Homework 3. Java shared memory performance races

Introduction

The synchronized in Java can help us make sure that a program thread-safe when running on multithread. However, the performance of synchronized is not satisfying. Therefore, we come up with some other programming methods: Unsynchronized and AcmeSafe. We also analyzed the performance and reliability of these methods.

Analysis of Packages

There are four packages that could be implemented for AcmeSafe. This part looks at the pros and cons of each package and discusses the final package choose of the study.

1.1. java.util.concurrent

The pros of this package is that it could enforce a strong memory order mode with a specific ordering by ranking the threads and let the highest ranked thread run first. In this case, the memory consistency is stronger. However, the cons is that the complexity of this method becomes very high. Therefore, this pacakage is not necessary. I choose not to utilize this package.

1.2. java.util.concurrent.atomic

This package contains classes that has no locks and volatile access on the elements. The package also limits the types of variables to Booleans, Integers, Longs, and Object Reference. Moreover, this package provide atomic access to the data but it doesn't resolve racing conditions. Specifically, though the read and write are not interrupted, it doesn't make sure that element won't be changed by other threads. It is used in the Acmesafe file but the atomic operation is not used. I used AtomicIntegerArray, in this package, the entire array is locked when a thread is going to make a change. It works equally effective as locking the swap method which only increase and decrease values in the arrays. Therefore, I choose to opt with atomicIntegerArray instead of atomic.

1.3. java.util.concurrent.locks

This package provides a framework which has various type of lock and wait conditions. The lock in this package has the same basic behavior with the implicit monitor lock accessed using synchronized methods. But one pro of this method is that it provides a simple and reliable lock to synchronize threads. It is faster than the Synchronized implementation. Specifically, the ReentrantLock, in this case, it locks the race condition instead of the entire methods. when a thread involes lock, it will return and acquire a lock when the lock is not owned by another thread, otherwise, it will return immediately. Therefore, it frees the CPU for other threads that acquired or doesn't need to acquire lock.

1.4. java.lang.invoke.VarHandle

This package is very similar to atomic. It allows for interrupts. The pro is that we are able to synchronize many different types of elements that aren't just integers. But since it allows for interrupts which could potentially break sequential consistency and we only process integers. I choose not to use this package.

Analysis of Classes Implementations

2.1. Synchronized

Synchronized is a given method which provides thread synchronization. This method ensures that only one thread is running while other threads are blocked. Since the method blocks the entire swap methods, it is DRF. It is great in terms of reliability, however, its performance is very slow. It locks the entire swap method, therefore, it is not actually taking advantage of multithreading since other threads will be forced to wait. Moreover, the method is not supporting fairness, which means it could potentially cause starvation.

2.2 Unsynchronized

This is an implementation of the State class that doesn't use the synchronized keyword. It is exactly the opposite of Synchronized class. Multiple threads can execute swap at the same time while no lock is applied. In that case, the race condition could happen so it is not DRF. It is possible that the thread read and write incorrect values.
However, As far as performance is concerned, this is the most efficient implementation as there is no locking or blocking of threads.

2.3 AcmeSafe

AcmeSafe class is designed a half way between Synchronized and Unsynchronized class. It uses java.util.concurrent.atomic.AtomicIntegerArray class and the associated get and set methods as opposed to

using the synchronized. Since there is no Synchronized used, multiple threads can run the swap method at the same time. However, as increment and decrement consist of two operations (read and write), they need to be handled in a single atomic operation (which get and set methods don't do). Therefore, it is not DRF since the race conditions could happen when thread are interleaved.(Command: *java UnsafeMemory AcmeSafe 8 10000 127 12 3 4 56*) In terms of performance, it is faster than synchronized since it only synchronizes the value of the elements instead of locking the entire method.

Performance and reliability results

We compare Synchronized, Unsynchronized and AcmeSafe with different threads, swaps and elements.

Table1. Time for Transitions with 10,000 swaps and 20 elements

Memory Order Mode	1 thread	4 thread	8 thread	DRF
Synchronized	1111.59	5375.84	12249.4	yes
Unsynchronized	679.115	4802.46	6739.45	No
AcmeSafe	939.405	5117.03	8486.64	No

Table2. Time for Transitions with 8 threads and 50 elements

Memory Order Mode	10000 swaps	100000 swaps	1000000 swaps	DRF
Synchronized	14613.7	3904.19	1220.17	yes
Unsynchronized	9352.35	2400.98	960.676	No
AcmeSafe	12602.8	2907.48	1183.58	No

Table3. Time for Translations with 1000,000 swaps and 8 threads

Memory Order Mode	4 elements	40 elements	400 elements	DRF
Synchronized	4023.92	930	2839.78	yes
Unsynchronized	3544.69	1319.02	1562.56	No
AcmeSafe	3421.21	1765.98	4256.51	No

3. Analysis

From the table we can tell that at most of the time, Synchronized is the slowest method and it is not surprising because Synchronized method is blocking the entire swap method to wait for threads. Moreover, one situation that is not showed from the tables is that Unsynchronized method always return mismatch which means the reliability of the Unsynchronized method is also very low. It is also not surprising because it just lets threads run and execute without locking anything. In contrast, Synchronized method shows 100 percent reliability. In terms of AcmeSafe, we can tell that the performance is better than synchronized at most of the time because it avoids synchronized key word. Moreover, it is more reliable than Unsynchronized because it does synchronized when read and write to access memory form data structure.

Some surprising points in these tables are when the element changes and others hold. Synchronized method perform better than others in terms of forty elements and perform better than AcmeSafe in terms of four hundreds elements.

Discussion & Conclusion

After careful analysis and testing of the various implementations, the AcmeSafe implementation is relatively reliable and has faster performance.

Reference

[1]https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/atomic/AtomicIntegerArray.html

[2]https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/package-summary.html

[3]https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/atomic/package-summary.html

[4]https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/locks/package-summary.html

[5]https://docs.oracle.com/javase/9/docs/api/java/lang/invoke/VarHandle.html

Java Version:

openjdk version "13.0.1" 2019-10-15
OpenJDK Runtime Environment (build 13.0.1+9)
OpenJDK 64-Bit Server VM (build 13.0.1+9, mixed mode, sharing)

Meminfo:

MemTotal: 65799636 kB
MemFree: 729388 kB
MemAvailable: 13980476 kB
Buffers: 446692 kB
Cached: 11648236 kB

Cpuinfo:

cpu family : 6
model : 85
model name : Intel(R) Xeon(R) Silver 4116 CPU @

2.10GHz
cpu MHz : 2095.079
cache size : 16896 KB