Programming Assignment 4
Comparison of Wait-Free and Obstruction-Free Atomic Snapshot Implementations of MRMW Registers

Submission Date: 2nd April 2022, 9:00 pm

Goal: The goal of this assignment is to compare the Wait-Free and Obstruction-Free atomic snapshot implementation on 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 atomic Wait-Free and Obstruction-Free snapshot algorithms on 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.

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.

Intuitively, it seems that Wait-Free snapshot algorithm will be faster faster. Please verify this using your experiments. To test the performance of locking algorithms, develop an application, snap-test as follows. Once, the program starts, it creates $n_w + n_s$ threads - n_w writer threads and n_s snapshot threads. 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 threads collect snapshots with a time delay that is exponentially distributed with an average of μ_s microseconds. The program terminates after each 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);
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_w writer threads;
16     create n_s snapshot collector threads;
17     ...
```

```
18      ...
19
20      wait until all the snapshot threads 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          MRMW-SnapObj.update(l, v);
14
15          record system time and the value v in a local log;
16
17          t1 = random value of time exponentially distributed with an avg of μ_w;
18          sleep(t1);
19      }
20  }
```

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 μ_s;
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: $n_w, n_s, M, \mu_w, \mu_s, k$. A sample input file is: 10 5 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:
The output from MRMW snapshot algorithm with the writers as:
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 Thr1's snapshot: l1-5 l2-17 l3-24 ..... ln-43 which finished at 11:30 (here l1 means location1).
Snapshot Thr2's snapshot: l1-5 l2-12 l3-34 ..... ln-55 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 Wait-Free and Obstruction-Free 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 M, μ_w, μ_s and k while varying the values of n_w, n_s (explained below). For instance consider the following: M as 20, μ_w as 0.5 , μ_s as 0.5, 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 n_w/n_s, i.e. the ratio of the number of writer threads to the number of snapshot threads. It will specifically consist of the following 11 values: 10, 8, ... 2, 1, 0.8, 0.6, ... 0.2, 0.1.

Each graph developed by you will have have **fours** curves: (1) two for Obstruction-Free snapshot algorithms - average and worst-case time for MRMW snapshot collection. (2) two Wait-Free snapshot algorithms - 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: 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 ProgAssn4-<rollno>.zip. Then upload it on the google classroom page of this course. Submit it by the above mentioned deadline.