Java Shared Memory Performance Races

CS 131 report
University of California, Los Angeles (UCLA)

Abstract

The purpose of this report is to investigate different concurrent mechanisms in Java with a simple prototype that manages a data structure that represents an array of integers. A state transition, called a swap, consists of subtracting 1 from one of the positive integers in the array, and adding 1 to an integer that is less than maxval. The sum of all the integers should therefore remain constant. I will also talk about its performance and reliability under different threads as well as the pros and cons for the types of lock.

1 BetterSafe

Package java.util.concurrent
Pros: utility classes commonly useful in concurrent programming. This has thread pools, asynchronous I/O, and lightweight task frameworks.
Cons: This package provides a lot of functionality but it does not specifically talk about in the condition of read and write.
Package java.util.concurrent.atomic
Pros: a small toolkit of classes that support lock-free thread-safe programming on single variables. Instances of Atomic classes maintain values that are accessed and updated using methods
Cons: This only provide thread-safe programming on single variable; however, we may want to work with different data types.
Package java.util.concurrent.locks
Pros: locking and waiting for conditions that is distinct from built-in synchronization and monitors. The framework permits much greater flexibility in the use of locks and conditions, at the expense of more awkward syntax.
Cons: This seems like what I wants for my case.
Package java.lang.invoke.VarHandle
Pros: A VarHandle is a dynamically strongly typed reference to a variable, or to a parametrically-defined family of variables, including static fields, non-static fields, array elements, or components of an off-heap data structure.
Cons: This seems to be a good fit for our case but I will try the lock first.

I decide to use ReentrantLock because this locks may be shared among readers but are exclusive to writers This is what I want in our simple data structure array where we allow thread to increment and decrements.
Implementation:
I created a class with a simple byte array, which this is very similar to the previous Unsynchronized class, all I added to my new class was having a ReentrantLock in the swap.
In the swap where all the magic happens, I lock when the thread come in or give the lock to the thread, the thread will do the swapping and then release the lock to next thread. This guaranteed that no thread will interrupt the process of swapping.

2 Benefit

The table 2 shows the performance between BetterSafe and Synchronized.

Mechanisms	8 threads	16 threads	32 thread	DRF
Synchronized	2.91	7.49	2.62	Yes
BetterSafe	6.9	1.20	2.52	Yes

Table 1: Every result here is in ns/transition

why my BetterSafe implementation is faster than Synchronized and why it is still 100 % reliable?
For obvious reason, my BetterSafe is faster than the Synchronized based on the statistic above.
Why Synchronized is slower?

The first reason can be that Synchronized is a synchronized block or method you just need to write synchronized keyword (and provide associated object) acquiring lock and releasing it is done implicitly; however, BetterSafe is using ReentrantLock during the swap. It explicitly specifies when to give the lock to the thread and releasing the lock when it done. This might have a advantage over Synchronized in which one thread need to wait until the previous done the execution. This difference between ReentrantLock and Synchronized may not seem so obviously if you run less threads in both of them; however, ReentrantLock performs much faster than the traditional 'lock' synchronized, because next thread will immediately know when the previous release the lock.
Does BetterSafe 100 % reliable?
Yes, because it use lock to lock when a thread comes in to do the swapping and it will release its lock to another thread when it finishes. This prevents any interruption.

3 Test Result and Comparison

The table 2 shows all the mechanisms in concurrent programming.

Mechanism	8 threads	16 threads	32 threads	DRF
Null	4.0	9.30	2.48	No
Synchronized	2.91	7.49	2.62	Yes
Unsynchronized	4.06	9.25	2.50	No
GetNSet	4.30	9.54	2.29	No
BetterSafe	6.9	1.20	2.52	Yes

Table 2: All results here are in ns/transition

Null performance is very similar to rest of the mechanisms in both 8 threads and 32 threads; however, there is a slight catch here, it performs much slower. The implementation of Null does have involve with any of adding and subtracting in compassion with other classes. It just return true.
Unsynchronized performs much similar to GetNSet in a sense that they both do not care about race conditions, this is one of the reasons why sometimes they may run into deadlock or sum mismatch. For both of these mechanism, they want to go as fast as they can; however, the catch here is if there is current thread does not complete its operation before the next thread, they will run into each or step on each other. This could be a big trouble in parallelism, because they may overwrite the current value. They are DRF, because they occur without any synchronization. When I run the java command, these two mechanism have the same problem of sum mismatch. It is a problem to debug, because it mess up the array operation and it is difficult to keep track on.
Threads average 2.50195e+07 ns/transition
sum mismatch (35 != 34)
BetterSafe and Synchronized are much similar as we have discussed in the previous section. They use different lock mechanisms. BetterSafe implements use explicit Reentrantlock in which allows to lock the thread and unlock the swap once it complete the current operations; however, Synchronized uses the built-in keywords to do the similar actions, except thread implicitly release the lock to the next thread. They are DRF, because they do not run into race conditions.

4 Problems

It is kind of difficult to choose what mechanism to use in order to achieve better performance while ensuring reliability. I spent a lot of time reading all the option that the homework specs and list pros and cons for all of them, which allowed me to narrow down options.

References

https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/package-summary.html
http://gee.cs.oswego.edu/dl/html/j9mm.html

Notes

[1] Remember to use endnotes, not footnotes!