Batch: E-2	**Roll No.: 16010123325**
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.

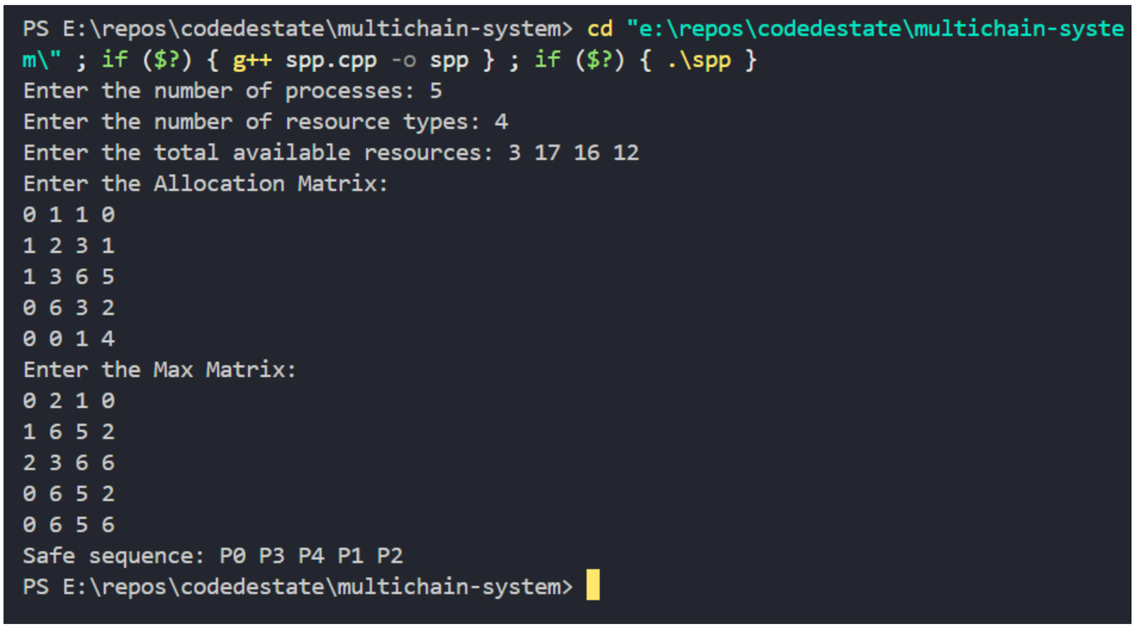
1. Available
It is a 1-D array of size 'm' indicating the number of available resources of each type.
Available[j] = k means there are 'k' instances of resource type Rj
2. Max
It is a 2-d array of size 'n*m' that defines the maximum demand of each process in a system.
Max[i, j] = k means process Pi may 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:

```
#include <bits/stdc++.h>
using namespace std;

#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

int main() {
    int n, m;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    printf("Enter the number of resource types: ");
    scanf("%d", &m);

    int Max[MAX_PROCESSES][MAX_RESOURCES],
Allocation[MAX_PROCESSES][MAX_RESOURCES];
    int Need[MAX_PROCESSES][MAX_RESOURCES], Available[MAX_RESOURCES];
    int Total[MAX_RESOURCES];


    printf("Enter the total available resources: ");
    for (int i = 0; i < m; i++) {
        scanf("%d", &Total[i]);
    }


    printf("Enter the Allocation Matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            scanf("%d", &Allocation[i][j]);
        }
    }

    printf("Enter the Max Matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            scanf("%d", &Max[i][j]);
        }
    }


    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            Need[i][j] = Max[i][j] - Allocation[i][j];
        }
    }
    for (int j = 0; j < m; j++) {
        int sum = 0;
        for (int i = 0; i < n; i++) {
            sum += Allocation[i][j];
        }
        Available[j] = Total[j] - sum;
    }


    int safeSequence[MAX_PROCESSES], work[MAX_RESOURCES];
    bool finish[MAX_PROCESSES] = {false};


    for (int i = 0; i < m; i++) {
        work[i] = Available[i];
    }

    int count = 0;
    while (count < n) {
        bool found = false;
        for (int i = 0; i < n; i++) {
            if (!finish[i]) {
                bool canAllocate = true;
                for (int j = 0; j < m; j++) {
                    if (Need[i][j] > work[j]) {
                        canAllocate = false;
                        break;
                    }
                }
                if (canAllocate) {
                    for (int j = 0; j < m; j++) {
                        work[j] += Allocation[i][j];
                    }
                    safeSequence[count++] = i;
                    finish[i] = true;
                    found = true;
                }
            }
        }
        if (!found) {
            printf("The system is in an unsafe state!\n");
            return 0;
        }
    }
    printf("Safe sequence: ");
    for (int i = 0; i < n; i++) {
        printf("P%d ", safeSequence[i]);
    }
    printf("\n");

    return 0;
}
```

Output



```
PS E:\repos\codedestate\multichain-system> cd "e:\repos\codedestate\multichain-syste
m\" ; if ($?) { g++ spp.cpp -o spp } ; if ($?) { .\spp }
Enter the number of processes: 5
Enter the number of resource types: 4
Enter the total available resources: 3 17 16 12
Enter the Allocation Matrix:
0 1 1 0
1 2 3 1
1 3 6 5
0 6 3 2
0 0 1 4
Enter the Max Matrix:
0 2 1 0
1 6 5 2
2 3 6 6
0 6 5 2
0 6 5 6
Safe sequence: P0 P3 P4 P1 P2
PS E:\repos\codedestate\multichain-system>
```

Conclusion :

The above experiment introduces implementation of deadlock avoidance or deadlock handing by use of Banker's Algorithm, which checks if processes for a given set of resources lead into deadlock otherwise gives their safe sequence of execution.

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

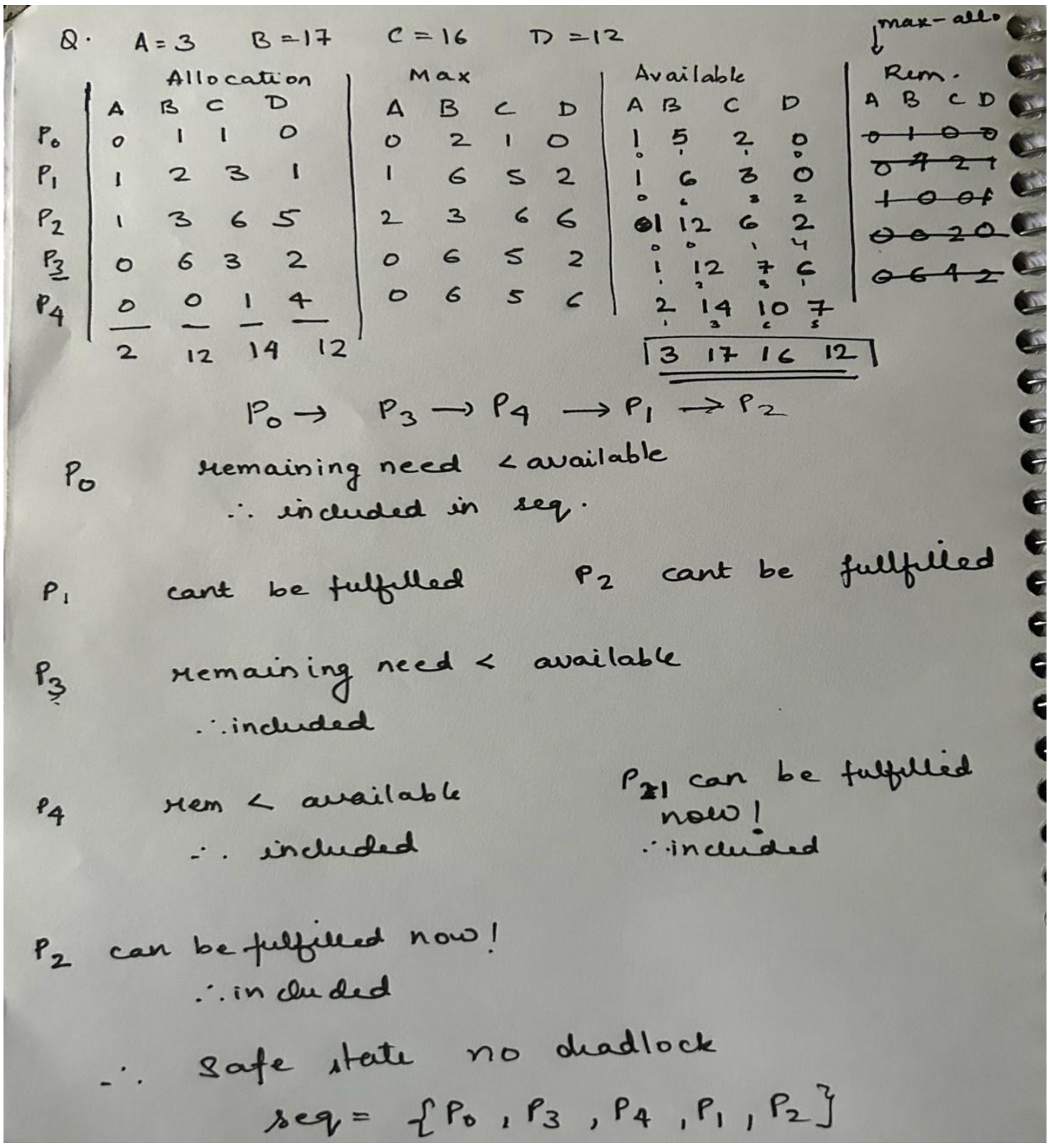
Max Instances of Resource Type B = 17

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.



Q· A = 3 B = 17 C = 16 D = 12

	Allocation				Max				Available				Rem. (max − allo)			
	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	6	3	0	~~0~~	~~4~~	~~2~~	~~1~~
P_2	1	3	6	5	2	3	6	6	~~0~~1	12	6	2	~~1~~	~~0~~	~~0~~	~~1~~
P_3	0	6	3	2	0	6	5	2	1	12	7	6	~~0~~	~~0~~	~~2~~	~~0~~
P_4	0	0	1	4	0	6	5	6	2	14	10	7	~~0~~	~~6~~	~~4~~	~~2~~
	2	12	14	12					3	17	16	12				

$P_0 \rightarrow P_3 \rightarrow P_4 \rightarrow P_1 \rightarrow P_2$

P_0 remaining need < available
∴ included in seq.

P_1 cant be fulfilled P_2 cant be fullfilled

P_3 remaining need < available
∴ included

P_4 rem < available
∴ included

P_1 can be fulfilled now!
∴ included

P_2 can be fulfilled now!
∴ included

∴ Safe state no deadlock
seq = {P_0, P_3, P_4, P_1, P_2}

Date: __________   **Signature of faculty in-charge**