Assignment 10
Implementing Banker's algorithm
Submission Date: 22nd November 2015, 9:00 pm

Introduction: This is an extra credit assignment. You have to implement Banker's algorithm as discussed in the class. For this project, you will write a multithreaded program that implements the bankers algorithm discussed in Section 7.5.3 of the textbook. Several customers (threads) request and release resources from the bank. The banker (a decentralized scheduler) will grant a request only if it leaves the system in a safe state. A request that leaves the system in an unsafe state will be denied. This programming assignment combines three separate topics: (1) multithreading, (2) preventing race conditions, and (3) deadlock avoidance.

In the following description, **we have used thread & processes interchangeably**.

User Thread Details. For this project, you have to build an application that creates multiple threads. These threads request for multiple resources periodically and then release the resources. The objective of the scheduler (which is distributed) is to try to satisfy these requests. The scheduler can either satisfy a request immediately or delay the request and then satisfy it eventually.

Similarly, the threads wishes to release the resources that it has acquired. A thread requests the scheduler to release the resources which are then released.

To achieve this, some of the data-structures are maintained by each thread is described here:

- **myMax**: A vector of size m defining the maximum demand of each process P_i. If $myMax[j]$ equals k, then process P_i may request at most k instances of resource type R_j.
- **myAlloc**: A vector of size m defining the number of resources of each type currently allocated to process P_i. If $myAlloc[j]$ equals k, then process P_i is currently allocated k instances of resource type R_j.
- **canClaim**. A vector of size m defining the remaining resource need of each process. If $canClaim[j]$ equals k, then process P_i may need k more instances of resource type R_j to complete its task. Note that $canClaim[j]$ equals $myMax[j] - myAlloc[j]$.
- **newReq**. A vector of size m defining the new request made by a process. If $newReq[j]$ equals k, then process P_i is requesting k more instances of resource type R_j to complete its task. Note that $newReq \leq canClaim$.
- **myRelease**. A vector of size m defining the number of resources to be released by a process. If $newReq[j]$ equals k, then process P_i is releasing k instances of resource type R_j. Note that $myRelease \leq myAlloc$.

In addition to these data-structures, each thread maintains a few more variables. The details of these are given in the pseudocode below:

Listing 1: main thread

```
void main()
{
    ...
    ...
    Initialize Max and Available arrays;
    create n worker threads;
    ...
    ...
}
```

Listing 2: Worker thread

```
void workerThread()
{
    id = thread.getID();

    // resIters is the number of iteration this thread has to execute
    for (i=0; i < resIters; i++)
    {
        t1 = exponentially averaged time with mean μ;
        sleep(t1);    // simulates performing some other tasks

        randVal = an uniformly random value between 0 and 1;

        // acqRelRatio stands for ratio of acquiring & releasing resources
        if (randVal =< acqRelRatio) {
            // compute the resources to be acquired
            for (i =0; i<m; i++) {
                // compute the max value that can be claimed
                canClaim[m] = myMax[m] − myAlloc[m];

                // compute the actual value of resource to claim
                newReq[m] = random integer value between 0 − canClaim[m];

                // Allocate these resources to the process preemptively
                myAlloc[m] += newReq[m];
            }

            reqTime = getSysTime();
            cout << i << "th resource request by thread " << id <<
            " made at time " << reqTime << " consisting
             of the following items " << newReq;

            // Request the resources to the distributed scheduler which performs safety check
            request(newReq);

            grantTime = getSysTime();
            cout << i << "th resource request by thread " << id <<
            " granted at time " << grantTime;
        }

        else {    // Resources to be released
```

```
41              // compute the resources to be released
42              for (i =0; i<m; i++) {
43
44                  // compute the resources to release
45                  myRelease[m] = random integer value between 0 − myAlloc[m];
46
47                  // Release the resources from the process preemptively
48                  myAlloc[m] −= myRelease[m];
49              }
50
51              reqTime = getSysTime();
52              cout << i << "th release request by thread " << id <<
53              " made at time: " << reqTime << " consisting
54               of the following items " << myRelease;
55
56              // Request the distributed scheduler to release the resources
57              release(myRelease);
58
59              grantTime = getSysTime();
60              cout << i << "th release request by thread " << id <<
61              " granted at time " << grantTime;
62          }
63      }
64
65      // Finally release any other allocated resources
66      t1 = exponentially averaged time with mean μ;
67      sleep(t1);   // simulates performing some other tasks
68
69      reqTime = getSysTime();
70      cout << "Final release request by thread " << id <<
71      " made at time " << reqTime << " consisting
72       of the following items " << myRelease;
73
74      release(myAlloc);
75
76      grantTime = getSysTime();
77      cout << i << "Final release request by thread " << id <<
78      " granted at time " << grantTime;
79  }
```

In this application, the methods *request* and *release* have to be implemented by the scheduler. The details of the scheduler are described below.

Deadlock Detection Scheduler Details. You have to implement the distributed scheduler given in the book. The following data-structures as described in the textbook are used:

- **Available**: A vector of length m which indicates the number of available resources of each type. If $Available[j]$ equals k, then k instances of resource type R_j are available.
- **Max**: An $n \times m$ matrix defining the maximum demand of each process. If $Max[i][j]$ equals k, then process P_i may request at most k instances of resource type R_j.
- **Allocation**: An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If $Allocation[i][j]$ equals k, then process P_i is currently allocated k instances of resource type R_j.

- **Need**. An $n \times m$ matrix indicates the remaining resource need of each process. If $Need[i][j]$ equals k, then process P_i may need k more instances of resource type R_j to complete its task. Note that $Need[i][j]$ equals $Max[i][j] - Allocation[i][j]$.

It can be seen that the matrices $Allocation$ and $Need$ keep changing over time. Thus to protect these matrices from synchronization issues, you can assume that there is a single lock for these structures.

Input: The input to the program will be a file, named inp-params.txt, consisting of all the parameters described above:$m, Available_j$ (the initial instance of resources available for each resource type j), $n, Max[n, m]$(the maximum need of each process P_i for each resource), $resIters$ (the number of iterations of a process P_i), $acqRelRatio$ (ratio of acquire to release for process P_i), μ.
A sample input file is:
$m = 10$
$Available_1 = 10, Available_2 = 7,$ ..... , $Available_{10} = 11$
$n = 5,$
$Max[1, 1] = 6, Max[1, 2] = 3,$ ..... , $Max[1, 10] = 10$
$Max[2, 1] = 4, Max[1, 2] = 6,$ ..... , $Max[1, 10] = 5$
.....
$Max[5, 1] = 8, Max[5, 2] = 5,$ ..... , $Max[5, 10] = 3$

$resIters = 10$
$\mu = 5$
$acqRelRatio = 2$

Output: Your program should output to a file in the format given in the pseudocode for each algorithm. The output of all the threads must be combined. A sample output is as follows:
1st resource request made by thread1 at time 10:00 consisting of the following items 4 5 6
1st resource request made by thread3 at time 10:02 consisting of the following items 8 4 7
2nd resource request made by thread1 at time 10:03 consisting of the following items 2 1 1
2nd resource release made by thread3 at time 10:05 consisting of the following items 1 2 3
.
.
.

Report: You have to submit a report for this assignment. This report should contain a comparison of the performance the banker's algorithm with various parameters. You must execute these threads multiple times to compare the performances and display the result in form of a graph (result graph and not a graph consisting of nodes and vertices).

In all these experiments, please have the same values for all the parameters mentioned above. The y-axis of the graph consist of two curves:

- one is the average time taken by the scheduler to process a request method made by a thread
- the other being the average time taken by the scheduler to process a release method made by a thread

The x-axis of the graph will consist of ratio of $acqRelRatio$. It will specifically consist of the following 11 values: 10, 8, ... 2, 1, 0.8, 0.6, ... 0.2, 0.1.

Finally, you must also give an analysis of the results while explaining any anomalies observed (like in the previous assignments).

Deliverables: You have to submit the following:

- The source file containing the actual program to execute named as bankers-<rollno>.c
- A readme.txt that explains how to execute the program
- The report as explained above

Zip all the three files and name it as Assn10-<rollno>.zip. Then upload it on the google classroom page of this course. Submit it by **22nd November 2015, 9:00 pm**.