TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

THAPATHALI CAMPUS

A Lab Report of Distributed Systems on Implementation of Banker's Algorithm for avoiding Deadlock

Submitted by:

Dipesh Bartaula

[THA076BCT015]

Submitted to:

Department of Electronics and Computer Engineering

Thapathali Campus

Kathmandu, Nepal

August 6, 2023

TITLE: IMPLEMENTATION OF BANKER'S ALGORITHM FOR AVOIDING DEADLOCK

THEORY:

The Banker's Algorithm is a deadlock avoidance algorithm used to manage resource allocation in a way that prevents potential deadlocks. It was introduced by Edsger W. Dijkstra and is commonly used in operating systems. The algorithm is designed to ensure that resource requests by processes are safe and will not lead to a deadlock situation.

In the context of the Banker's Algorithm, resources are represented as a set of types, and each type can have multiple instances. Processes request and release resources during their execution. The algorithm maintains information about the maximum demand of each process, the number of resources currently allocated to each process, and the remaining needs for each process.

The Banker's Algorithm works as follows:

- Initialization: The system starts with the number of available instances of each resource and the maximum demand for each process.
- Request: When a process requests resources, the algorithm checks if granting those resources will lead to an unsafe state or not. If the request can be safely granted, it is allocated; otherwise, the process is blocked until the requested resources are available.
- Release: When a process releases resources, the resources become available for other processes to use.
- Safety Check: The algorithm periodically checks if the system is in a safe state by simulating the resource allocation and deallocation to ensure no deadlock can occur. If the system is safe, it proceeds with resource allocation; otherwise, it waits until a safe state is achieved.

ALGORITHM:

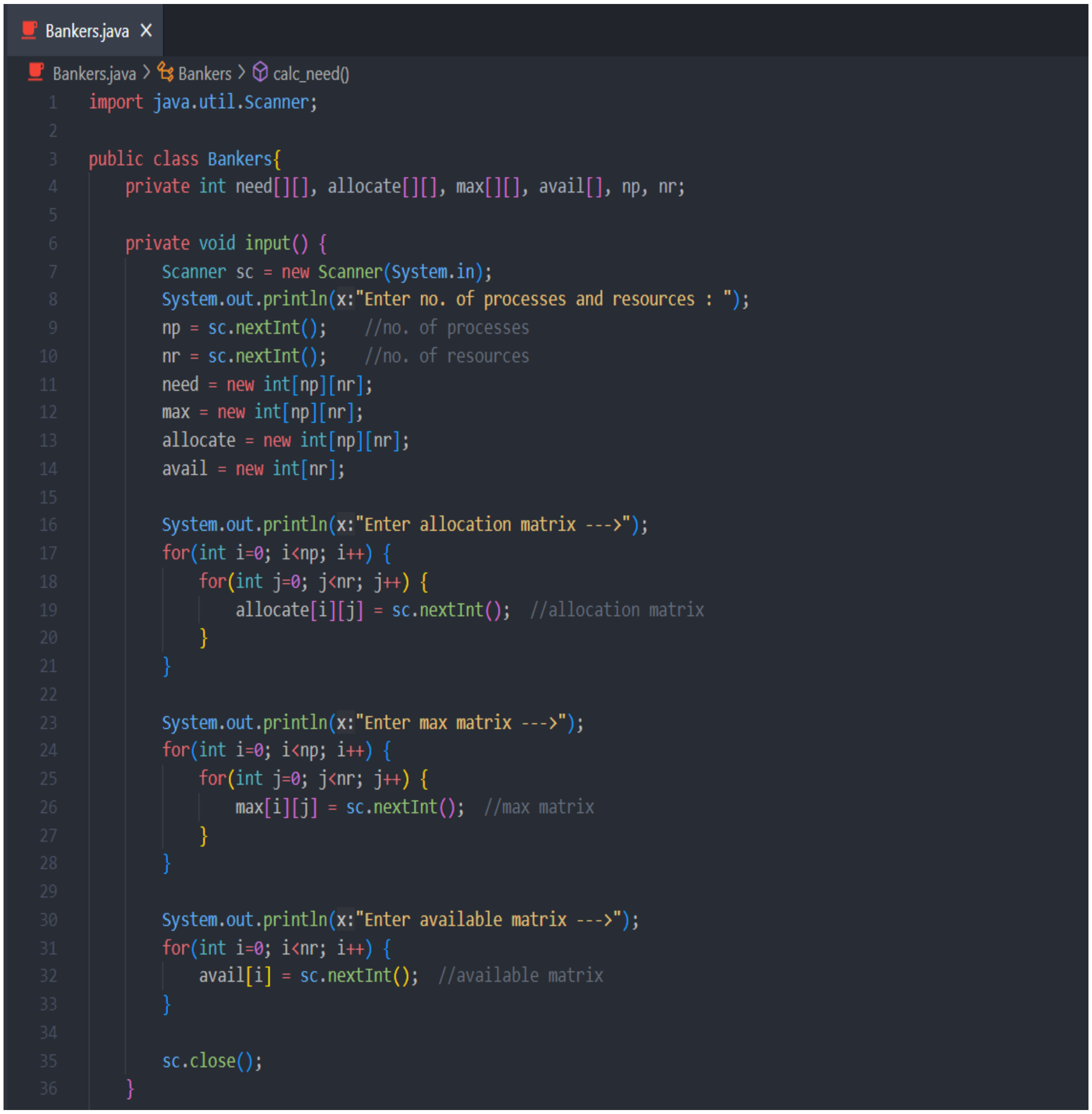
The Banker's Algorithm can be summarized in the following steps:

1) Initialize data structures for available, maximum, allocation, and need matrices.
2) Handle resource request:
   a) If the requested resources are less than or equal to the available resources, check if granting the request will lead to a safe state.
   b) If the system remains in a safe state, grant the requested resources and update the available and allocation matrices.
   c) Otherwise, block the process until the requested resources become available.

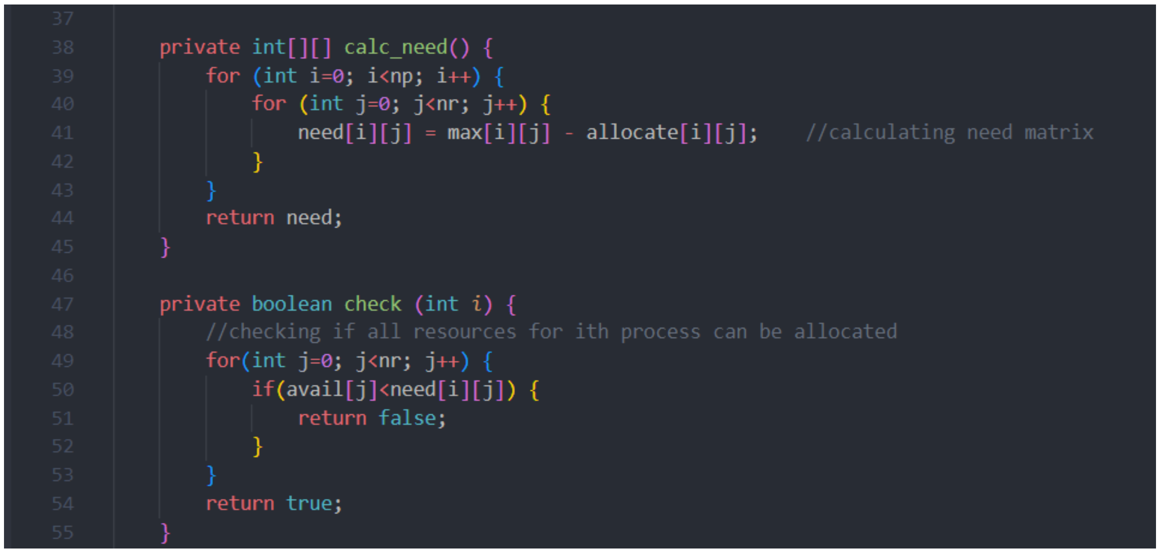
3) Handle resource release:
   a) Release the resources and update the available and allocation matrices.

4) Periodically perform the safety check to ensure the system remains in a safe state.

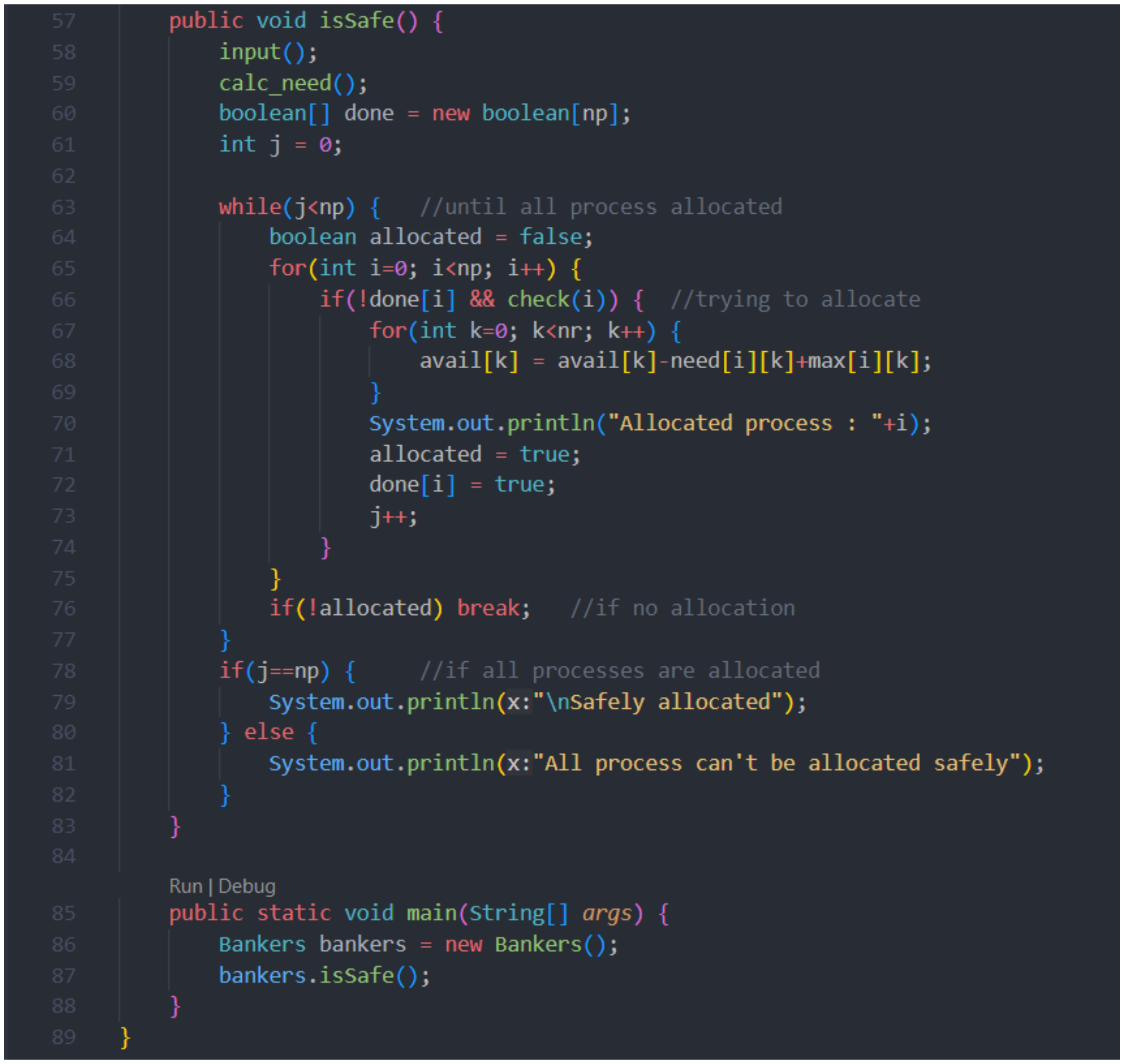
CODE:



```java
import java.util.Scanner;

public class Bankers{
    private int need[][], allocate[][], max[][], avail[], np, nr;

    private void input() {
        Scanner sc = new Scanner(System.in);
        System.out.println(x:"Enter no. of processes and resources : ");
        np = sc.nextInt();    //no. of processes
        nr = sc.nextInt();    //no. of resources
        need = new int[np][nr];
        max = new int[np][nr];
        allocate = new int[np][nr];
        avail = new int[nr];

        System.out.println(x:"Enter allocation matrix --->");
        for(int i=0; i<np; i++) {
            for(int j=0; j<nr; j++) {
                allocate[i][j] = sc.nextInt();  //allocation matrix
            }
        }

        System.out.println(x:"Enter max matrix --->");
        for(int i=0; i<np; i++) {
            for(int j=0; j<nr; j++) {
                max[i][j] = sc.nextInt();  //max matrix
            }
        }

        System.out.println(x:"Enter available matrix --->");
        for(int i=0; i<nr; i++) {
            avail[i] = sc.nextInt();  //available matrix
        }

        sc.close();
    }
```



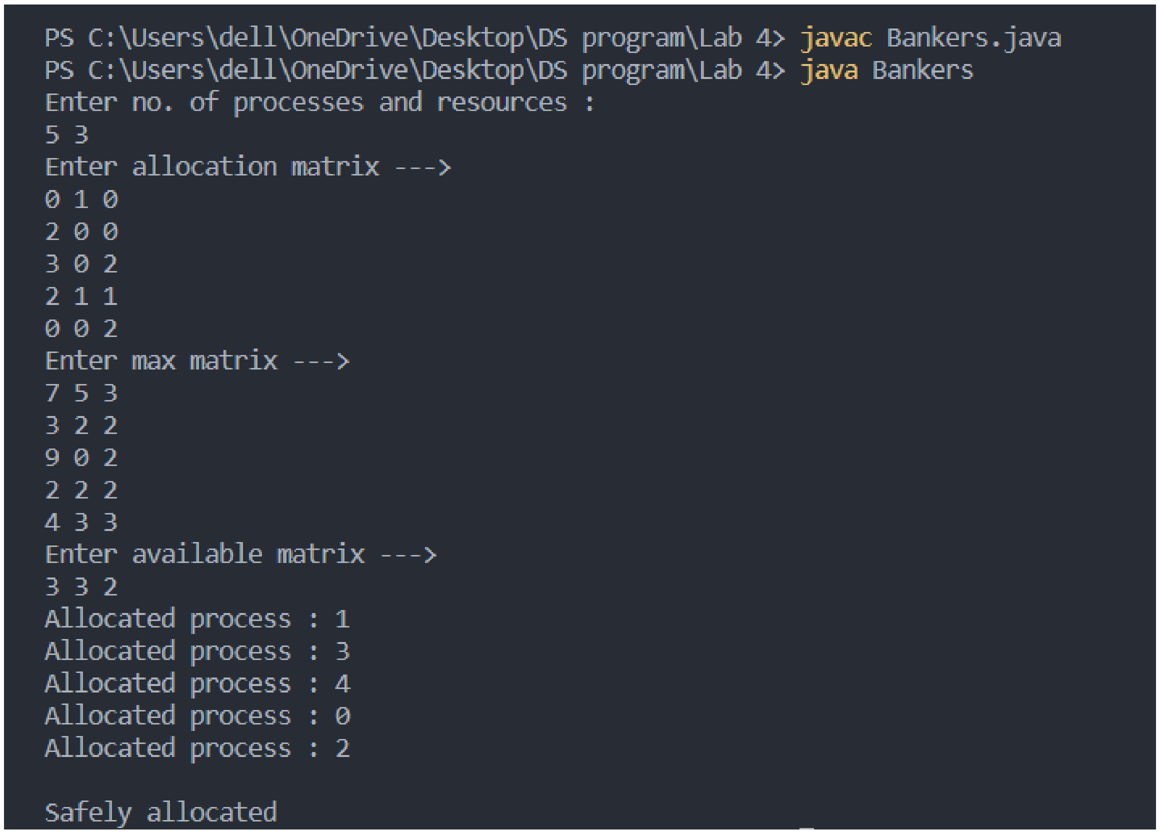
```java

    private int[][] calc_need() {
        for (int i=0; i<np; i++) {
            for (int j=0; j<nr; j++) {
                need[i][j] = max[i][j] - allocate[i][j];    //calculating need matrix
            }
        }
        return need;
    }

    private boolean check (int i) {
        //checking if all resources for ith process can be allocated
        for(int j=0; j<nr; j++) {
            if(avail[j]<need[i][j]) {
                return false;
            }
        }
        return true;
    }
```



```java
    public void isSafe() {
        input();
        calc_need();
        boolean[] done = new boolean[np];
        int j = 0;

        while(j<np) {   //until all process allocated
            boolean allocated = false;
            for(int i=0; i<np; i++) {
                if(!done[i] && check(i)) {  //trying to allocate
                    for(int k=0; k<nr; k++) {
                        avail[k] = avail[k]-need[i][k]+max[i][k];
                    }
                    System.out.println("Allocated process : "+i);
                    allocated = true;
                    done[i] = true;
                    j++;
                }
            }
            if(!allocated) break;   //if no allocation
        }
        if(j==np) {     //if all processes are allocated
            System.out.println(x:"\nSafely allocated");
        } else {
            System.out.println(x:"All process can't be allocated safely");
        }
    }

    Run | Debug
    public static void main(String[] args) {
        Bankers bankers = new Bankers();
        bankers.isSafe();
    }
}
```

The provided code implements the Banker's Algorithm for resource allocation in an operating system. It takes user input for the allocation, maximum need, and available resources for a set of processes and resources. The algorithm calculates the need matrix, which represents the remaining resources needed by each process to complete its task. It then tries to allocate resources to processes in a safe manner, ensuring that the system remains in a safe state and avoids deadlock. If all processes can be allocated resources safely, it prints the order of processes in which they are allocated and displays "Safely allocated." If any process cannot be allocated resources safely, it prints "All process can't be allocated safely."

OUTPUT:



```
PS C:\Users\dell\OneDrive\Desktop\DS program\Lab 4> javac Bankers.java
PS C:\Users\dell\OneDrive\Desktop\DS program\Lab 4> java Bankers
Enter no. of processes and resources :
5 3
Enter allocation matrix --->
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter max matrix --->
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter available matrix --->
3 3 2
Allocated process : 1
Allocated process : 3
Allocated process : 4
Allocated process : 0
Allocated process : 2

Safely allocated
```

The output of the program indicates the result of applying the Banker's Algorithm to the given resource allocation scenario. The algorithm successfully allocates resources to processes in a safe manner, avoiding potential deadlocks. It displays the order in which processes are allocated resources, represented by their process numbers. In this specific scenario, the processes are allocated resources in the following order: 1, 3, 4, 0, and 2. After all processes are allocated, the program prints "Safely allocated," indicating that the system remains in a safe state, and all processes have received the necessary resources to complete their tasks without causing any resource deadlock.

DISCUSSION:

In this lab, we implemented the Banker's Algorithm for deadlock avoidance in Java. The algorithm ensures that resources are allocated to processes safely, avoiding potential deadlocks. The output of the algorithm informs whether the system is in a safe state or not based on the given resource allocation and maximum demand of processes.

The Banker's Algorithm is an important approach to manage resource allocation in operating systems. It is widely used to prevent deadlocks in scenarios where multiple processes share resources. By performing a safety check before allocating resources, the algorithm ensures that processes can complete their tasks without being blocked indefinitely due to resource unavailability.

CONCLUSION:

The Banker's Algorithm is a powerful tool for avoiding deadlocks in operating systems. It ensures that resources are allocated safely to processes, preventing potential deadlocks and ensuring that the system remains in a safe state. By carefully managing resource requests and periodically performing a safety check, the algorithm maintains the stability and efficiency of the system.

In this lab, we successfully implemented the Banker's Algorithm in Java, demonstrated its usage with a sample scenario, and discussed its importance in preventing deadlocks. The knowledge gained from this lab will be valuable in understanding and dealing with resource allocation issues in real-world systems.