Contents

USCSP301-USCS303: Operating System(OS) Practical-06

Practical-06: Banker’s Algorithm

Practical Date: 21st Aug, 2021

Practical Aim: Banker’s Algorithm

* Banker’s Algorithm.............................................................................................................................1
* Data Structures required in Banker’s Algorithm.................................................................................2
* Algorithm(Both-Safety and Resource-Allocation)................................................................................3
* Solved Example.....................................................................................................................................4
* Question................................................................................................................................................5
* Implementaion......................................................................................................................................6
* Input.......................................................................................................................................................7
* Output.....................................................................................................................................................8
* Sample Output........................................................................................................................................9
* Content:
* For the banker’s algorithm to operate, each process has to a priority specify its maximum requirement of resources.
* Process:
* One can find out whether the system is in the safe state or not.
* One can also determine whether a process’s request for allocation of resources be safely granted immediately.
* Prior Knowledge:
* Data Structure used in bankers algorithm.
* Safety algorithm and resource request 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.
* This algorithm is commonly known as the **banker’s algorithm.**
* 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.

# How it works?

* Consider there are n account holders in a bank and the sum of the money in all of their accounts is S.
* Every time a loan has to be granted by the bank, it subtracts the loan amount from the total money the bank has.
* Then it checks if that difference is greater than S.
* It is done because, only then, the bank would have enough money even if all the n account holders draw all their money at once.

# Data Structures (Banker’s Algorithm)

* Several data structures 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 indicate the number of available resources of each type. If **Available[j]**

equals k, then k instances of resources 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]**

# 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 2.3:** **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 i [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 **Request i, <\_ Need i**, go to **Step 2**. Otherwise, raise an error condition, since the thread has exceeded its maximum claim.

**Step 2:** If **Request i, <\_ Available**, go to **Step 3.** Otherwise, T i , must wait, since the resources are not available.

**Step 3:** Have the system pretend to have allocated the requested resources to thread T i , by modifiying the state as follows:

**Available = Available – Request i**

**Allocation i = Allocation i + Request i**

**Need i = Need i – Request i**

If the resulting resource-allocation sate 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 **Request**, and the old resource-allocation state is restored.

# Example

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:

|  |  |  |  |
| --- | --- | --- | --- |
| **Thread** | **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

|  |  |  |  |
| --- | --- | --- | --- |
| **Thread** | **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**.

**Question – 01**

Write a Java Program that implements the blanker’s algorithm.

Implementation: Implement Banker’s Algorithm in Java

FileName: P6\_BankersAlgo\_SS.java

Source Code:

import java.util.Scanner;

public class P6\_BankersAlgo\_SS{

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 process

System.out.print("Enter no. of resources: ");

nr= sc.nextInt(); //no. of resources

need= new int[np][nr]; //initializing arrays

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 process PO: ");

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 nrrd 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: ");

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[0][k]= avail[0][k]- need[i][k] + max[i][k];

System.out.print("P" +i+ ">");

allocated= done[i]= true;

j++;

}// if block

if(!allocated)

break; // if no alloccation

} //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

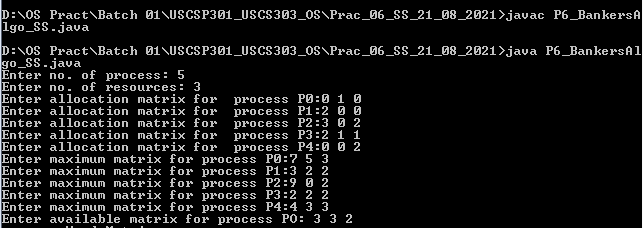
public static void main(String[] args){

new P6\_BankersAlgo\_SS().isSafe();

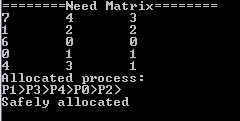
}

} //class ends

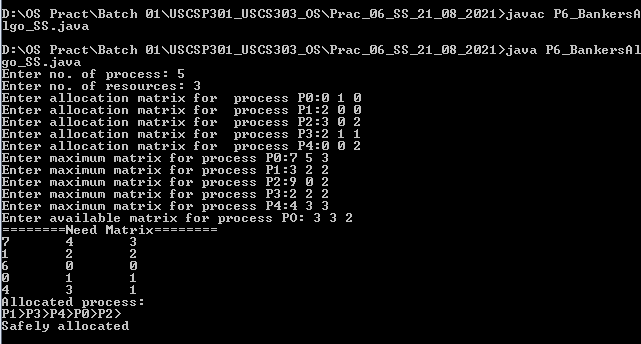
**Input:**

****

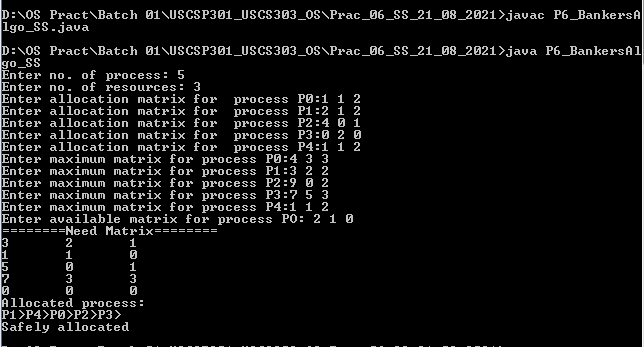
**Output:**

****

**Sample Output – 01:**

****

**Sample Output – 02:**

****