Programming Assignment 3
Wait-Free Atomic Snapshot Implementations Comparing the Solutions of MRSW and MRMW
Submission Date: 8th November 2020, 9:00 pm

Goal: The goal of this assignment is to implement the two Atomic Wait-Free Snapshots algorithms: the solutions for taking atomic snapshots of MRSW and MRMW registers that we discussed in the class. You have to compare the averages time taken to complete the snapshot by both the algorithms. Implement both these algorithms **in C++**.

Details. As explained above, you have to implement the two atomic wait-free snapshot algorithms: taking atomic snapshots of MRSW and MRMW registers. You have to implement them **in C++**. As explained in the class, with MRMW registers multiple threads can write to the same location while with MRSW registers each thread write to only one location.

The interface of the snapshot object for the MRMW, denoted as MRMW-Snap is:
$update(l, v), snapshot()$

Here the parameter l in update function is the location onto which the thread wants to perform a write. While the snapshot method returns the snapshot of all the registers. The interface for the MRSW-Snap is the same as above with one change. The update method does not take l since the location of each thread is fixed.

Intuitively, it seems that MRSW snapshot algorithm will be faster faster. Please verify this using your experiments. To test the performance of locking algorithms, develop an application, snap-test is as follows. Once, the program starts, it creates n threads - n writer threads and $n + 1^{st}$ snapshot collecting thread. Each writer thread regularly updates its entry in the shared array with a delay that is exponentially distributed with an average of μ_w microseconds. The snapshot thread collects snapshots with a time delay that is exponentially distributed with an average of μ_s microseconds. The program terminates after the snapshot thread has collected k snapshots. The pseudocode for the snap-test function is as follows:

Listing 1: main thread

```
1
2  // Create a snapshot object of size M
3  MRMW-Snap  MRMW-SnapObj  = new  MRMW-Snap (M) ;        // or MRSW-Snap object
4
5  // A variable to inform the writer threads they have to terminate
6  atomic⟨bool⟩  term ;
7
8  void main ( )
9  {
10      // Initialization of shared variables
11      initialize  MRMW-SnapObj ;
12      term = false ;
```

```
13        ...
14        ...
15     create n writer threads;
16     create a snapshot collector thread;
17        ...
18        ...
19
20     wait until the snapshot thread terminates;
21
22     term = true;        // Inform all the writer threads that they have to terminate
23     wait until all the writer threads terminate;
24
25     write all the log entries onto the file;
26  }
```

Listing 2: writer thread

```
1
2  void writer()
3  {
4      int v, pid;
5      int t1;
6
7      pid = get_threadid();
8      while (!term)            // Execute until term flag is set to true
9      {
10          v = getRand();             // Get a random integer value
11          l = getRand(M);            // Get a random location in the range 1..M
12
13          // For MRSW-Snap object, the location will not be provided
14          MRMW-SnapObj.update(l, v);
15
16          record system time and the value v in a local log;
17
18          t1 = random value of time exponentially distributed with an avg of $\mu_w$;
19          sleep(t1);
20      }
21  }
```

Listing 3: snapshot thread

```
1
2  void snapshot()
3  {
4      int i=0;
5      int t2, beginCollect, endCollect;
6
7      while (i < k)
8      {
9          beginCollect = system time before the ith snapshot collection;
10          MRMW-Snap.snapshot(); // collect the snapshot
11          endCollect = system time after the ith snapshot collection;
12          timeCollect = endCollect − beginCollect;
13
14          store ith snapshot and timeCollect;
15
16          t2 = random value of time exponentially distributed with an avg of $\mu_r$;
```

```
17            sleep(t2);
18
19            i++;
20        }
21    }
```

To verify the correctness of the snapshot algorithm, each writer thread logs the value it wrote its register entry followed by the timestamp. Similarly, the snapshot thread logs the values of each snapshot that it collects.

The program consists of a shared array consisting of atomic registers. A C++ library that implements atomic registers can be found here: http://www.cplusplus.com/reference/atomic/atomic/. You can use other atomic register implementations as well.

Input: The input to the program will be a file, named inp-params.txt, consisting of all the parameters described above: $n, M, \mu_w, \mu_r, k,$. A sample input file is: 10 20 100 100 5.

Output: Your program should first output the sequence of values written to the shared array and the snapshot collected. It should also output the times when these events occurred. A sample output format given as follows:

MRSW snapshot output:
Thr1's write of 10 at 10:00
Thr3's write of 24 at 10:03
Thr2's write of 17 at 10:05
Thr8's write of 32 at 10:06

.
.
.

Snapshot Thr's snapshot: l1-5 l2-17 l3-24 ...... ln-43 which finished at 11:30 (here l1 means location1).
Similarly, the output from MRMW snapshot algorithm:
Thr1's write of 10 on location 2 at 10:00
Thr3's write of 24 on location 5 at 10:02
Thr2's write of 17 on location 2 at 10:05
Thr8's write of 31 on location 10 at 10:06

.
.
.

Snapshot Thr's snapshot: l1-5 l2-17 l3-24 ...... ln-43 which finished at 11:33.
The snapshot output must demonstrate that it is consistent, i.e. linearizable.

Report: You have to submit a report for this assignment. This report should contain a comparison of the performance of average time and worst-case taken for the MRSW and MRMW snapshot implementations. You must run both these algorithms multiple times to compare the performances and display the result in form of a graph.

In all these experiments, please have the same values for n, M and k. For instance have the following: n as 10, M as 20 and k as 5. The y-axis of the graph is the average time taken for the snapshot algorithm to complete. It can be seen from the description above that in each execution, the time taken for the snapshot should be averaged over k.

The x-axis of the graph will consist of ratio of μ_s/μ_w, i.e. the ratio of the average delays of the snapshot thread to the writer thread. It will specifically consist of the following 11 values: 10, 8, ... 2, 1, 0.8, 0.6, ... 0.2, 0.1.

Thus, your graph will have have **fours** curves: (1) two for MRSW snapshot - average and worst-case time for MRSW snapshot collection. (2) two for MRMW snapshot - average and worst-case time for MRMW snapshot collection. Finally, you must also give an analysis of the results while explaining any anomalies observed (like in the previous assignment).

Deliverables: You have to submit the following:

- The source files containing the actual program to execute. Please name them as: mrsw-<rollno>.cpp, mrmw-<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 ProgAssn3-<rollno>.zip. Then upload it on the google classroom page of this course. Submit it by the above mentioned deadline.