5. Deadlock Control Using Banker's Algorithm

Aim:

To implement deadlock control using Banker's algorithm

Program:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Iterator;

/**
 * Java program to demonstrate the working
 * of Banker's algorithm for Deadlock Avoidance.
 *
 * @author jimil
 */
public class BankersImpl {

	static ArrayList<Process> processes = new ArrayList<Process>();
	static ArrayList<Integer> available = new ArrayList<Integer>();
	static ArrayList<Integer> safeSequence = new ArrayList<Integer>();
	static int resourceCount, processCount;

	/**
	 * Driver function for the program
	 *
	 * @param args
	 */
	public static void main(String[] args) throws Exception {
		BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
		/**
		 * Entering the snapshot details
		 */
		System.out.print("Enter the number of resource types: ");
```

```
        resourceCount = Integer.parseInt(br.readLine());
        System.out.println("");
        System.out.print("Enter the number of processes: ");
        processCount = Integer.parseInt(br.readLine());
        System.out.println("");
        System.out.println("Enter the Snapshot Details");
        for(int i = 0; i < processCount; i++) {
              Process p = new Process();
              p.setId(i);
              p.setFinished(false);
              ArrayList<Integer> temp = new ArrayList<Integer>();
              ArrayList<Integer> temp1 = new ArrayList<Integer>();
              System.out.println("Enter the allocation matrix: ");
              for(int j = 0; j < resourceCount; j++) {
                    temp.add(Integer.parseInt(br.readLine()));
              }
              p.setAllocationMatrix(temp);
              System.out.println("Enter the max need matrix: ");
              for(int j = 0; j < resourceCount; j++) {
                    temp1.add(Integer.parseInt(br.readLine()));
              }
              p.setMaxMatrix(temp1);
              processes.add(p);
        }
        System.out.println("Enter current available matrix: ");
        for(int j = 0; j < resourceCount; j++) {
              available.add(Integer.parseInt(br.readLine()));
        }
        /**
         * Print Snapshot Details
         */
        printSnapshot();
        /**
         * Start Banker's
         */
        boolean result = isSnapshotSafe();
        if(result) {
              System.out.println("SAFE SEQUENCE: " + safeSequence);
        } else {
              System.out.println("DEADLOCK!");
        }
  }
```

```
	/**
	 * Utility method that prints the current snapshot that
	 * is entered by the user for the safety algorithm.
	 */
	public static void printSnapshot() {
		System.out.println("Snapshot Details: ");
		System.out.println("Process\tAllocated\tMax\tNeed");
		for(Process p: processes) {
			System.out.print(p.getId() + "\t");
			System.out.print(p.getAllocationMatrix());
			System.out.print("\t");
			System.out.print(p.getMaxMatrix());
			System.out.print("\t");
			System.out.print(p.getNeedMatrix());
			System.out.println();
		}

System.out.println("--------------------------------------------------");
		System.out.println("Available: " + available);
	}

	/**
	 * Implements Banker's Algorithm to check whether
	 * the given snapshot is in safe state or not.
	 *
	 * Safe state ensures that the given snapshot won't
	 * cause a deadlock.
	 * Unsafe state indicates a high probability of
	 * occurence of a deadlock in the given snapshot.
	 *
	 * @return
	 */
	public static boolean isSnapshotSafe() {
		boolean result = false;
		int count = 0;
		while(count < processCount) {
			for(Iterator<Process> processIterator =
processes.iterator(); processIterator.hasNext();) {
				Process p = processIterator.next();
				if(isNeedSatisfiable(p)) {
					p.setFinished(true);
```

```
                                    safeSequence.add(p.getId());
                                    /**
                                     * Re-calculating the new available
                                     */
                                    for(int i = 0; i < resourceCount; i++)
{
                                          available.set(i, available.get(i)
+ p.getAllocationMatrix().get(i));
                                    }
                                    processIterator.remove();
                        }
                  }
                  if(processes.size() == 0) {
                        result = true;
                        break;
                  }
                  count++;
            }
            return result;
      }

      /**
       * Checks whether the condition Need <= Available
       * is satisfied for the Process p
       *
       * @param p
       * @return
       */
      public static boolean isNeedSatisfiable(Process p) {
            for(int i = 0; i < resourceCount; i++) {
                  if(p.getNeedMatrix().get(i) > available.get(i)) {
                        return false;
                  }
            }
            return true;
      }

}

/**
 * A POJO class that models a process that is
 * currently in the snapshot.
```

```
 *
 * @author jimil
 */
class Process {
	private int id;
	private boolean isFinished;
	private ArrayList<Integer> allocationMatrix;
	private ArrayList<Integer> maxMatrix;
	private ArrayList<Integer> needMatrix;
	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public boolean isFinished() {
		return isFinished;
	}
	public void setFinished(boolean isFinished) {
		this.isFinished = isFinished;
	}
	public ArrayList<Integer> getAllocationMatrix() {
		return allocationMatrix;
	}
	public void setAllocationMatrix(ArrayList<Integer> allocationMatrix){
		this.allocationMatrix = allocationMatrix;
	}
	public ArrayList<Integer> getMaxMatrix() {
		return maxMatrix;
	}
	public void setMaxMatrix(ArrayList<Integer> maxMatrix) {
		this.maxMatrix = maxMatrix;
		this.calcNeedMatrix();
	}
	public ArrayList<Integer> getNeedMatrix() {
		return needMatrix;
	}
	private void calcNeedMatrix() {
		this.needMatrix = new ArrayList<Integer>();
		for(int i = 0; i < maxMatrix.size(); i++) {
			this.needMatrix.add((int)this.maxMatrix.get(i) -
(int)this.allocationMatrix.get(i));
```

```
            }
      }
}
```

Output:

Safe Sequence:

```
Enter the number of resource types: 3
Enter the number of processes: 5
Enter the Snapshot Details
Enter the allocation matrix:
0 1 0
Enter the max need matrix:
7 5 3
Enter the allocation matrix:
2 0 0
Enter the max need matrix:
3 2 2
Enter the allocation matrix:
3 0 2
Enter the max need matrix:
9 0 2
Enter the allocation matrix:
2 1 1
Enter the max need matrix:
2 2 2
Enter the allocation matrix:
0 2 2
Enter the max need matrix:
4 3 3
Enter current available matrix:
3 3 2
Snapshot Details:
Process     Allocated   Max         Need
0           [0, 1, 0]   [7, 5, 3]   [7, 4, 3]
1           [2, 0, 0]   [3, 2, 2]   [1, 2, 2]
2           [3, 0, 2]   [9, 0, 2]   [6, 0, 0]
3           [2, 1, 1]   [2, 2, 2]   [0, 1, 1]
4           [0, 2, 2]   [4, 3, 3]   [4, 1, 1]
--------------------------------------------------
Available: [3, 3, 2]
SAFE SEQUENCE: [1, 3, 4, 0, 2]
```

Deadlock:

```
Enter the number of resource types: 3
Enter the number of processes: 5
Enter the Snapshot Details
Enter the allocation matrix:
0 1 0
Enter the max need matrix:
7 5 3
Enter the allocation matrix:
2 0 0
Enter the max need matrix:
3 2 2
Enter the allocation matrix:
3 0 2
Enter the max need matrix:
9 0 2
Enter the allocation matrix:
2 1 1
Enter the max need matrix:
2 2 2
Enter the allocation matrix:
0 2 2
Enter the max need matrix:
4 3 3
Enter current available matrix:
0 0 0
Snapshot Details:
Process     Allocated   Max         Need
0           [0, 1, 0]   [7, 5, 3]   [7, 4, 3]
1           [2, 0, 0]   [3, 2, 2]   [1, 2, 2]
2           [3, 0, 2]   [9, 0, 2]   [6, 0, 0]
3           [2, 1, 1]   [2, 2, 2]   [0, 1, 1]
4           [0, 2, 2]   [4, 3, 3]   [4, 1, 1]
--------------------------------------------------
Available: [0, 0, 0]
DEADLOCK!
```

Conclusion:

Thus, I studied and understood the concept of deadlock in cooperative processes, and I understood and implemented the Banker's algorithm that is used for deadlock avoidance.