Concurrency Control in Transactional Systems: Spring 2025
Programming Assignment 2: Implementing BTO and MVTO algorithms
Submission Date: As shown below.

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

Details. As shown in the book, you have to implement BTO and MVTO algorithms studied in the class **in C++**. But unlike the version that we studied in the class, you have to implement these algorithms you are using optimistic concurrency control approach, i.e. all writes become visible only after commit. It can be seen that the advantage of this approach: upon an abort of a transaction, no rollback is necessary as none of the writes of the transactions will ever be visible.

MVTO Variants: An important requirement with MVTO algorithm is to delete the unwanted (garbage) versions. There are two ways to do it. Either through garbage collection. Or have a fixed number of of versions, say like 5. Once all the 5 versions have been written onto, the next transaction write overwrites the first version. This way at any time only a fixed number of versions (5 in this case) will be in the system for any data-item.

So, you have to implement the garbage collection procedure as well that we studied in the class. So for MVTO, you have to implement three variants:

1. MVTO: Normal MVTO that we studied in the class.
2. MVTO-gc: The variant of MVTO with garbage collection. Here the garbage collection routine is invoked whenever a transaction terminates.
3. K-MVTO: The variants with k versions of a data-item maintained any time.

Methods to Implement: Just like in previous assignment, 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 l.
- $tryC(i)$: Transaction t_i wants to commit. The return of this function is either a for abort or c for commit.

To test the performance of both the algorithms, develop an application, opt-test which you developed in the previous assignment. It is as follows: Once, the program starts, it creates n threads and an array of m shared variables. 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. The synchronization is performed using the above mentioned methods of BTO & MVTO.

Test Program: To test your libraries, you have to develop a test program, say opt-test like in the previous assignment. The pseudocode opt-test given is as follows:

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 = -1;         // keeps track of abort count
6
7      long critStartTime, critEndTime;
8
9      critStartTime = getSysTime();      // keep track of critical section start time
10
11     // getRand(k) function used in this loop generates a random number in the range 0 .. k
12     do
13     {
14         id = begin_trans(); // begins a new transaction id
15         randIters = getRand(m); // gets the number of iterations to be updated
16
17         int locVal;
18         for (int i=0; i<randIters; i++)
19         {
20             // gets the next random index to be updated
21             randInd = getRand(m);
22
23             // gets a random value using the constant constVal
24             randVal = getRand(constVal);
25
26             // reads the shared value at index randInd into locVal
27             read(id, shared[randInd], locVal);
28
29             logFile << "Thread id " << pthread_self() << "Transaction " << id <<
30             " reads from" << randInd << " a value " << locVal << " at time " <<
31             getSysTime;
32
```

```
33                  // update the value
34                  locVal += randVal;
35
36                  // request to write back to the shared memory
37                  write(id, shared[randInd], locVal);
38
39                  logFile ≪ "Thread id " ≪ pthread_self() ≪ "Transaction " ≪ id ≪
40                  " writes to " ≪ randInd ≪ " a value " ≪ locVal ≪ " at time " ≪
41                  getSysTime;
42
43                  // sleep for a random amount of time which simulates some complex computation
44                  randTime = getExpRand(λ);
45                  sleep(randTime);
46              }
47
48              status = tryCommit(id);        // try to commit the transaction
49              logFile ≪ "Transaction " ≪ id ≪ " tryCommits with result "
50              ≪ status ≪ " at time " << getSysTime;
51              abortCnt++;                    // Increment the abort count
52          }
53          while (status! = commit);
54
55          critEndTime = getSysTime(); // keep track of critical section end time
56      }
```

Here $randTime$ is an exponentially distributed with an average of λ mill-seconds. The objective of having this time delays 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, *commitDelay* is defined as $criteEndTime - critStartTime$.

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

Output: Your program should output to a file in the format given in the pseudocode for each algorithm. A sample output is as follows:

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 BTO & MVTO's variants similar to FOCC and BOCC comparison. 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.
    - for K-MVTO algorithm, have K as 5

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.
    - for K-MVTO algorithm, have K as 5

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.
    - for K-MVTO algorithm, have K as 5

4. Varying K in for K-MVTO algorithm: This experiment is specific only to K-MVTO. The x-axis is the value of K varying from 5 to 25 in the increments of 5. 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.
    - the total number transactions as 1000.

Please clearly mention in your report, the number of threads that you considered for the experiments 1, 2 and 4 mentioned above. As mentioned above, Experiment 4 is specific only to K-MVTO.

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 four curves, one for each of the algorithms: (1) BTO (2) MVTO (3) MVTO-gc (4) K-MVTO.

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 two sets of deliverables with different deadlines as follows.

Deliverables1: In the first set of deliverables you will have to submit the detailed pseudocode for the above mentioned MVTO variants: (1) MVTO (2) MVTO-gc (3) K-MVTO.
Zip all the three files and name it as Pseudocode_MVTO-<rollno>.zip. Please submit it on google classroom by **6th April 2025, 9:00 pm**.

Deliverables2: In the second set of deliverables you will have to submit the following:

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

Zip all the three files and name it as ProgAssn2-BTO_MVTO-<rollno>.zip. Then upload it on the google classroom page of this course by **9th April 2025, 9:00 pm**.