EXPERIMENT NO. 06

Objective

To implement the Banker's Algorithm using C to determine whether a system is in a safe state and to avoid deadlock.

Theory

Banker's Algorithm is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra. It 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.

Key Component	Description
Max Matrix	Maximum demand of each process
Allocation Matrix	Current resource allocation to each process
Need Matrix	Remaining needs = Max - Allocation
Available Vector	Currently available resources
Safe Sequence	Order in which all processes can safely execute without deadlock

Example

Process	Max	Allocation	Need
P0	7 5 3	0 1 0	7 4 3
P1	3 2 2	2 0 0	1 2 2
P2	9 0 2	3 0 2	6 0 0
P3	2 2 2	2 1 1	0 1 1
P4	4 3 3	0 0 2	4 3 1

Banker's Algorithm

1. Input the number of processes and resource types.
2. Input the Allocation, Max, and Available matrices.
3. Calculate the Need matrix as Need[i][j] = Max[i][j] - Allocation[i][j].
4. Initialize Finish[] array to false for all processes.
5. Repeat until all processes are marked Finish = true.
6. Find a process whose Need ≤ Available and Finish is false.
7. If such a process is found, allocate its resources and add to Available.
8. Mark that process as Finish = true and add it to the safe sequence.
9. If no such process is found and not all Finish = true, exit as unsafe.
10. If all processes finish, print the safe sequence.

C Program for Banker's Algorithm

```c
#include <stdio.h>

#include <stdbool.h>

#define NUM_PROCESSES 5
#define NUM_RESOURCES 3

int main() {
    int allocated[NUM_PROCESSES][NUM_RESOURCES] = {
        {0, 1, 0}, {2, 0, 0}, {3, 0, 2},
        {2, 1, 1}, {0, 0, 2}
    };

    int maximum[NUM_PROCESSES][NUM_RESOURCES] = {
        {7, 5, 3}, {3, 2, 2}, {9, 0, 2},
        {2, 2, 2}, {4, 3, 3}
    };

    int available[NUM_RESOURCES] = {3, 3, 2};
    int need[NUM_PROCESSES][NUM_RESOURCES];
    int finish[NUM_PROCESSES] = {0};
    int safeSequence[NUM_PROCESSES];

    // Calculate the need matrix
    for (int i = 0; i < NUM_PROCESSES; i++) {
        for (int j = 0; j < NUM_RESOURCES; j++) {
            need[i][j] = maximum[i][j] - allocated[i][j];
        }
    }

    int count = 0;
    while (count < NUM_PROCESSES) {
        bool found = false;
        for (int p = 0; p < NUM_PROCESSES; p++) {
            if (!finish[p]) {
                bool canAllocate = true;
                for (int r = 0; r < NUM_RESOURCES; r++) {
                    if (need[p][r] > available[r]) {
                        canAllocate = false;
                        break;
                    }
                }
                if (canAllocate) {
                    for (int r = 0; r < NUM_RESOURCES; r++) {
                        available[r] += allocated[p][r];
```

```
                }
                safeSequence[count++] = p;
                finish[p] = 1;
                found = true;
            }
        }
    }
    if (!found) {
        printf("System is in an UNSAFE state. Deadlock possible.\n");
        return 1;
    }
}

printf("System is in a SAFE state.\nSafe Sequence is: ");
for (int i = 0; i < NUM_PROCESSES; i++) {
    printf("P%d ", safeSequence[i]);
}
printf("\n");
return 0;
}
```

Sample Output

System is in a SAFE state.
Safe Sequence is: P1 P3 P4 P0 P2