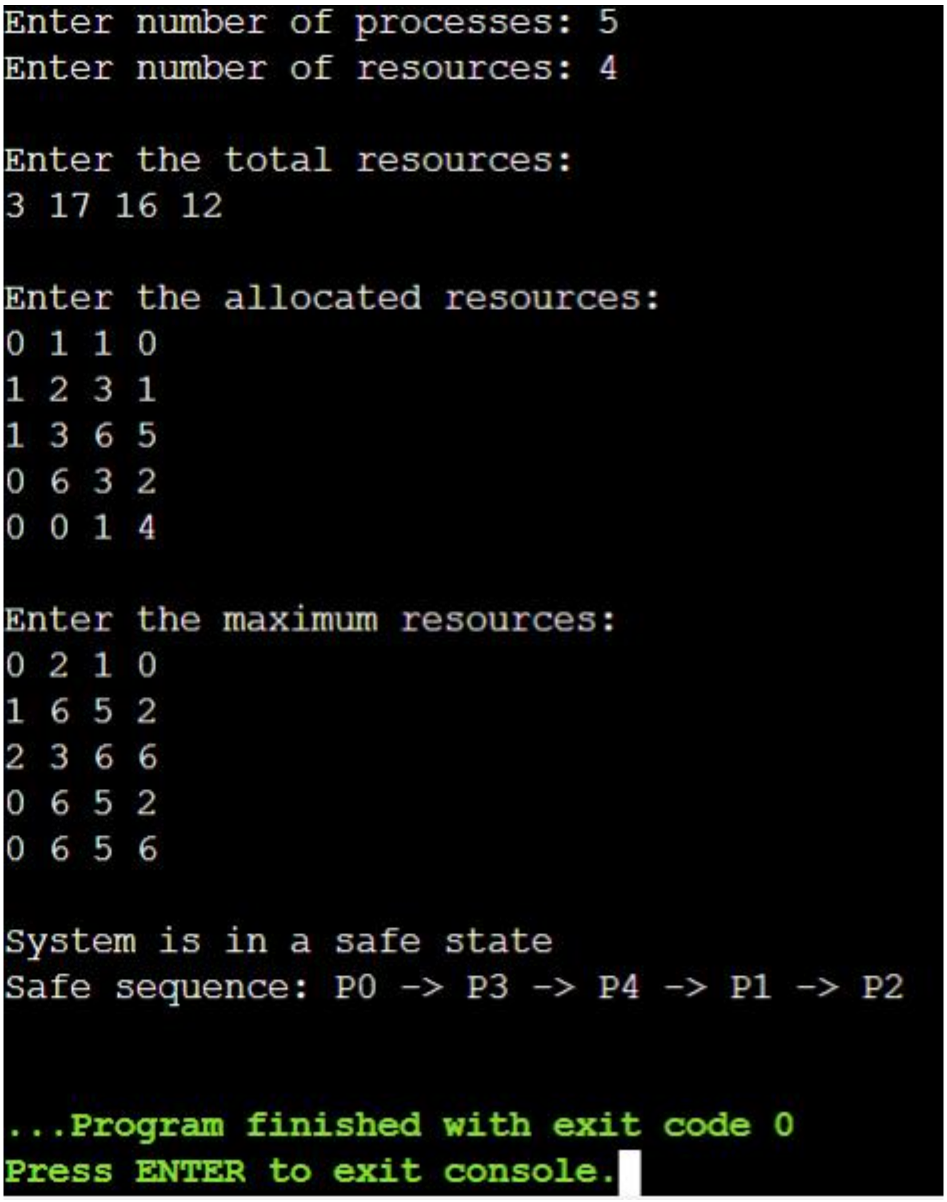
**Unsafe State:** Even though the system can still allocate resources to some processes, there is no guarantee that all processes can finish without potentially causing a deadlock.

**Implementation details:**

```cpp
#include <bits/stdc++.h>
#define endl '\n'
using namespace std;

int main()
{
    int n, m;
    cout << "Enter number of processes: ";
    cin >> n;
    cout << "Enter number of resources: ";
    cin >> m;

    vector<int> total(m), avail(m);
    vector<vector<int>> max(n, vector<int>(m));
    vector<vector<int>> need(n, vector<int>(m));
    vector<vector<int>> alloc(n, vector<int>(m));

    cout << "\nEnter the total resources: " << endl;
    for (int i = 0; i < m; i++)
    {
        cin >> total[i];
    }

    cout << "\nEnter the allocated resources: " << endl;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            cin >> alloc[i][j];
        }
    }

    cout << "\nEnter the maximum resources: " << endl;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            cin >> max[i][j];
        }
    }

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            need[i][j] = max[i][j] - alloc[i][j];
        }
    }

    for (int j = 0; j < m; j++)
    {
        int sumAlloc = 0;
        for (int i = 0; i < n; i++)
        {
            sumAlloc += alloc[i][j];
        }
        avail[j] = total[j] - sumAlloc;
    }

    vector<bool> finish(n, false);
    vector<int> safeSeq(n), work(m);
    for (int i = 0; i < m; i++)
    {
        work[i] = avail[i];
    }

    int count = 0;
    while (count < n)
    {
        bool found = false;
        for (int i = 0; i < n; i++)
        {
            if (!finish[i])
            {
                bool possible = true;
                for (int j = 0; j < m; j++)
                {
                    if (need[i][j] > work[j])
                    {
                        possible = false;
                        break;
                    }
                }
                if (possible)
                {
                    for (int j = 0; j < m; j++)
                    {
                        work[j] += alloc[i][j];
                    }
                    safeSeq[count++] = i;
                    finish[i] = true;
                    found = true;
                }
            }
        }
        if (!found)
        {
            cout << "\nSystem is in an unsafe state!" << endl;
            return 0;
        }
    }

    cout << "\nSystem is in a safe state\nSafe sequence: ";
    for (int i = 0; i < n; i++)
    {
        cout << "P" << safeSeq[i];
        if (i != n - 1)
            cout << " -> ";
    }
    cout << endl;
    return 0;
}
```



```
Enter number of processes: 4
Enter number of resources: 4

Enter the total resources:
15 10 12 7

Enter the allocated resources:
3 2 2 1
2 1 1 2
3 1 1 0
1 1 1 1

Enter the maximum resources:
8 4 6 3
6 2 3 2
9 3 5 4
4 2 2 1

System is in a safe state
Safe sequence: P0 -> P1 -> P2 -> P3


...Program finished with exit code 0
Press ENTER to exit console.
```



```
Enter number of processes: 5
Enter number of resources: 4

Enter the total resources:
3 17 16 12

Enter the allocated resources:
0 1 1 0
1 2 3 1
1 3 6 5
0 6 3 2
0 0 1 4

Enter the maximum resources:
0 2 1 0
1 6 5 2
2 3 6 6
0 6 5 2
0 6 5 6

System is in a safe state
Safe sequence: P0 -> P3 -> P4 -> P1 -> P2


...Program finished with exit code 0
Press ENTER to exit console.
```

**Conclusion:**

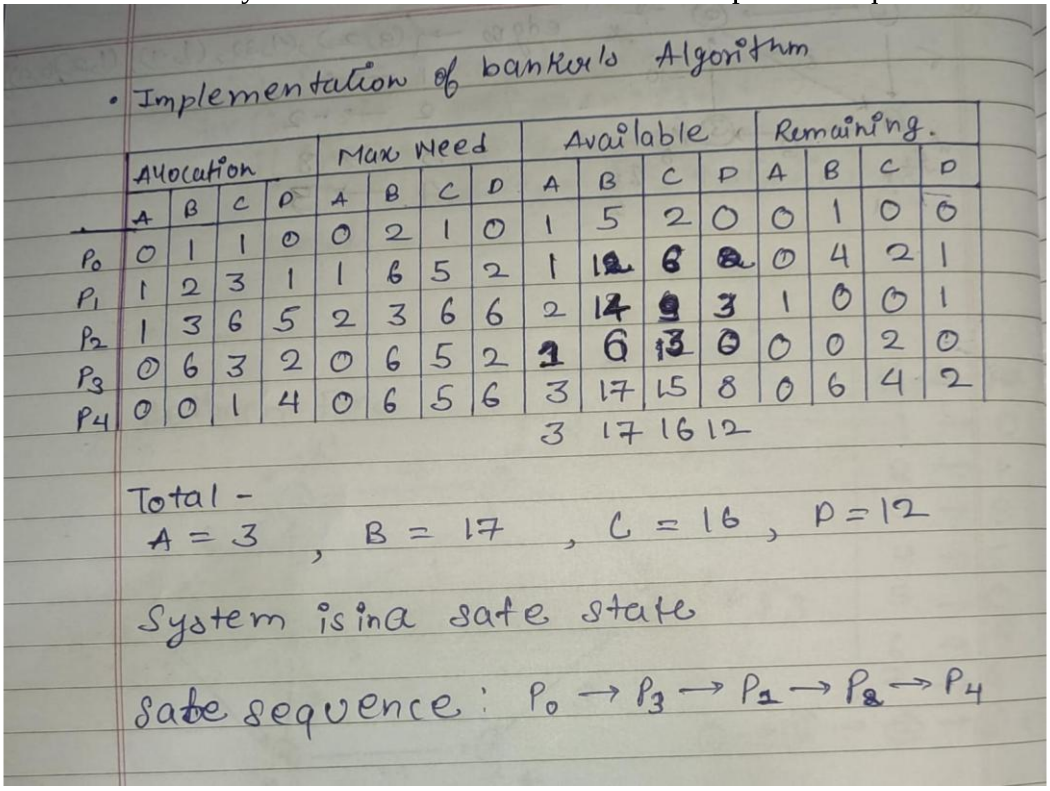
I have successfully implemented the Banker's Algorithm to ensure safe resource allocation and prevent deadlock in a multi-process system. Through this experiment, I have gained a deeper understanding of how operating systems manage resources dynamically while maintaining system stability. By evaluating process requests against available resources and determining a safe execution sequence, the algorithm effectively prevents unsafe states.

**Post Lab Descriptive Questions**

**A.** Assume that there are 5 processes, P0 through P4, and 4 types of resources. At T0 we have the following system state:
Max Instances of Resource Type A = 3
Max Instances of Resource Type B = 17
Max Instances of Resource Type C = 16
Max Instances of Resource Type D = 12

| | **Allocation Matrix** (N0 of the allocated resources By a process) | | | | **Max Matrix** Max resources that may be used by a process | | | |
|---|---|---|---|---|---|---|---|---|
| | **A** | **B** | **C** | **D** | **A** | **B** | **C** | **D** |
| **$P_0$** | 0 | 1 | 1 | 0 | 0 | 2 | 1 | 0 |
| **$P_1$** | 1 | 2 | 3 | 1 | 1 | 6 | 5 | 2 |
| **$P_2$** | 1 | 3 | 6 | 5 | 2 | 3 | 6 | 6 |
| **$P_3$** | 0 | 6 | 3 | 2 | 0 | 6 | 5 | 2 |
| **$P_4$** | 0 | 0 | 1 | 4 | 0 | 6 | 5 | 6 |

Find whether the system is in a safe state or not with the process sequence.



• Implementation of banker's Algorithm

| | Allocation | | | | Max Need | | | | Available | | | | Remaining. | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | D | A | B | C | D | A | B | C | D | A | B | C | D |
| $P_0$ | 0 | 1 | 1 | 0 | 0 | 2 | 1 | 0 | 1 | 5 | 2 | 0 | 0 | 1 | 0 | 0 |
| $P_1$ | 1 | 2 | 3 | 1 | 1 | 6 | 5 | 2 | 1 | 12 | 6 | 2 | 0 | 4 | 2 | 1 |
| $P_2$ | 1 | 3 | 6 | 5 | 2 | 3 | 6 | 6 | 2 | 14 | 9 | 3 | 1 | 0 | 0 | 1 |
| $P_3$ | 0 | 6 | 3 | 2 | 0 | 6 | 5 | 2 | 1 | 6 | 13 | 0 | 0 | 0 | 2 | 0 |
| $P_4$ | 0 | 0 | 1 | 4 | 0 | 6 | 5 | 6 | 3 | 17 | 15 | 8 | 0 | 6 | 4 | 2 |
| | | | | | | | | | 3 | 17 | 16 | 12 | | | | |

Total -
A = 3 , B = 17 , C = 16 , D = 12

System is in a safe state

Safe sequence : $P_0 \rightarrow P_3 \rightarrow P_1 \rightarrow P_2 \rightarrow P_4$