CS5300 - Parallel & Concurrent Programming
Fall 2023
Comparing Obstruction Free and Wait-Free Snapshot Algorithms
Submission Date: 25th October 2023 9:00 pm

Objective: 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. Similar to the earlier assignment, you can implement both these algorithms in one of the languages **C++/Rust/Go**.

Details. In the class we discussed two methods for concurrent Snapshot:

- Obstruction Free Snapshot Algorithm: The Obstruction Free Snapshot algorithm is described in figure 4.17 on page 94 of the textbook (page 111 of the pdf). Note this algorithm has been described for MRSW case. However, this algorithm can be easily modified for MRMW case.
- Wait-Free Snapshot Algorithm: The Wait-Free Snapshot algorithm for MRMW case was discussed in the class. The corresponding material was uploaded on the classroom page.

The interface of the snapshot object for with MRMW registers, 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. 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 shown below.

You have to implement both the algorithms Wait-Free Snapshot and Obstruction-Free Snapshot in one of the languages **C++/Rust/Go**.. You can respectively name them as ⟨OFS⟩.extn and ⟨WFS⟩.extn.

Experimental Setup: To test the Snapshot, please develop a program which is outlined below.

Listing 1: main thread

```

    // Create a snapshot object of size M
    MRMW-Snap MRMW-SnapObj = new MRMW-Snap(M);

    // A variable to inform the writer threads they have to terminate
    atomic<bool> term;

```

```
void main()
{
    // Initialization of shared variables
    initialize MRMW-SnapObj ;
    term = false;
    ...
    ...
    create $n_w$ writer threads;
    create $n_s$ snapshot collector threads;
    ...
    ...

    wait until all the snapshot threads terminates;

    term = true;        // Inform all the writer threads that they have to terminate
    wait until all the writer threads terminate;

    write all the log entries onto the file;
}
```

Listing 2: writer thread

```

void writer()
{
    int v, pid;
    int t1;

    pid = get_threadid();
    while(!term)            // Execute until term flag is set to true
    {
        v = getRand();              // Get a random integer value
        l = getRand(M);             // Get a random location in the range 1..M

        MRMW-SnapObj .update(l, v);

        record system time and the value v in a local log;

        t1 = random value of time exponentially distributed
        with an avg of $\mu_w$;
        sleep(t1);
    }
}
```

Listing 3: snapshot thread

```

void snapshot()
{
    int i=0;
    int t2, beginCollect, endCollect;

    while (i < k)
    {
        beginCollect = system time before the $i$th snapshot collection;
        MRMW-SnapObj .snapshot(); // collect the snapshot
        endCollect = system time after the $i$th snapshot collection;
```

```
            timeCollect = endCollect − beginCollect;

            store $i$th snapshot **and** timeCollect;

            t2 = random value of time exponentially distributed
            with an avg of $\mu_s$;
            sleep(t2);

            i++;
        }
    }
```

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.

Here the function $computeStats()$ as the name suggests computes the various statistics on the times taken. It should compute the average time taken for: (a) all the update operations (b) all the scan operations (c) all the operations including both the update and scan.

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: 20 5 40 0.5 0.5 5. Consider the average sleeping time to be in milli-seconds.

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. The report should first explain the design of your program while explaining any complications that arose in the course of programming.

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 atleast 5 times and then take its average to compare the performances and display the result in form of graphs for the following experiments.

1. **Average-case Scalability:** In this graph, the Y-axis will represent two values, the average time taken by Obstruction-Free and Wait-Free snapshot algorithms in each of the following cases: (1) update threads, (2) scan threads, (3) all threads (update + scan). The X-axis will depict the number

of threads. The number of threads will vary from 4 to 32 in powers of 2. To maintain consistency in the experiment, all other parameters will remain constant as follows: $n_w/n_s = 4, M = 40, \mu_w = 0.5, \mu_s = 0.5, k = 5$.

2. **Worst-case Scalability:** In this graph, the Y-axis will represent two values, the worst-case time taken by Obstruction-Free and Wait-Free snapshot algorithms in each of the following cases: (1) update threads, (2) scan threads, (3) all threads (update + scan). The X-axis will depict the number of threads. The number of threads will vary from 4 to 32 in powers of 2. To maintain consistency in the experiment, all other parameters will remain constant as follows: $n_w/n_s = 4, M = 40, \mu_w = 0.5, \mu_s = 0.5, k = 5$.

3. **Impact of update operation on Scan in the average-case:** In this graph, the Y-axis will represent two values, the average-case time taken by Obstruction-Free and Wait-Free snapshot algorithms in each of the following cases: (1) update threads, (2) scan threads, (3) all threads (update + scan). The X-axis will depict the 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 6 values: 10, 8, 6, 4, 2, 1. Have the remaining parameters as: $n_s = 4, M = 20, \mu_w = 0.5, \mu_s = 0.5, k = 5$. Given the value of n_s as 4, based on the ratio of n_w/n_s, the value of n_w can be determined.

4. **Impact of update operation on Scan in the worst-case:** In this graph, the Y-axis will represent two values, the worst-case time taken by Obstruction-Free and Wait-Free snapshot algorithms in each of the following cases: (1) update threads, (2) scan threads, (3) all threads (update + scan). The X-axis will depict the 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 6 values: 10, 8, 6, 4, 2, 1. Have the remaining parameters as: $n_s = 4, M = 20, \mu_w = 0.5, \mu_s = 0.5, k = 5$. Given the value of n_s as 4, based on the ratio of n_w/n_s, the value of n_w can be determined.

Each plot developed by you will have have **six** curves: two for each of (1) update threads, (2) scan threads, (3) all threads (update + scan) for Obstruction-Free and Wait-Free algorithms in each case. 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 file containing the actual program to execute. Please name it as Src-⟨rollno⟩.cpp. Please follow this convention. Otherwise, your program will not be evaluated. If you are submitting more than one file name it as wfs-⟨rollno⟩.extn and obs-⟨rollno⟩.extn.
- 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 above mentioned deadline. Please follow the naming convention. Otherwise, your assignment will not be evaluated by the TAs.

Evaluation: The break-up of evaluation of your program is as follows:

1. Program design as explained in the report - 30%
2. The Graphs obtained and the corresponding analysis shown in the report - 30%
3. Program execution - 30%
4. Code documentation and indentation - 10%.

Please make sure that you your report is well written since it accounts for 60% of the marks.

Late Submission and Plagiarism Check: All assignments for this course has the late submission policy of a penalty of 10% each day after the deadline for 6 days Submission after 6 days will not be considered.

Kindly remember that all submissions are subjected to plagiarism checks.