USCS303 – OS: Practical – 06: Banker's Algorithm

Practical Aim: Banker's Algorithm

Banker's Algorithm

The **resource-allocation-graph-algorithm** is not applicable to a resource allocation system with multiple instances of each resource type.

The **deadlock-avoidance algorithm** that we describe next is applicable to such a system but is less efficient than the resource-allocation graph scheme.

Banker's algorithm is a **deadlock avoidance algorithm**. It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not.

The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

Banker's Algorithm – how it works?

Consider there are **n** account holders in a bank and the money in all of their accounts is **S**.

Every time a loan has to be granted by the bank, it subtracts the **load amount** from the **total money** the bank has.

When a new thread enters the system, it must declare the maximum number of instances of each resource type that it may need.

This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.

Data Structures – Banker's Algorithm

Several data structure must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where n is the number of threads in the system and m is the number of resource types:

Available: A vector of length m indicates the number of available resources of each type. If **Available[j]** equals k, then k instances of resource type Rj are available.

Max: An n x m matrix defines the maximum demand of each thread. If **Max[i][j]** equals k, then thread Ti may request at most k instances of resource type Rj.

Allocation: An n x m matrix defines the number of resources of each type currently allocated to each thread. If **Allocation[i][j]** equals k, then thread Ti is currently allocated k instances of resource type Rj.

Need: An n x m matrix indicates the remaining resource need of each thread. If **Need[i][j]** equals k, then thread Ti may need k more instances of resource type Rj to complete its task.

Need[i][j] = Max[i][j] – Allocation[i][j]

Algorithm

Safety Algorithm

Step 1: Let **Work** and **Finish** be vectors of length m and n, respectively. Initialize **Work=Available** and **Finish[i]=false** for i=0, 1, …, n-1.

Step 2: Find an index I such that both

Step 2.1: Finish[i] == false

Step 2.2: Need <= Work

If no such I exists, go to **Step 4.**

Step 3: Work = work + allocation;

Finish[i] = true

Go to **step 2.**

Step 4: If Finish[i] == true for all i, then the system is in a safe state.

Resource – Request Algorithm

Let **Requesti** be the request vector for thread Ti.

If **Requesti[j] == k**, then thread Ti wants k instances of resource type Rj.

When a request for resources is made by thread Ti, the following actions are taken:

Step 1: If **Requesti <= Needi**, go to **step 2.** Otherwise, raise an error condition, since the thread has exceeded its maximum claim.

Step 2: If **Requesti <= Available**, go to **Step 3**. Otherwise, Ti must wait, since the resources are not available.

Step 3: Have the system pretend to have allocated the requested resources to thread Ti by modifying the state as follows:

Available = Available – Requesti

Allocationi = Allocationi + Requesti

Needi = Needi – Requesti

If the resulting resource-allocation state is safe, the transaction is completed, and thread Ti is allocated its resources. However, if the new state is unsafe, then Ti must wait for **Requesti,** and the old resource-allocation state is restored.

Solved Example

Example 1: Consider a system with five threads To through T4 and three resource types A, B and C. Resource type A has ten instances, resource type B has five instances, and resource type C has seven instances. Suppose that the following snapshot represents the current state of the system:

Threads	Allocation			Max			Available		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
T0	0	1	0	7	5	3	3	3	2
T1	2	0	0	3	2	2			
T2	3	0	2	9	0	2			
T3	2	1	1	2	2	2			
T4	0	0	2	4	3	3			

Solve:

Need Matrix = Max – Allocation

Threads	Allocation			Max			Available			Need		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
T0	0	1	0	7	5	3	3	3	2	7	4	3
T1	2	0	0	3	2	2				1	2	2
T2	3	0	2	9	0	2				6	0	0
T3	2	1	1	2	2	2				0	1	1
T4	0	0	2	4	3	3				4	3	1

We claim that the system is currently in a **safe state**.

Indeed, the sequence **<t1, t3, t4, t0, t2>** satisfies the safety criteria.

Example 2: Consider the following System

Threads	Allocation			Max			Available		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P0	1	1	1	4	3	3	2	1	0
P1	2	1	2	3	2	2			
P2	4	0	1	9	0	2			
P3	0	2	0	7	5	3			
P4	1	1	2	1	1	2			

Solve:

Need Matrix = Max - Allocation

Threads	Allocation			Max			Available			Need		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P0	1	1	1	4	3	3	2	1	0	7	4	3
P1	2	1	2	3	2	2				1	2	2
P2	4	0	1	9	0	2				6	0	0
P3	0	2	0	7	5	3				0	1	1
P4	1	1	2	1	1	2				4	3	1

We claim that the system of current in safe state in the sequence <P1, P4, P0, P2, P3> satisfy the supplies criteria.

Example 3: Consider the following example containing five processes and 4 type of resources:

Threads	Allocation				Max				Available			
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P0	1	1	1	0	0	2	1	0	1	5	2	0
P1	2	1	2	1	1	6	5	2				
P2	4	0	1	5	2	3	6	6				
P3	0	2	0	2	0	6	5	2				
P4	1	1	2	4	0	6	5	6				

Solve:

Need matrix = Max – Allocation

Threads	Allocation				Max				Available				Need			
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P0	1	1	1	0	0	2	1	0	1	5	2	0	0	1	0	0
P1	2	1	2	1	1	6	5	2					0	4	2	1
P2	4	0	1	5	2	3	6	6					1	0	0	1
P3	0	2	0	2	0	6	5	2					0	2	2	0
P4	1	1	2	4	0	6	5	6					0	6	4	2

We claim that the system of current in safe state in the sequence P0>P3>P4>P1>P2 satisfy the supplies criteria.

Implementation

Code:

```
// Name: Sumit Telawane
// Batch : B1
// PRN: 2020016400840595
// Date: 21st August, 2021
// Prac-06: Banker's Algorithm

import java.util.Scanner;

public class P6_BankersAlgo_ST {
	private int need[][], allocate[][], max[][], avail[][], np, nr;

	private void input() {
		Scanner sc = new Scanner(System.in);
		System.out.print("Enter no. of Process: ");
		np = sc.nextInt();		// no of processes
		System.out.print("Enter no. of resources: ");
		nr = sc.nextInt();		// no of resources

		// initializing arrays
		need = new int[np][nr];
		max = new int[np][nr];
		allocate = new int[np][nr];
		avail = new int[1][nr];

		for (int i=0; i<np; i++) {
```

```
                System.out.print("Enter allocation matrix for Process P " + i + ": ");
                for (int j=0; j<nr;j++)
                        allocate[i][j] = sc.nextInt();     // allocation matrix
        }

        for (int i=0; i<np; i++) {
                System.out.print("Enter maximum matrix for process P " + i + ": ");
                for (int j=0; j<nr; j++)
                        max[i][j] = sc.nextInt();                  // max matrix
        }

        System.out.print("Enter available matrix for processs P0: ");
        for (int j=0; j<nr;j++)
                avail[0][j] = sc.nextInt();                        // available matrix

        sc.close();
}       // input() ends

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

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

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

		// printing Need Matrix
		System.out.println("================ Need Matrix
================");
		for (int a=0; a<np; a++) {
			for (int b=0; b<nr; b++) {
				System.out.print(need[a][b] + "\t");
			}
			System.out.println();
		}

		System.out.println("Allocated process: ");
		// until all process allocated
```

```
		while(j<np) {
			boolean allocated = false;


			for (int i=0; i<np; i++) {
				// trying to allocate
				if (!done[i] && check(i)) {
					for(int k=0; k<nr; k++)
						avail[0][k]=avail[0][k]-need[i][k]+max[i][k];
					System.out.print("P"+i+">");
					allocated=done[i]=true;
					j++;
				}	// if block


			}	// for ends
			if(!allocated)
				break;  // if no allocation
		}	// while ends
		if(j==np)	// if all processes are allocated
			System.out.println("\nSafely allocated");
		else
			System.out.println("All/Remaining process can\'t be allocated Safely");
	}	// isSafe() ends
	public static void main(String[] args) {
		new P6_BankersAlgo_ST().isSafe();
	}
}
```

Input

Input question 1:

```
D:\os\BATCH_B1\USCS3P01_USCS303_OS_B1_P6>javac .\P6_BankersAlgo_ST.java

D:\os\BATCH_B1\USCS3P01_USCS303_OS_B1_P6>java P6_BankersAlgo_ST
Enter no. of Process: 5
Enter no. of resources: 3
Enter allocation matrix for Process P 0: 0 1 0
Enter allocation matrix for Process P 1: 2 0 0
Enter allocation matrix for Process P 2: 3 0 2
Enter allocation matrix for Process P 3: 2 1 1
Enter allocation matrix for Process P 4: 0 0 2
Enter maximum matrix for process P 0: 7 5 3
Enter maximum matrix for process P 1: 3 2 2
Enter maximum matrix for process P 2: 9 0 2
Enter maximum matrix for process P 3: 2 2 2
Enter maximum matrix for process P 4: 4 3 3
Enter available matrix for processs P0: 3 3 2
```

Input question 2:

```
D:\os\BATCH_B1\USCS3P01_USCS303_OS_B1_P6>
D:\os\BATCH_B1\USCS3P01_USCS303_OS_B1_P6>java P6_BankersAlgo_ST
Enter no. of Process: 5
Enter no. of resources: 3
Enter allocation matrix for Process P 0: 1 1 2
Enter allocation matrix for Process P 1: 2 1 2
Enter allocation matrix for Process P 2: 4 0 1
Enter allocation matrix for Process P 3: 0 2 0
Enter allocation matrix for Process P 4: 1 1 2
Enter maximum matrix for process P 0: 4 3 3
Enter maximum matrix for process P 1: 3 2 2
Enter maximum matrix for process P 2: 9 0 2
Enter maximum matrix for process P 3: 7 5 3
Enter maximum matrix for process P 4: 1 1 2
Enter available matrix for processs P0: 2 1 0
```

Input question 3:

```
D:\os\BATCH_B1\USCS3P01_USCS303_OS_B1_P6>java P6_BankersAlgo_ST
Enter no. of Process: 5
Enter no. of resources: 4
Enter allocation matrix for Process P 0: 0 1 1 0
Enter allocation matrix for Process P 1: 1 2 3 1
Enter allocation matrix for Process P 2: 0 6 3 2
Enter allocation matrix for Process P 3: 0 0 1 4
Enter allocation matrix for Process P 4: 1 3 6 5
Enter maximum matrix for process P 0: 0 2 1 0
Enter maximum matrix for process P 1: 1 6 5 2
Enter maximum matrix for process P 2: 0 6 5 2
Enter maximum matrix for process P 3: 0 6 5 6
Enter maximum matrix for process P 4: 2 3 6 6
Enter available matrix for processs P0: 1 5 2 0
```

Output

Sample Output 1:

```
================ Need Matrix ================
7       4       3
1       2       2
6       0       0
0       1       1
4       3       1
Allocated process:
P1>P3>P4>P0>P2>
Safely allocated
```

Sample Output 2:

```
================ Need Matrix ================
3       2       1
1       1       0
5       0       1
7       3       3
0       0       0
Allocated process:
P1>P4>P0>P2>P3>
Safely allocated
```

Sample Output 3:

```
================ Need Matrix ================
0       1       0       0
0       4       2       1
0       0       2       0
0       6       4       2
1       0       0       1
Allocated process:
P0>P2>P3>P4>P1>
Safely allocated
```