CS131 HW3 Report

Chuhua Sun
UCLA

Abstract

In Java, Synchronization based on the Java memory model(JMM) is frequently used in applications to avoid data races when accessing shared memory safely. However, sometimes, Synchronization in Java constructs a well-defined data-race free(DRF) environment with the cost of performance. Therefore, breaking the safe simulation provided by Java's Synchronization or applying some other synchronization strategies that are less heavyweight would speed up the application. Those other methods could be unreliable due to breakage or good-enough and faster than using the build-in synchronized keywords.

1 Introduction

For this homework, we are supposed to improve the application for a startup company Ginormous Data Inc(GDI), specializing in finding patterns in large amounts of data. Ideally, we will find and test a better way to perform concurrency with high reliability, improving the current slow application's performance. In particular, we are provided a Java program that will perform an operation called ”Swap” on an array with large size and many times. Specifically, whenever we call the ”Swap” operation, we pass in two indices to the program. The program will then decrease the value stored at the first index by one and increase the value stored at the second index by one. Moreover, the program is designed to be flexible in that users are allowed to set the number of threads to be used, the amount of the ”Swap” operation to be performed, and the size of the array. This program's original operating approach is based on the built-in Java synchronized keyword to ensure reliability but with low speed. We will implement two other techniques to perform the task. The first one will only focus on increasing the speed but ignore the reliability, and the second one is a method that will be both data-race free(DRF) and faster than the original approach. We are also provided with a ”Null” approach that basically does nothing. It will be just used as a comparable reference to the other three methods. In this report, we will measure and characterize all of the mentioned approaches' performance and reliability. In the next section, we will discuss the implementations of different methods in detail. In section 3, we will show the results of running the test harness with different approaches, along with various values for the state array's size and the number of threads. In section 4, we will compare and analyze the different strategies. Finally, we will include a conclusion.

2 Implementation

Besides the provided two classes NullState and SynchronizedState, I implement two other classes UnsynchronizedState and AcmeSafeState. In this section, we will discuss the tails of all of those different implementations in detail.

2.1 Null State

NullState is a class that does nothing. Using this class to perform the ”Swap” operation will have no effect since this class's swap function is empty. Ideally, this class is used for timing the scaffolding of the simulation.

2.2 Synchronized State

Synchronized State is a data-race free(DRF) synchronized class, so it is safe but slow. More specifically, there will be only one thread perform the synchronized operation at a time. All the other threads that invoke the same object's synchronized functions will be blocked and wait for the previous thread to finish its job. Hence, we could consider threads are executing to prevent data races. Since there is no data racing at all in this approach, the result is entirely reliable. However, as everything is performing in order, this approach is slow.

2.3 Unsynchronized State

UnsynchronizedState is a high-speed but unreliable class that is very similar to SynchronizedState class. It is constructed by merely removing the synchronized keyword in SynchronizedState class. Since all the threads can execute simultaneously without blocking, we can perform the operation very fast. However, this implementation also causes a lot of data races, leading to a highly unreliable result.

2.4 AcmeSafe State

AcmeSafeState is a safe class without using the synchronized keyword. The implementation utilizes the java.util.concurrent.atomic. AtomicLongArray package to achieve better performance while retaining safety. In detail, as we are using an AtomicLongArray, we can atomically modify the elements stored in it. For our purpose, we could use the getAndIncrement(index) and getAndDecrement(index) functions from the package to perform our desired ”Swap” operations. The pros of this implementation are its speed and its high reliability compared to the UnsynchronizedState class. Since there is no synchronized keyword involved, there is no thread blocking, and then we can achieve a high-speed. On the other hand, this is not a data-race free(DRF) implementation since we could still incur data racing errors if we failed to apply appropriate restrictions can cause interleaved threads. For instance, we could have a hard-to-replicate error if we don't enforce the required constraints. The result's correctness could be hard to establish when there are not enough constraints to be sure that a given Read has only one possible Write that it could read from.

3 Problems

The major problem that we have to overcome to do the measurements properly is to find the representative values of the number of threads used and the size of the array. Otherwise, it might be hard to figure out the performances and characteristics of the different implementations. Other than the provided values for the number of threads, 1, 8, and 40, I pick 25 as the other value as it is higher than the number of processes of both servers but much lower than the upper bound. Hence it could be treated as a middle value among all of them. For the size, I want to test our implementations' performance in a relatively large array, which might take more effort for the AtomicLongArray to handle. Hence we could see the performance and reliability of different implementations with both small and large arrays.

4 Measurements and Characterizations

For each class SynchronizedState, UnsynchronizedState, and AcmeSafeState, we will measure and characterize the class's performance and reliability by using two SEASnet GNU/Linux servers with Java version 15.0.2 and varying the state array size and the number of threads). The first server, lnxsrv06, has a Intel(R) Xeon(R) E5620 2.40GHz CPU and 16 processors with 4 cores. It has a operating system version of 7.8 (Maipo). The second server, lnxsrv11, has a Intel(R) Xeon(R) Silver 4116 2.10GHz CPU and 4 processors with 4 cores. It has a operating system version of 8.2 (Ootpa).

4.1 Results on lnxsrv06

4.1.1 NullState

Threads/Size	Size: 5	Size: 100	Size: 1000
Threads: 1	14.1707	14.0103	13.9442
Threads: 8	20.5585	23.1298	19.9582
Threads: 25	98.6913	99.3204	119.998
Threads: 40	114.441	190.752	148.528

Real Average Swap Time(ns)

4.1.2 SynchronizedState

Threads/Size	Size: 5	Size: 100	Size: 1000
Threads: 1	30.0821	29.2333	28.8932
Threads: 8	1960.33	2170.68	2177.41
Threads: 25	4923.47	5328.69	5350.44
Threads: 40	8548.57	10116.6	9200.29

Real Average Swap Time(ns)

4.1.3 UnsynchronizedState

Threads/Size	Size: 5	Size: 100	Size: 1000
Threads: 1	16.0253	16.8025	17.1072
Threads: 8	423.504	429.451	515.350
Threads: 25	859.171	968.217	1384.14
Threads: 40	1426.44	1525.18	2178.13

Real Average Swap Time(ns)

4.1.4 AcmeSafeState

Threads/Size	Size: 5	Size: 100	Size: 1000
Threads: 1	22.7772	23.8037	22.3688
Threads: 8	1140.69	839.391	250.612
Threads: 25	2211.19	1818.06	790.600
Threads: 40	3470.58	3108.01	1202.21

Real Average Swap Time(ns)

4.2 Results on lnxsrv11

4.2.1 NullState

Threads/Size	Size: 5	Size: 100	Size: 1000
Threads: 1	11.2740	10.9210	10.9085
Threads: 8	32.0072	27.3983	28.9710
Threads: 25	291.817	92.1990	93.6006
Threads: 40	149.016	153.656	498.772

Real Average Swap Time(ns)

4.2.2 SynchronizedState

Threads/Size	Size: 5	Size: 100	Size: 1000
Threads: 1	32.9884	32.3133	32.4004
Threads: 8	818.038	474.109	532.353
Threads: 25	3546.89	1487.70	1317.96
Threads: 40	4895.32	2464.69	2172.30

Real Average Swap Time(ns)

4.2.3 UnsynchronizedState

Threads/Size	Size: 5	Size: 100	Size: 1000
Threads: 1	12.2369	12.1964	12.5597
Threads: 8	157.471	275.881	184.258
Threads: 25	605.676	838.421	558.137
Threads: 40	849.022	1447.40	1560.48

Real Average Swap Time(ns)

Threads/Size	Size: 5	Size: 100	Size: 1000
Threads: 1	25.8258	25.3339	24.7755
Threads: 8	1158.17	592.755	273.461
Threads: 25	3599.90	1903.42	903.958
Threads: 40	5953.18	3237.58	1527.26

Real Average Swap Time(ns)

4.2.4 AcmeSafeState

5 Comparasion and Analysis

5.1 Overall Performance

First, we compare the overall performance of the two servers. When applying the NullState class, which does not have a real effect, as shown on the table, Inxsrv11 generally has better performance than Inxsrv06 except for running 40 threads with an array with size 1000. For SynchronizedState and UnsynchronizedState, Inxsrv11 also has a much better performance than Inxsrv06, although Inxsrv06 has more processors and higher CPU frequency. However, Inxsrv06 has better performance on AcmeSafeState. One possible explanation for this is Inxsrv11 might have a better ability of context switching and threads management. So even though Inxsrv06 has more processors, it might spend much more time on context switching and thread-relative things than Inxsrv11 does. Hence, Inxsrv11 generally perform better than Inxsrv06 when we are applying NullState, SynchronizedState, and UnsynchronizedState. On the other hand, AcmeSafeState might require much less effort for context switching or thread management. And then Inxsrv06 might be able to better utilize its advantage of more processors, resulting in a better performance than Inxsrv11.

Second, for the four different implementations we discussed, NullState has the best performance since it does not really perform the ”Swap” operation. SynchronizeState is the slowest one among them since thread-blocking significantly slows the execution time of the operation. UnsynchronizedState is faster than AcmeSafeState when the size of the array we use is relatively small. AcmeSafeState almost has the same performance as UnsynchronizedState when the array’s size is getting large, especially when we are using 40 threads. Therefore, it will be very efficient to apply AcmeSafeState in this case.

5.2 Reliability

In this part, we discuss the reliability of the four different implementations. First, NullState is totally unreliable as it does not perform the ”Swap” operation at all. Then SynchronizedState is perfectly reliable since it applies the Java built-in synchronized keyword with thread-blocking. The UnsynchronizedState is highly unreliable since it incurs many data racing errors and results in an incorrect array with significant error. Finally, AcmeSafeState, although not data-race free(DRF), performs with perfect reliability in our test as it does not give any incorrect result.

6 Conclusion

As a conclusion. AcmeSafeState is the best approach among the four different implementations we discussed in this report as it perform in very fast while successfully ensure safety.

References

1. https://web.cs.ucla.edu/classes/winter21/cs131/hw/hw3.html
2. http://gee.cs.oswego.edu/dl/html/j9mm.html