CS131 HW3 - Report

Abstract

Using Synchronized keywords in Java can prevent thread interference and memory consistency errors in the shared server. However, it is not the best way, it is safe but slow. This report is to measure the performance and reliability for different thread safety approaches through using a sequential-consistency-violating testing program, The tests will be conducted on two servers, along with various values for the size of the state array, the number of threads. Eventually, the result would be gathered, compared and analysed.

1. Implementations Analysis

Four different synchronization methods is measured:

1.1 Synchronized State

An implementation uses the Synchronized class. With this method, only one thread would be executed, and other threads that invoke synchronized methods for the same object would be blocked until the first thread finished executing. This state blocks the entire swap method, which means it is data-race free (DRF). It has a very strong reliability but it also has slow performance.

1.2 Unsynchronized State

This implementation is similar with the Synchronized State but it does not have synchronized keywords. Through this implementation, multiple threads can execute at the same time. It will not block other threads. It is not DRF since the data race would happen in this case.

1.3 Null State

An implementation of State that does nothing. Swapping has no effect. This is used for timing the scaffolding of the simulation.

1.4 AcmeSafe State

This class aims to keep a good reliability but also have fast performance.

It uses java.util.concurrent.atomic.AtomicLongArray class, which is a long array in which elements may be updated atomically. This implementation also doesn't use synchronized keywords. So it would not block other threads while a thread is executing. In this state, `getAndDecrement` and `getAndIncrement` are atomic operations consisting of Read and write operations. It is not DRF since data race may happen when the threads are interleaved. And the classes in java.util.concurrent.atomic cannot resolve racing conditions. Some data race errors could result when there are not enough constraints to be sure that a given Read has only one possible Write that it could read from.

2. Performance and Reliability Results

The measurement is conducted in Java version 13.0.2. The test is operated on two servers with different types of CPUs: The first is lnxsrv06, its CPU model is Intel(R) Xeon(R) CPU E5620 with clock rate 2.40GHz. It has 16 processors and with 4 cores on each.; The second is lnxsrv10, its CPU model is Intel(R) Xeon(R) Silver 4116 CPU, clock rate is 2.10GH, which has four processors and each has 4 cores

The below are the tables of the average swap real time for each state in two different servers. The row heading is thread numbers, the column heading is the size of the state array. The total number of swap transitions for the measurement is 100 million.

2.1 Lnxsrv 06 Server:

2.1.1 SynchronizedState:

Thread Numbers\ Size of state array	State Array of 5 Entries	State Array of 100 Entries	State Array of 500 Entries
1 thread	22.4399	26.6184	23.3147
8 threads	2125.18	1785.36	2093.53
16 threads	3562.28	3475.34	3684.98
40 threads	8226.02	9198.80	9569.58

Avg Swap Real Time (ns)

2.1.2 UnsynchronizedState:

Thread Numbers\ Size of state array	State Array of 5 Entries	State Array of 100 Entries	State Array of 500 Entries
1 thread	16.1902	22.4741	17.3461
8 threads	253.800	377.050	543.112
16 threads	415.263	528.768	853.451
40 threads	1152.36	1352.92	2358.18

Avg Swap Real Time (ns)

2.1.3 NullState

Thread Numbers\ Size of state array	State Array of 5 Entries	State Array of 100 Entries	State Array of 500 Entries
1 thread	22.0912	28.6049	13.6613
8 threads	22.0015	21.2536	21.2026
16 threads	39.5595	41.6873	40.2558
40 threads	247.091	130.203	127.033

Avg Swap Real Time (ns)

2.1.4 AcmeSafeState

Thread Numbers\ Size of state array	State Array of 5 Entries	State Array of 100 Entries	State Array of 500 Entries
1 thread	22.8493	24.2219	23.0546
8 threads	535.419	495.617	321.771
16 threads	1142.86	818.758	569.666
40 threads	2843.67	1551.10	1526.85

Avg Swap Real Time (ns)

2.2 Lnxsrv 10 Server:

2.2.1 SynchronizedState

Thread Numbers\ Size of state array	State Array of 5 Entries	State Array of 100 Entries	State Array of 500 Entries
1 thread	16.4782	17.0792	17.0357
8 threads	416.005	381.784	378.015
16 threads	822.269	748.738	711.022
40 threads	2169.10	1917.58	1812.27

Avg Swap Real Time (ns)

2.2.2 UnsynchronizedState

Thread Numbers\ Size of state array	State Array of 5 Entries	State Array of 100 Entries	State Array of 500 Entries
1 thread	12.2052	12.1494	12.1538
8 threads	155.489	284.945	216.152
16 threads	309.055	554.762	453.682
40 threads	302.713	587.555	1007.62

Avg Swap Real Time (ns)

2.2.3 NullState

Thread Numbers\ Size of state array	State Array of 5 Entries	State Array of 100 Entries	State Array of 500 Entries
1 thread	10.7991	11.0170	10.9802
8 threads	30.7191	35.1146	29.3972
16 threads	99.2957	76.9164	83.5340
40 threads	173.734	213.865	154.107

Avg Swap Real Time (ns)

2.2.4 AcmeSafeState

Thread Numbers\ Size of state array	State Array of 5 Entries	State Array of 100 Entries	State Array of 500 Entries
1 thread	25.3934	24.9055	24.6413
8 threads	1148.98	526.384	335.158
16 threads	2184.29	1190.92	704.697
40 threads	6006.77	1798.19	1771.75

Avg Swap Real Time (ns)

3. Comparison & Analysis

3.1 Different Test Platforms

Suppose the same state is tested with the same number of threads and the same size of state array. Although lnxsrv06 has more cores and higher CPU frequency than lnxsrv10, the performance of state in lnxsrv10 is better than in lnxsrv06 in general.

However, at the same time, the performance of AcmeSafe state in lnxsrv06 is better than in lnxsrv10. In my deduction, the reason why the states perform better in lnxsrv10 is because the cache size of each processor in lnxsrv 10 (16896 KB) is larger than lnxsrv 06 (12288 KB). And the program didn't use all the cores.

3.2 States

Support the states are running on the same server with the same size of state array and the same number of threads.

In terms of performance, in lnxsrv 10, surprisingly SynchronizedState has better performance than AcmeSafeState. In my guess, the reason comes from the processor architecture, it may be similar as mentioned in 4.1 section. In lnxsrv 06, SynchronizedState has the slowest speed. The reason is using the synchronized method would lock the swap method to wait for threads.

In terms of the reliability, through the results, in both servers, the reliability of NullState and SynchronizedState is 100% as predicted. AcmeSafeState has good reliability, it synchronizes when Read and Write operations are conducted to access memory. And during the test, UnsynchronizedState frequently returns mismatches, which means it has a very low reliability. It has also been predicted since this implementation merely executes multiple threads at the same time.

4. Problems

This homework is a straightforward task. All need to do is gather the results and analyze them. A problem I did meet is the performance of AcmeSafeState and Synchronized both have different performance in two different testing platforms. I cannot explain the reason clearly, finally I got some hints from piazza. The reason comes from the different process architecture of two servers.

5. Conclusion

After careful analysis and testing of the various implementations, The AcmeSafeState implementation has strong reliability and good performance, which is a better way than SynchronizedState implementation.

Reference

[1] Class AtomicLongArray. *In Java SE 13 & JDK 13,* 2019.

https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/concurrent/atomic/AtomicLongArray.html

[2] Doug Lea. *Using JDK 9 Memory Order Modes*. 2018. http://gee.cs.oswego.edu/dl/html/j9mm.html

[3] William Pugh. *The Java Memory Model.*

https://www.cs.umd.edu/~pugh/java/memoryModel/