Concurrency Control in Transactional Systems: Spring 2025
Programming Assignment 1: Implementing BOCC and FOCC algorithms
Submission Date: Schedule Given Below

Goal: The goal of this assignment is to implement BOCC and FOCC algorithms studied in the class. Implement both these optimistic algorithms in C++.

Details. As shown in the book, you have to implement both the optimistic concurrency algorithms **in C++**: BOCC & FOCC. Since, you are using optimistic concurrency control approach, all writes become visible only after commit. Thus on abort of a transaction, no rollback is necessary as none of the writes of the transactions will ever be visible.

You have to implement the following methods for both the algorithms:

- $begin\_trans()$: It begins a transactions and returns a unique transaction id, say i
- $read(i, x, l)$: Transaction t_i reads data-item x into the local value l.
- $write(i, x, l)$: Transaction t_i writes to data-item x with local value of l.
- $tryC(i)$: Transaction t_i wants to commit. The return of this function is either a for abort or c for commit.

For FOCC algorithm, please implement the two variants: (1) the validating transaction gets aborted on detection of a conflict. Let us call this as *current-transaction-abort* or *CTA*. (2) the transactions conflicting with validating transaction gets aborted. We call this *other-transaction-abort* or *OTA*. We call these variants as: (1) FOCC-CTA (2) FOCC-OTA. As discussed in the class, FOCC-OTA is extra-credit.

There are no such variants for BOCC. Hence, there is only variant for BOCC. Thus there are two or three variants to implement (depending on the extra credit) for both the algorithms.

To test the performance of both the algorithms, develop an application, opt-test is as follows. Once, the program starts, it creates n threads and an array of m shared variable. Each of these threads, will update the shared array randomly. Since the threads could simultaneously update the shared variables of the array, the access to shared variables have to be synchronized. This synchornization is performed using the above mentioned methods of BOCC, FOCC and the variants.

To pseudocode opt-test given is as follows explains the idea better:

Listing 1: main thread

```
1 void main()
2 {
```

```
3       ...
4       ...
5       // create a shared array of size m
6       shared[] = new SharedArray[m];
7       ...
8       ...
9       create n updtMem threads;
10  }
```

Listing 2: updtMem thread

```
1
2   void updtMem()
3   {
4       int status = abort;        // declare status variable
5       int abortCnt = 0;          // keeps track of abort count
6
7       long critStartTime, critEndTime;
8
9       // Each thread invokes numTrans transactions. numTrans is computed from totTrans.
10      for (int curTrans=0; curTrans<numTrans; curTrans++)
11      {
12          abortCnt = 0;                       // Reset the abort count
13          critStartTime = getSysTime();       // keep track of critical section start time
14
15          // getRand(k) function used in this loop generates a random number in the range 0 .. k
16          do
17          {
18              id = begin_trans(); // begins a new transaction id
19
20              int locVal;
21              for (int i=0; i<numIters; i++)
22              {
23                  // gets the next random index to be updated
24                  randInd = getRand(m);
25
26                  // gets a random value using the constant constVal
27                  randVal = getRand(constVal);
28
29                  // reads the shared value at index randInd into locVal
30                  read(id, shared[randInd], locVal);
31
32                  logFile << "Thread id " << pthread_self() << "Transaction " << id <<
33                  " reads from" << randInd << " a value " << locVal << " at time " <<
34                  getSysTime;
35
36                  // update the value
37                  locVal += randVal;
38
39                  // request to write back to the shared memory
40                  write(id, shared[randInd], locVal);
41
42                  logFile << "Thread id " << pthread_self() << "Transaction " << id <<
43                  " writes to " << randInd << " a value " << locVal << " at time " <<
44                  getSysTime;
45
```

```
46                  // sleep for a random amount of time which simulates some complex computation
47                  randTime = getExpRand(λ);
48                  sleep(randTime);
49              }
50
51              status = tryCommit(id);       // try to commit the transaction
52              logFile ≪ "Transaction " ≪ id ≪ " tryCommits with result "
53              ≪ status ≪ " at time " << getSysTime;
54              abortCnt++;                   // Increment the abort count
55          }
56          while (status! = commit);
57
58          critEndTime = getSysTime(); // keep track of critical section end time
59          record commitDelay & abortCnt;        // Record these values for collecting statistics
60      }   // End numTrans for
61  }
```

Here $numIters$ is obtained as an input parameter from the user (as discussed in the class). $numTrans$, the number of transactions executed by a thread is computed from parameter, $totTrans$ which is the total number of transactions executed by all the threads. $totTrans$ is obtained as a user input and $numTrans$ is obtained by dividing it equally among all the threads.

$randTime$ is an exponentially distributed with an average of λ milli-seconds. The objective of having this time delay is to simulate that these threads are performing some complicated time consuming tasks. It can be seen that the time taken by a transaction to commit, *commit-Delay* is defined as $critEndTime - critStartTime$.

Input: The input to the program will be a file, named inp-params.txt, consisting of all the parameters described above:$n, m, totTrans, constVal, \lambda, numIters$. A sample input file is: 10 10 5000 100 20.

As mentioned above, you will have to compute $numTrans$ from the parameter $totTrans$ by dividing it by n, the number of threads.

Output: Your program should output two files in the format given in the pseudocode for each algorithm: (1) BOCC-log.txt (2) FOCC-CTA-log.txt (3) FOCC-OTA-log.txt. A sample output is as follows:

The Output of your program (for any of the variants mentioned):
Thread 1 Transaction 1 reads 5 a value 0 at time 10:00
Thread 2 Transaction 2 reads 7 a value 0 at time 10:02
Thread 1 Transaction 1 writes 5 a value 15 at time 10:05
Thread 2 Transaction 2 tryCommits with result abort at time 10:10

.
.
.

The output is essentially a history. By inspecting the output one should be able to verify the serializability of your implementations.

Report: You have to submit a report for this assignment. This report should contain a comparison of the performance of the various variants of BOCC & FOCC. The comparison must consist of the following graphs. In each of the following cases, the x-axis varies while the y-axis measure two metrics: (a) Average time taken for a transaction to commit (b) Average number of aborts of a transaction before it can commit. Thus, you will have to plot two graphs for each metric of y-axis.

1. Number of Transactions: The x-axis should be the number of transactions varying from 1000 to 5000 in the increments of 1000; the y-axis is the average commitDelay and the number of aborts (as described above). The other parameters are as follows:

   - the number of threads in the system the same as number of cores of your laptops such as 16.
   - the total number of variables in the database as 1000.

2. Number of variables in the database: The x-axis should be the number of variables of the database varying from 1000 to 5000 in the increments of 1000. The other parameters are as follows:

   - the number of threads in the system the same as number of cores of your laptops such as 16.
   - the total number transactions as 1000.

3. Number of threads in the system: The x-axis should be the number of threads varying from 2 to 32 in the powers of 2. All these threads execute until the total number of transactions executed are 1000. Once all the threads execute 1000 transactions, the system terminates. The other parameters are as follows:

   - the total number of variables in the database as 1000.
   - the total number transactions as 1000.

Please clearly mention in your report, the number of threads that you considered for the experiments 1, 2 mentioned above. And, for all the above experiments, please consider the following parameters:

- $numIters$: You can choose to be 20.
- $constVal$: You can have this as 1000.
- λ: Let this be 20 ms.

All the graphs will have two curves, one for each of the algorithms: (1) BOCC (2) FOCC-CTA. And in case you implement extra credit, you will have another curve - (3) FOCC-OTA.

You must run these algorithms multiple times to obtain performances values. Finally in your report, you must also give an analysis of the results while explaining any anomalies observed.

Deliverables: You have to submit two kinds of deliverables:

Pseudocode: First, you will have to submit the Pseudocode. The pseudocode of BOCC & FOCC. Name them as: BOCC-<rollno>.pcode, FOCC-CTA-<rollno>.pcode, FOCC-OTA-<rollno>.pcode. The objective of this exercise is to ensure that you have spent sometime in understanding the difficulty of the problem and developed the pseudocode accordingly. Some of the issues of the pseudocode were discussed in the class.
Zip all the three files and name it as ProgAssn1_PCode-<rollno>.zip. Then upload it on the google classroom page of this course. Submit by the deadline shown below.

Actual Code: Next, you will have to submit the actual code. The details are as follows:

- The source files of BOCC & FOCC coded in C++. Name them as: BOCC-<rollno>.cpp, FOCC-CTA-<rollno>.cpp, FOCC-OTA-<rollno>.cpp.

- A readme.txt that explains how to execute the program.
- The report as explained above

Zip all the three files and name it as ProgAssn1_Code-<rollno>.zip. In summary, the submission schedule is as follows:

1. **Pseudocode**: 20th March 2025, 9:00 pm
2. **Actual running code in C++**: 27th March 2025, 9:00 pm