Experiment No 7
Class: SE Comp
Year: 2020-21
Performed by: Danyl Fernandes, 72

Aim: Write a program to demonstrate the concept of deadlock avoidance through Banker's Algorithm.

Theory:

Banker's Algorithm:

- The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.
- Banker's algorithm is named so because it is used in the banking system to check whether a loan can be sanctioned to a person or not.
- Suppose there are n number of account holders in a bank and the total sum of their money is S.
- If a person applies for a loan then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than S then only the loan is sanctioned.
- It is done because if all the account holders come to withdraw their money then the bank can easily do it.
- In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always.

Let 'n' be the number of processes in the system and 'm' be the number of resources types, then:

Available:

- It is a 1-d array of size 'm' indicating the number of available resources of each type.
- Available[j] = k means there are 'k' instances of resource type Rj

Max:

- It is a 2-d array of size 'n*m' that defines the maximum demand of each process in a system.
- Max[i, j] = k means process Pi may request at most 'k' instances of resource type Rj.

Allocation:

- It is a 2-d array of size ‘n*m’ that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process Pi is currently allocated ‘k’ instances of resource type Rj

Need:

- It is a 2-d array of size ‘n*m’ that indicates the remaining resource need of each process.
- Need [i,   j] = k means process Pi currently needs ‘k’ instances of resource type Rj for its execution.
- Need [i,   j] = Max [i,   j] – Allocation [i,   j]

Banker’s algorithm consists of Safety algorithm and Resource request algorithm:

Safety Algorithm:

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1. Let Work and Finish be vectors of length ‘m’ and ‘n’ respectively.
    a. Initialize: Work = Available
    b. Finish[i] = false; for i=1, 2, 3, 4….n
2. Find an i such that both
    a. Finish[i] = false
    b. Needi <= Work
    c. if no such i exists goto step (4)
3. Work = Work + Allocation[i]
    a. Finish[i] = true
    b. goto step (2)
4. if Finish [i] = true for all i
    a. then the system is in a safe state

Resource-Request Algorithm:

Let Requesti be the request array for process Pi. Requesti [j] = k means process Pi wants k instances of resource type Rj. When a request for resources is made by process Pi, the following actions are taken:

1. 1) If Requesti <= Needi
    a. Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If Requesti <= Available
    a. Goto step (3); otherwise, Pi must wait, since the resources are not available.

3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:
    a. Available = Available – Requesti
    b. Allocationi = Allocationi + Requesti
    c. Needi = Needi– Requesti

Conclusion:

- In this experiment, we were successfully able to demonstrate the concept of deadlock avoidance through Banker’s Algorithm.

Implementation:

```
public class BankersAlgorithm {
    int n = 5;
    int m = 3;
    int need[][] = new int[n][m];
    int [][]max;
    int [][]alloc;
    int []avail;
    int safeSequence[] = new int[n];

    void initializeValues() {
     alloc = new int[][] {
          { 0, 1, 0 },
          { 2, 0, 0 },
          { 3, 0, 2 },
          { 2, 1, 1 },
          { 0, 0, 2 }
     };

     max = new int[][] {
          { 7, 5, 3 },
          { 3, 2, 2 },
          { 9, 0, 2 },
          { 2, 2, 2 },
          { 4, 3, 3 }
     };

     avail = new int[] { 3, 3, 2 };
    }

    void isSafe() {
    int count=0;

    boolean visited[] = new boolean[n];

     for (int i = 0;i < n; i++) {
        visited[i] = false;
    }

    int work[] = new int[m];
    for (int i = 0;i < m; i++) {
```

```
        work[i] = avail[i];
    }


    while (count<n) {
        boolean flag = false;
        for (int i = 0;i < n; i++) {
            if (visited[i] == false) {
            int j;
            for (j = 0;j < m; j++) {
                if (need[i][j] > work[j])
                break;
            }
            if (j == m) {
               safeSequence[count++]=i;
               visited[i]=true;
               flag=true;

                for (j = 0;j < m; j++) {
                work[j] = work[j]+alloc[i][j];
                }
            }
             }
        }
        if (flag == false) {
            break;
        }
    }
    if (count < n) {
        System.out.println("The System is UnSafe!");
    }
    else {
          System.out.println("Following is the SAFE Sequence");
                for (int i = 0;i < n; i++) {
            System.out.print("P" + safeSequence[i]);
            if (i != n-1)
            System.out.print(" -> ");
        }
    }
    }
```

```
void calculateNeed() {
    for (int i = 0;i < n; i++) {
        for (int j = 0;j < m; j++) {
        need[i][j] = max[i][j]-alloc[i][j];
         }
    }
    }

    public static void main(String[] args) {
      BankersAlgorithm bAlg = new BankersAlgorithm ();

      bAlg.initializeValues();
      bAlg.calculateNeed();
      bAlg.isSafe();
    }
}
```

Output:

```
Following is the SAFE Sequence
P1 → P3 → P4 → P0 → P2
```