**Experiment No. 9**

**NAME:** Omkar Sunil Khanvilkar **ROLL NO:** 07

**CLASS:** TY\_IT\_B **BATCH**: 2

**DATE OF PERFORMANCE:** 26/09/2024

**Write a program to check whether given system is in safe state or not using Banker’s Deadlock Avoidance algorithm.**

## **CODE:**

import java.util.\*;  
  
import static java.lang.System.*exit*;  
  
public class BankersAlgorithm {  
 int n; // number of processes  
 int m; // types of resources  
 int [] totalResources; // total resources vector  
 int [] available; // available vector  
 int [][] allocation; // resource allocation matrix  
 int [][] max; // Max matrix  
 int [][] need; // need matrix (need = total - allocated)  
  
 public static void main(String[] args) {  
 System.*out*.println("\n-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*- Banker's Algorithm for Deadlock Avoidance \*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*-\*\n");  
 BankersAlgorithm algorithm = new BankersAlgorithm();  
 algorithm.init();  
 }  
  
 public static void printMatrix(int [][] matrix){  
 for (int[] row : matrix) {  
 for (int el : row) {  
 System.*out*.print(el + " ");  
 }  
 System.*out*.println();  
 }  
 }  
 private void init(){  
  
 Scanner sc = new Scanner(System.*in*);  
 System.*out*.println("Enter total number of processes : ");  
 n = sc.nextInt();  
 System.*out*.println("Enter total types of resources : ");  
 m = sc.nextInt();  
  
 // initialization  
 totalResources = new int[m];  
 available = new int[m];  
 allocation = new int[n][m];  
 max = new int[n][m];  
 need = new int[n][m];  
  
 // Taking total resources vector from user  
 System.*out*.println("Enter total resources vector (separated by space) : ");  
 for(int i = 0; i < m; i++){  
 totalResources[i] = sc.nextInt();  
 }  
  
 // Filling allocation matrix by taking user input  
 for(int i = 0; i < n; i++){  
 System.*out*.println("Enter allocation vector for Process P"+ i +" :");  
 for (int j = 0; j < m; j++){  
 allocation[i][j] = sc.nextInt();  
 }  
 }  
  
 // Calculating Available vector. (available = total - allocated)  
 int[] allocated = new int[m];  
 for (int k = 0; k < m; k++) {  
 for (int[] row : allocation) {  
 allocated[k] += row[k];  
 }  
 }  
  
 // Validation : if allocated > total then exit with error message  
 for(int i = 0; i < m; i++){  
 if(allocated[i] > totalResources[i]){  
 System.*out*.println("Invalid resources allocation !!");  
 *exit*(1);  
 }  
 }  
  
 // If allocation is valid, calculate available vector.  
 for(int i = 0; i < m; i++){  
 available[i] = totalResources[i] - allocated[i];  
 }  
  
  
 // Take max matrix from the user  
 for(int i = 0; i < n; i++){  
 System.*out*.println("Enter Max vector for Process P"+ i +" :");  
 for (int j = 0; j < m; j++){  
 max[i][j] = sc.nextInt();  
 }  
 }  
  
 // Calculate need vector. (need = max - allocation)  
 for(int i = 0; i < n; i++){  
 for (int j = 0; j < m; j++){  
 need[i][j] = max[i][j] - allocation[i][j];  
 }  
 }  
 System.*out*.println("Need Matrix : ");  
 *printMatrix*(need);  
  
 checkForSafeSequence(need,available,allocation);  
 sc.close();  
 }  
  
 private void checkForSafeSequence(int [][] need, int [] available,int [][] allocation){  
 boolean [] flag = new boolean[n]; // to keep track of whether the process is completely executed or not.  
 List<String> safeSequence = new ArrayList<>(); // to store safe sequence  
  
   
 while (atLeastOneProcessCanBeExecuted(need, available, flag) != -1){  
 int idx = atLeastOneProcessCanBeExecuted(need, available, flag);  
 for(int j = 0; j < m; j++){  
 available[j] += allocation[idx][j];  
 allocation[idx][j] = 0;  
 }  
 flag[idx] = true;  
 safeSequence.add("P"+idx);  
 }  
  
 // If all processes are marked as true means they are done executing, we got the safe sequence.  
  
 System.*out*.println("Available Vector : " + Arrays.*toString*(available));  
  
 if(safeSequence.size() == n){  
 System.*out*.println("Safe sequence : "+ safeSequence);  
 System.*out*.println("Allocation Matrix : ");  
 *printMatrix*(allocation);  
 }  
 else{  
 System.*out*.println("DEADLOCK!!");  
 }  
 }  
  
 private int atLeastOneProcessCanBeExecuted(int[][] need, int[] available,boolean []flag) {  
 for(int i = 0; i < n; i++){  
 if(!flag[i] && canComplete(need[i],available)){  
 return i;  
 }  
 }  
 return -1;  
 }  
  
 private boolean canComplete(int[] currNeed, int[] available) {  
 for (int i = 0; i < m; i++){  
 if(currNeed[i] > available[i]){  
 return false;  
 }  
 }  
 return true;  
 }  
  
}

## **OUTPUT:**

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated A black rectangle with white dots

Description automatically generated