SMT. CHANDIBAI HIMATHMAL MASUKHANI COLLEGE

Contents

SMT. CHANDIBAI HIMATHMAL MASUKHANI COLLEGE

USCSP301_USCS303_Operating System (OS) Practical-06

Practical 06: Banker's Algorithm

Practical Date: 21 August 2021

Practical Aim: Data Structures (Banker's Algorithm)

Banker's Algorithm

(i) The **resource-allocation-graph algorithm** is not applicable to a resource allocation system with multiple instance of each resource type.

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

(iii) The algorithm I commonly known as the as the **banker's algorithm.**

(iv) Banker's Algorithm is a **deadlock avoidance algorithm.**

(v) It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not.

(vi)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.**

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

(viii) This number may not exceed the total number of resources in the system.

(ix) 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.

(x) If it will, the resources are allocated ; otherwise, the thread must wait until some other thread releases enough resources.

Data Structures required in Banker's Algorithm

(i) Several data structures must be maintained to implement the banker's algorithm.

(ii) These data structures encode the state of the resource-allocation system.

(iii) We need the following data structures, where n is number of threads in the system and m is the number of resource types:

Data Structures

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 resources 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 **Request**, be the request vector for thread Ti.

If **Request, [j] == k**, then thread T, 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<=Need,** go to **Step 2.** Otherwise, raise an error condition, since the thread has exceeded its maximum claim.

Step 2: I**f 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 EXAMPLES:

Consider a system with five threads T0 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	

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:

Processes	Allocation	Max	Available
	A B C	A B C	A B C
P0	1 1 2	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	

Need Matrix = Max - Allocation

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

EXAMPLE 3:

Consider the following example containing five processes and 4 types of resources.

Calculate the Need Matrix and the sequence of safety allocation.

Given Matrices			
	Allocation Matrix (NO of the allocated resource By a process)	Max Matrix (Max resources that may be used by a process)	Available Matrix (Not allowed Resources)

	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	1	1	0	0	2	1	0	1	5	2	0
P1	1	2	3	1	1	6	5	2				
P2	1	3	6	5	2	3	6	6				
P3	0	6	3	2	0	6	5	2				
P4	0	0	1	4	0	6	5	6				

QUESTION:
Write a java program that implements the banker's algorithm.

IMPLEMENTATION

```
//Name: Ritika Sahu

//Batch : B1

//PRN:2020016400783543

//Date: 21 August, 2021.

//Practical 6: Banker's Algorithm


import java.util.Scanner;


public class P6_BankersAlgo_RS {

   private int need[][], allocate[][], max[][], avail[][], np, nr;


private void input() {

  Scanner sc = new Scanner(System.in);

  System.out.printIn("Enter no. of processes:");

  np = sc.nextInt(); // no.of resources
```

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

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

for (int i = 0;i < np;i++) {
       System.out.printIn("Enter maximum matrix for process P" + i + ":");
       for (int j = 0; j < nr;j++)
            max[i][j] = sc.nextInt(); // max matrix
}
System.out.printIn("Enter available matrix for process P0:");
for (int j = 0; j < n;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 < np;j++) // calculating need matrix
            need[i][j] = max [i][j] - allocation[i][j];


  return need;
} // calc_need() ends
private boolean check(int i)  {
   // checking f 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.printIn("========Need Matrix==========");
     for (int a = 0;a < np;a++) {
          for(int b = 0;b < nr;b++) {
               System.out.printIn(need[a][b] + "\t");
           }
           System.out.printIn();
     }
```

```
 System.out.printIn("Allocated process:);


 while (j < np) ( // until all proces 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[0][k] = avail[0][k] - need[i][k] + max[i][k];
           System.out.printIn("P" + i + " > ");
           allocated = done[i] = true;
           j++;
           } // if block
         if (!allocated)
            break; // if no allocation
} // while ends
 if (i == np) // if all processes are allocated
   System.out.printIn("\nSafely allocated");
 else
  System.out.printIn("All/Remaining process can\'t be allocated safely");
 } // isSafe() ends


 public static void main(String[] args) {
       new P6_BankersAlgo_RS.isSafe();
 }
} // class ends
```

INPUT:

```
F:\USCSP301_USCS303_OS_B0\Prac_06_Banker_16_08_2021>javac P6_BankersAlgo_NR.java

F:\USCSP301_USCS303_OS_B0\Prac_06_Banker_16_08_2021>java P6_BankersAlgo_NR
Enter no. of processes : 5
Enter no. of resources : 3
Enter allocation matrix for process P0: 0 1 0
Enter allocation matrix for process P1: 2 0 0
Enter allocation matrix for process P2: 3 0 2
Enter allocation matrix for process P3: 2 1 1
Enter allocation matrix for process P4: 0 0 2
Enter maximum matrix for process P0: 7 5 3
Enter maximum matrix for process P1: 3 2 2
Enter maximum matrix for process P2: 9 0 2
Enter maximum matrix for process P3: 2 2 2
Enter maximum matrix for process P4: 4 3 3
Enter available matrix for process P0: 3 3 2
```

OUTPUT:

```
========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:

SAMPLE OUTPUT 1:

```
Enter no. of processes : 5
Enter no. of resources : 3
Enter allocation matrix for process P0: 0 1 0
Enter allocation matrix for process P1: 2 0 0
Enter allocation matrix for process P2: 3 0 2
Enter allocation matrix for process P3: 2 1 1
Enter allocation matrix for process P4: 0 0 2
Enter maximum matrix for process P0: 7 5 3
Enter maximum matrix for process P1: 3 2 2
Enter maximum matrix for process P2: 9 0 2
Enter maximum matrix for process P3: 2 2 2
Enter maximum matrix for process P4: 4 3 3
Enter available matrix for process P0: 3 3 2
========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:

```
Enter no. of processes : 5
Enter no. of resources : 3
Enter allocation matrix for process P0: 1 1 2
Enter allocation matrix for process P1: 2 1 2
Enter allocation matrix for process P2: 4 0 1
Enter allocation matrix for process P3: 0 2 0
Enter allocation matrix for process P4: 1 1 2
Enter maximum matrix for process P0: 4 3 3
Enter maximum matrix for process P1: 3 2 2
Enter maximum matrix for process P2: 9 0 2
Enter maximum matrix for process P3: 7 5 3
Enter maximum matrix for process P4: 1 1 2
Enter available matrix for process P0: 2 1 0
========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
```

SMT. CHANDIBAI HIMATHMAL MASUKHANI COLLEGE

SAMPLE OUTPUT 3:

```
Enter no. of processes : 5
Enter no. of resources : 3
Enter allocation matrix for process P0: 1 1 2
Enter allocation matrix for process P1: 2 1 2
Enter allocation matrix for process P2: 4 0 1
Enter allocation matrix for process P3: 0 2 0
Enter allocation matrix for process P4: 1 1 2
Enter maximum matrix for process P0: 4 3 3
Enter maximum matrix for process P1: 3 2 2
Enter maximum matrix for process P2: 9 0 2
Enter maximum matrix for process P3: 7 5 3
Enter maximum matrix for process P4: 1 1 2
Enter available matrix for process P0: 2 1 0
========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
```