GoodLuck | Page No.
Date

Assignment - C2

Title :- Implementation of Banker's Algorithm

Problem Statement :-

Write a Java program to implement Banker's algorithm.

Objective :-

- To study the algorithm for finding out whether a system is in a safe state.
- To study the resource request algorithm for deadlock avoidance.
- To study & implement Banker's algorithm to avoid deadlock.

Outcome :-

students will be able to,

- Implement deadlock aveidance algorithm
- Compute resource allocation sequences which lead to safe state
- Demonstrate limitations of deadlock avoidance algorithm

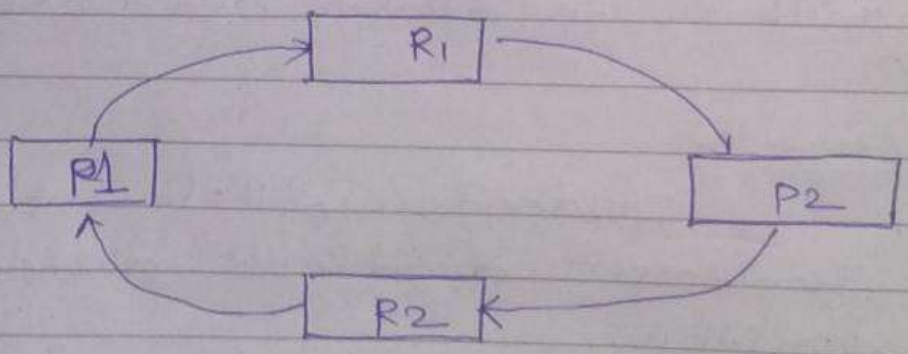
S/W & H/W requirements :- 64 bit open source linux, eclipse IDE, Java, I3 & I5 machines

Theory:-

Deadlock :- A set of processes is in deadlock state when every processes in the set is waiting for an event that can only cause by another process in the set. ex. resource acquisition & release.

As shown in diagram process P1 is holding the resource P2 & requesting R1.

Process P2 is holding resource R1 & requesting resource R2. So no process can proceed further, indicating deadlock.



Four Basic conditions for deadlock to happen :

1] mutual exclusion :- At least one resource must be held in non-sharable mode.

2] hold & wait :- there must be process holding one resource & waiting for another.

3] No premption :- resources cannot be preempted

4] Circular wait :- there must be exist a set of processess [P1, P2, ...] such that P1 is waiting for P2, P2 for P3, & so on & Pn waits for P1

Approaches to used to handle deadlock :-
- Deadlock Avoidance
- Deadlock prevention
- Deadlock Detection & Recovery.

Banker's Algorithm :-

It is deadlock avoidance algorithm. The name was chosen since this algorithm can be used in banking system to ensure that the bank never allocates it's available cash in such a way that it can no longer satisfy feathers request for cash.

When a new process 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. The resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Algorithm :-

1] If request [i] > need [i] then error (asked too much)
2] If request [i] > avail [i] then wait (can't supply now)
3] Resources available to satisfy request :

Lets assume that we satisfy request, then we would have :

available = available - request [i]
allocation [i] = allocation [i] + request [i]
need [i] = need [i] - request [i]

Now, check if this would leave is in safe state; if yes, grant request if no, leave the state as is & process to wait

Conclusion :-

We have learnt & successfully implemented the Banker's Algorithm.

```
package Banker;
import java.util.Scanner;

public class Bankers_Algo {


   public static void main(String args[])
   {
      int n,m,available[],max[][],allocation[][],need[][],work[];
      boolean finish[];
      Scanner reader = new Scanner(System.in);

      System.out.print("Enter number of processes: ");
      n = reader.nextInt();
      System.out.print("Enter number of resources: ");
      m = reader.nextInt();

      available = new int[m];
      max = new int[n][m];
      allocation = new int[n][m];
      need = new int[n][m];
      finish = new boolean[n];
      work = new int[m];

      System.out.println("Enter the available resources: ");
      for(int i=0;i<m;i++){
         available[i] = reader.nextInt();
         work[i] = available[i];
      }

      System.out.println("Enter the Max matrix: ");
      acceptInput(max,n,m);

      System.out.println("Enter the allocation matrix: ");
      acceptInput(allocation,n,m);

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

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

      int safeseq[] = new int[n];
      int count = 0;

      while(count < n)
      {
         boolean flag = false;

         for(int i=0;i<n;i++)
```

```
        {

            int j;
            if(finish[i] == false)
            {
                for(j=0;j<m;j++)
                {
                    if(need[i][j]>work[j])
                    {
                        break;
                    }
                }
                if(j == m)
                {
                    safeseq[count++] = i;
                    finish[i] = true;
                    flag = true;

                    for(j=0;j<m;j++)
                    {
                        work[j] = work[j] + allocation[i][j];
                    }
                }
            }
        }
        if(flag == false)
        {
            break;
        }

    }
    if(count < n)
    {
        System.out.println("System is unsafe");
    }

    else
    {
        System.out.println("Safe sequence is: ");
        for(int i=0;i<n;i++)
        {
            System.out.print("P"+(safeseq[i] + 1 )+"\t");
        }
    }
}

public static void acceptInput(int matrix[][], int rows, int cols)
{
    Scanner reader = new Scanner(System.in);
    for(int i=0;i<rows;i++)
    {
        for(int j=0;j<cols;j++)
        {
            int a = reader.nextInt();
```

```
            matrix[i][j] = a;
        }

    }

  }
}
```

```
Enter number of processes: 5
Enter number of resources: 4
Enter the available resources:
1 5 2 0
Enter the Max matrix:
0 0 1 2
1 7 5 0
2 3 5 6
0 6 5 2
0 6 5 6
Enter the allocation matrix:
0 0 1 2
1 0 0 0
1 3 5 4
0 6 3 2
0 0 1 4
Safe sequence is:
P1 P3 P4 P5 P2
```

```
Enter number of processes: 5
Enter number of resources: 3
Enter the available resources:
3
3
2
Enter the Max matrix:
7 5 3
3 2 2
9 0 2
4 2 2
5 3 3
Enter the allocation matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Safe sequence is:
P2 P4 P5 P1 P3
```