CS 131 Homework 3: Java Shared Memory Performance Races

David Chen, *UCLA*

Abstract

This test was done in order to compare the tradeoffs between different java concurrency methods or lack thereof. This was tested a on a program that swaps numbers by way of increment and decrement operators. Notably, this experiment tests ‘synchronized’, ‘java.util.concurrent.atomic.AtomicIntegerArray’, and ‘java.util.concurrent.locks.ReentrantLock’, with an unsynchronized implementation and a dummy implementation as controls.

1. Test Environment

Java Version:

- openjdk version "1.8.0_161"
- OpenJDK Runtime Environment (build 1.8.0_161-b14)
- OpenJDK 64-Bit Server VM (build 25.161-b14, mixed mode)

CPU & Memory:

- Server: SEASNet lnxsrv06
- CPU: Intel(R) Xeon(R) CPU E5620
- Clock Speed: 2.40GHz
- Memory:
    - MemTotal: 65796288 kB
    - MemFree: 54306468 kB
    - MemAvailable: 64586516 kB
- Cores: 16

2. Testing Procedure

2.1. Test Subjects

- Null: dummy implementation
- Synchronized: uses synchronized
- Unsynchronized: same as synchronized, but without synchronization
- GetNSet: uses exclusively get() and set() from AtomicIntegerArrays
- Better Safe: uses ReentrantLock

2.1. Test Methods

Testing was done through a test script for most of the subjects, excluding Unsynchronized and GetNSet, due to the possibility of hanging from data races. For these, testing was done individually, with a truncated set of variables. For the rest, each method was tested using a variation of 1, 2, 4, 8, 16, or 32 threads; 1,000 100,000, or 100,000,000 swap operations and 6 or 16 elements on the array.

3. BetterSafe

When choosing the implementation methods for better safe, the two main considerations were its reliability and its performance, namely being better than synchronized. This condition was considerable slackened after the clarification was made that BetterSafe did not necessary have to be better than Synchronized in all aspects but in a specific case. Both atomics and locks could possibly improve upon Synchronized, especially in case of contention, but considering the prior use of atomics for GetNSet, I decided on using locks, as sources have indicated the advantage of specifically ReentrantLock over Synchronized in cases where there is high contention. [1] Meanwhile, VarHandles, while mentioned by the TA, seemed to be a more generic version of AtomicInteger, which in turn related to AtomicIntegerArrays. Due to the lack of success in GetNSet – although that may be more attributed to use of get() and set() as two separate atomic functions – I decide to look into this if locks ended up not serving their purpose.

4. Test Results

A sample of the test results are as follows:

This was run with 1000000 swaps and 16 array elements. It should be noted that Unsynchronized would almost always result in an inaccurate sum and GetNSet would also often exhibit this behavior.

Method	**Average runtime/transition (ns) for n thread(s)**		
	n = 1	**n = 2**	**n = 4**
Null	35.1247	144.767	384.445
Synchronized	60.9971	336.124	1594.98
Unsynchronized	48.5342	224.254	519.658
GetNSet	68.2965	172.431	1122.17
BetterSafe	66.5669	765.943	557.484
	n = 8	**n = 16**	**n = 32**
Null	2624.66	2317.30	7007.39
Synchronized	2407.18	4737.82	10380.4
Unsynchronized	2218.14	4235.54	9172.30
GetNSet	2370.43	5350.53	18795.5
BetterSafe	870.040	1925.00	4276.37

4.1. Reliability & Performance

It should be noted here that the main points of focus should be Synchronized and BetterSafe. Synchronized initially outperforms BetterSafe at low numbers of threads, but quickly loses as the number of threads. This is because locks perform much better than synchronized in case of high contention, such as when multiple threads are competing for one resource.

4.2. Data Race Free

Synchronized and BetterSafe are both data race free(DRF), as they maintain total ordering of locked regions. Specifically, synchronized guarantees the block can only be accessed by one thread at one time. However, this reliability is also what causes its drop in performance in comparison to the other methods. BetterSafe does the something but in a more lightweight manner, which is why it performs much better, especially at levels of high contention.

GetNSet and Unsynchronized, however, are both not DRF. They break at times – Unsynchronized significantly more than GetNSet—especially at high thread counts and when the array size is low. Examples of such cases are as follows:

```
java UnsafeMemory Unsynchronized 32 1000000 6 5 6 3 0 3
java UnsafeMemory GetNSet 32 1000000 6 5 6 3 0 3
```

Unsynchronized lacks any sort of protection, thus threads can access and modify the array elements all at once, causing rampant errors in the array sum and corrupting the array. GetNSet has similar issues, mainly in that get(), set() and the increment and decrement operator are all separate, meaning threads could run them at the same time, causing race conditions.

4.3. Why BetterSafe?

BetterSafe uses ReentrantLocks to lock the threads in the same critical section as Synchronized, making it thread safe in the same manner, but due to its use of locks, operate better at higher contention. Additionally, its improved performance is seen in the conditionals, as it can unlock earlier given a condition. This, while seemingly insignificant, is slightly less coarse than synchronized, which acts upon the entire function.

5. Conclusion

Given the results seen in the experimentation, if reliability was of utmost importance, BetterSafe would be the best choice as it performs much better than Synchronized. If some error is allowed, GetNSet may be considered, but is still quite unreliable; however, the amount of data it acts upon is rather large, so given a comparatively smaller number of threads, it's feasible to use this.

6. References

[1]http://lycog.com/concurency/performance-reentrantlock-synchronized/