Java Synchronization Reliability and Performance

Adam Cole

1. Overview

In this assignment, I experimented with the Java Memory Model (JMM) and it's synchronization tools that are commonly used in multithreaded programs for my startup company Ginormous Data Inc. (GDI). Our startup wants to test the boundaries between performance and reliability because in our industry, we want to beat our competition to the calculation - and we are willing to decrease our precision to get there. In this experiment I want to measure how often GDI programs are likely to break if we switch to inadequate-but-faster synchronization methods, and find any safe technique that is faster than the 'synchronized' keyword in Java.

2. SEASNet Servers

I performed the tests on two different Linux Servers at UCLA, the SEASNet servers lnxsrv09.seas.ucla.edu and lnxsrv10.seas.ucla.edu. Before beginning the experiment, I reviewed the specifications of both servers using the following:

```
lnxsrv09]$ cat /proc/cpuinfo
...
processors       : 32
model name      : Intel(R) Xeon(R)
CPU E-5-2640 @ 2.00GHz
cpu MHz         : 1394.165
cache size      : 20480 KB
cpu cores       : 8
address sizes   : 46 bits physical,
48 bits virtual
```

By executing this command, I am able to see that the lnxsrv09 server allocates 32 processors to me. Each of these processors have 1.39 GHz processing power with 8 cores. At first glance, I assume lnxsrv09 will be a very fast server since I am allocated 32 processors to run my multithreaded code.

```
lnxsrv10]$ cat /proc/cpuinfo
...
processors       : 4
model name      : Intel(R) Xeon(R)
Silver 4116 CPU @ 2.10GHz
cpu MHz         : 2095.079
cache size      : 16896 KB
cpu cores       : 4
address sizes   : 44 bits physical,
48 bits virtual
```

On the lnxsrv10 server, however, we see a slightly different story. Here, I am only allocated 4 processors with 2.09 GHz processing power with 4 cores. The Xeon Silver 4116 is a 12 core CPU, but only 4 cores show up. This leads me to believe that lnxsrv10 is running a Virtual Machine (VM) with only 4 CPUs allocated to it.

Looking at the executing hardware in each of these servers, it seems as though lnxsrv09 should execute much faster.

3. Synchronization Methods

I researched Java synchronization methods that could make our GDI programs unreliable and reliable - but most importantly fast. I then attempt to compare these methods to the current method, using the 'synchronized' keyword

3.1 SynchronizedState.java

I created SynchronizedState.java to be a class that was safely but slowly synchronized with keyword 'synchronized' - but that also had the potential to create Data Race Conditions (DRCs) in our shared memory structures. All state classes were given the potential to create DRCs by including a swap() function which incremented and decremented values in shared long[] arrays. The synchronized keyword internally uses a lock-based system to ensure atomicity.

3.2 UnsynchronizedState.java

UnsynchronizedState.java seeks to test how often our GDI programs will break when we try to reach performance ultimately over reliability.

In order to create a standardized level of comparison, I used the same functionality in SynchronizedState and UnsynchronizedState, only differing in the synchronized keyword. In UnsynchronizedState, I do not include the synchronized keyword.

3.3 AcmeSafeState.java

AcmeSafeState.java was my solution to finding a reliable and efficient synchronization technique to ultimately improve our GDI program's efficiency with minimal precision loss. I implemented AcmeSafeState.java by using a private AtomicLongArray from java.util.concurrent.atomic.AtomicLongArray. Using this data structure instead of a regular long[], we can access the AtomicLongArray atomically and inherently through member functions instead of using Java's synchronized keyword.

4. Testing and Results

To put my synchronization experiments to the test, I ran it on a test file called UnsafeMemory with parameters passed to it encapsulating the number of threads to be used in the multithreaded program, the number of iterations that swap() will be called, the number of shared long arrays, and the synchronization method (Unsynchronized, Synchronized, or AcmeSafe). In my testing I used each combination of server (lnxsrv09 or lnxsrv10), threads (1, 8, 16, or 40), number of arrays (5, 100, or 300), synchronization method (Unsynchronized, Synchronized, or AcmeSafe), and kept the number of swap() iterations constant at 100,000,000. In each of the trials, the cells represent nanoseconds / CPU instruction.

Lnxsrv09, N=5

#Threads	1	8	16	40
Synchron.	20.72	1626	4687	9001
Unsync.	14.88	375.9*	622.4*	1055*
AcmeSafe	30.17	2264	3586	10194

Lnxsrv10, N=5

#Threads	1	8	16	40
Synchron.	17.66	402.4	791.9	2019
Unsync.	12.09	273.8*	559.6*	1060*
AcmeSafe	28.03	507.3	1026	2592

Lnxsrv09, N=100

#Threads	1	8	16	40
Synchron.	20.49	1595	4222	11275
Unsync.	15.07	347.1*	608.9*	1197*
AcmeSafe	29.88	2647	4429	12982

Lnxsrv10, N=100

#Threads	1	8	16	40
Synchron.	17.24	398.0	770.3	1899
Unsync.	12.14	278.0*	585.2*	1384*
AcmeSafe	28.28	520.9	1040	2573

Lnxsrv09, N=300

#Threads	1	8	16	40
Synchron.	20.78	2071	4840	12608
Unsync.	14.83	244.8*	403.0*	835.7*
AcmeSafe	29.78	2576	5282	14522

Lnxsrv10, N=300

#Threads	1	8	16	40
Synchron.	17.11	349.2	684.0	1792
Unsync.	12.16	225.9*	449.7*	1169*
AcmeSafe	28.03	515.5	1048	2649

* = Mismatched Sum (Data Race Condition)

5. Analysis

After running the synchronization methods against the various parameters, the results were clear. UnsynchronizedState.java proved to be the most efficient by far, but broke almost every run. The cases where it didn't break were when UnsafeMemory was run on only 1 thread, guaranteeing no Data Race Conditions would occur. From the perspective of GDI, we must be careful making a transition to unreliant unsynchronized multithreaded programs, because the mismatched sum widely varied (sometimes the result was off by a little, sometimes by a lot). For this reason, Unsynchronized multithreaded programs are unreliable beyond use.

SynchronizedState.java proved to be the next fastest in comparison, and not AcmeSafeState.java. While I believed that using an AtomicLongArray would be faster than 'synchronized' keyword's lock system, it proved to run steadily ahead in all comparisons. From a GDI perspective, 'synchronizing' our programs still proves to be the most optimal route. While less efficient than unsynchronized, it provides accurate results free of Race Conditions and faster than other synchronization methods, like AcmeSafeState.java implements.

6. Conclusion

In conclusion, the startup company GDI should continue to use the synchronized programs implemented with the Java 'synchronized' keyword. It is currently the fastest synchronization method to accurately report information, so while we would like to improve performance, more research into this field is required to find a better solution.

While the UnsynchronizedState.java implementation ran much more efficiently than SynchronizedState.java did, it came at a price of accuracy. While GDI is willing to lower their precision to increase performance, the result's accuracy varied too much to implement into our company.