HW3: Java shared memory performance races

Chengyu Wang
204613387

1. Four implementation ideas

In this part we will try to compare four packages and find the one that provides 100% reliability and good performance.

1.1. java.util.concurrent

In this package, I found the class called Semaphore useful. It maintains a set of permits to access the resources so that each requester thread needs to call acquire() method before accessing the resource. It is a relatively high-level synchronization mechanism. If all we need is just a mutual exclusion of resources, this class might be overkill.

1.2. java.util.concurrent.atomic

This package supports lock-free thread-safe programming on single variables. However, if we are dealing with some complex data structures such as arrays, this package might not provide 100% reliability because it can only provide the thread safety of single variable but not the entire data structure.

1.3. java.util.concurrent.locks

This package provides a framework that has more flexibility in the use of locks and conditions. With the ReentrantLock class in this package, we can control the access of a specific piece of code. Thus, we can ensure the safety of a thread. The drawback of this package is that it has an awkward syntax.

1.4. java.lang.invoke.VarHandle

This package provides a class called VarHandle that extends the Object Class. It is a dynamically strongly typed reference to a variable or to a parametrically-defined family of variables. And access to these variable will need different access modes. This is a very complicated class and might be overkill for easy operations like in our problem.

2. BetterSafe

I use reentrant lock for the implementation of BetterSafe. The actual use of reentrant lock here is similar to the keyword “synchronized”, but the lock itself’s implementation is designed to be more scalable and give better performance especially when the number of threads is large. In my test case, I also find an interesting fact that when the number of successful transitions requird is small, the Synchronized class might have better performance. I think this might be due to the fact that the use of reentrant lock requires some extra resources. Thus, BetterSafe is faster than Synchronized especially when the number of operations is large and the number of threads is large. Meanwhile, BetterSafe ensures 100% reliability through locking access into dangerous blocks that might have race conditions in certain time, and unlocking the access when the operations are finished.

3. Unsynchronized

This class is not data-race free (DRF) because it does not have the keyword “synchronized” before the swap function. It is 100% not safe. When testing this class, the program might lead to two situations. The first is that an error message shows that the sum does not remain constant. The second situation is that the program goes into an infinite loop. When the number of successful transitions required is small, the first situation will happen. When the number of successful transitions required is large, the second situation will happen.

4. Synchronized

When testing the cases on Synchronized class, the correctness is 100%. But its speed is much slower than the BetterSafe. The reason has already been state before.

5. GetNSet

This class is not data-race free (DRF). It is intended to avoid race competitions by using AtomicIntegerArray with volatile access to each array element. However, this does not ensure 100% correctness. For example, let Arr[0] = 10. The first thread reads the value from memory, and then it is blocked. And then the second thread reads the value from memory, please notice that at this time thread 1 only reads that value but has not modified the value yet. Thus, thread 1 and thread 2 both do the increments on the original value and write 11 to the memory. However, the expected value should be 12. When I test this class on Seasnet, the result is similar to Unsynchronized class. That is, when the number of

operations is low, there will be incorrect answers; when the number of operations is high, the program will go to infinite loop.

6. Test Case

java UnsafeMemory Synchronized 8 1000000 6 5 6 3 0 3
Theads average 2556.03 ns/transition

java UnsafeMemory BetterSafe 8 1000000 6 5 6 3 0 3
Theads average 1258.02 ns/transition

java UnsafeMemory Unsynchronized 8 1000000 6 5 6 3 0 3
Infinite loop

Java UnsafeMemory GetNSet 8 1000000 6 5 6 3 0 3
Infinite loop
……
java UnsafeMemory Synchronized 16 1000000 6 5 6 3 0 3
Theads average 5823.69 ns/transition

java UnsafeMemory BetterSafe 16 1000000 6 5 6 3 0 3
Theads average 3231.88 ns/transition
……
java UnsafeMemory Synchronized 8 100000 6 5 6 3 0 3
Theads average 6666.34 ns/transition

java UnsafeMemory BetterSafe 8 100000 6 5 6 3 0 3
Theads average 5515.07 ns/transition

java UnsafeMemory Unsynchronized 8 100000 6 5 6 3 0 3
Infinite loop

Java UnsafeMemory GetNSet 8 100000 6 5 6 3 0 3
Infinite loop
……
java UnsafeMemory Synchronized 16 100000 6 5 6 3 0 3
Theads average 12944.0 ns/transition

java UnsafeMemory BetterSafe 16 100000 6 5 6 3 0 3
Theads average 10435.1 ns/transition
……
java UnsafeMemory Synchronized 8 10000 6 5 6 3 0 3
Theads average 10057.3 ns/transition

java UnsafeMemory BetterSafe 8 10000 6 5 6 3 0 3
Theads average 14563.3 ns/transition

java UnsafeMemory Unsynchronized 8 10000 6 5 6 3 0 3
Threads average 8381.26 ns/transition
sum mismatch (17 != 16)

Java UnsafeMemory GetNSet 8 10000 6 5 6 3 0 3
Infinite loop
……
java UnsafeMemory Synchronized 16 10000 6 5 6 3 0 3
Theads average 19152.9 ns/transition

java UnsafeMemory BetterSafe 16 10000 6 5 6 3 0 3
Theads average 31010.9 ns/transition

java UnsafeMemory Unsynchronized 16 10000 6 5 6 3 0 3
Infinite loop

Java UnsafeMemory GetNSet 16 10000 6 5 6 3 0 3
Infinite loop
……
Java UnsafeMemory GetNSet 8 1000 6 5 6 3 0 3
Infinite loop

Java UnsafeMemory GetNSet 8 100 6 5 6 3 0 3
Threads average 308062 ns/transition
sum mismatch (17 != 16)

7. Conclusion

(1). When the number of operations is larger than 10000, BetterSafe will give better performance than Syncrhonized and the difference becomes larger as the threads and operations increase. They are both 100% reliable.

(2). When number of operations is smaller than 1000, GetNSet will generate incorrect answers instead of going to infinite loop. This boundary line for Unsynchronized is 10000. Both GetNSet and Unsynchronized are 100% unreliable.