CS 131 Homework 3. Java shared memory performance races

Nikola Samardzic
University of California, Los Angeles

Abstract

This report is part of my submission for homework 3 of prof. Paul Eggert's Computer Science 131 – Programming Languages class for Fall 2017. It relates to multithreading performance and race conditions encountered in Java libraries used for concurrency.

1 Java Concurrency Libraries

The four suggested libraries for implementing a more concurrent solution were:

1. `java.util.concurrent`
2. `java.util.concurrent.atomic`
3. `java.util.locks`
4. `java.lang.invoke.VarHandle`

The `java.util.concurrent` libraries contains the implementation of `synchronized` statements. If an object is visible to more than one thread, the `synchronized` statement helps us lock access to that variable (and only that variable) within a specified block (usually during reads/writes to that variable). This is what I used for my implementation described in the *Final Implementation* section of this report.

The `java.util.concurrent.atomic` library provides utilities that atomize many instruction on variables. It also provides special array wrapper functions that allow atomization of specific array elements. Although this library provides a robust framework for modifying variables without running into race conditions, it was not flexible enough for the problem at hand. Therefore, I decided not to use it.

The `java.util.locks` library provides a way to lock code (versus locking specific resources). Locks allow us to lock any code segment and works well with branched statements. When a lock is taken by a thread, no other thread will be able to enter the code segments between the lock and unlock statement. Only once the lock is returned (i.e. the thread that had the lock reaches an `unlock()` call) will another thread be able to take the lock and run the code under lock. This obviously provides a good framework for eliminating race conditions, but is too broad of a lock mechanism for us. We need something that can lock specific variables and not lines of code.

The `java.lang.invoke.VarHandle` library is useful for atomicizing primitive variables. I decided to forgoe using it and create my own class that wraps around Java's `byte` type (`MutableByte`). I did this mostly because using `VarHandle` would be overkill.

2 The Final Implementation

First of all, notice that the entire swap array need not be locked for the swap to be executed in a thread safe manner. The only resources accessed/modified during a swap are the i^{th} and j^{th} array element. Therefore, we only need to lock access to these two elements. This approach will allow other thread to swap array elements freely (as long as they are not the same elements some other thread has already locked). In Java, this can be accomplished with the `synchronized` keyword.

This approach doesn't allow any threads to write/read data at the same time, but there is still one issue to worry about: We need to be sure that the code won't run into a deadlock. Since we need to lock both array elements passed to swap before we do anything to them, we have to think about in which order we are going to lock. If we lock the first element first, then we might have a problem: Imagine that swap is called in two different threads on pairs (i, j), (j, i). This situation might cause a deadlock: the first thread might have the i^{th} element locked, while the second thread might have the j^{th} element locked. How can we get around this?

The solution is to always lock the smaller element first

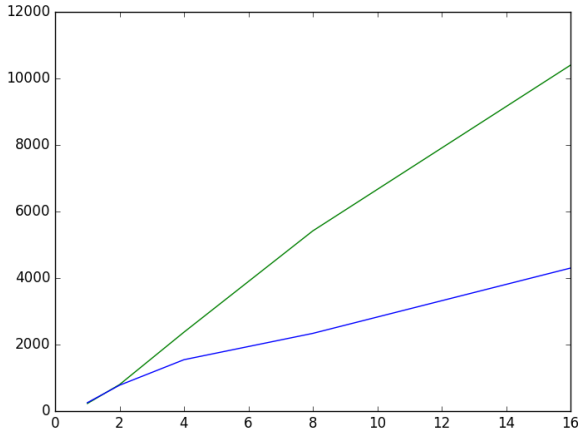

Figure 1: Graph of latency (ns/swap) vs. number of threads running. Again, the new implementation is in blue, and the old one is green.

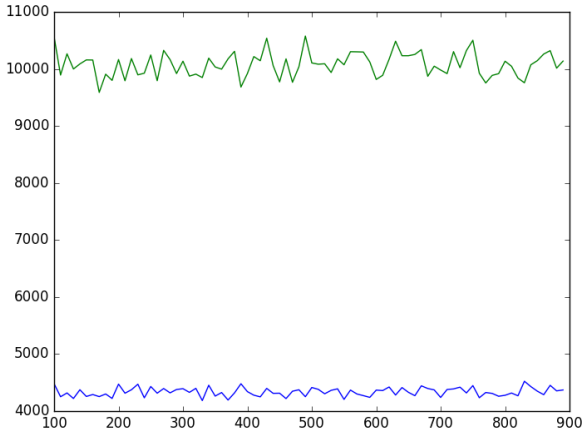

Figure 2: Graph of swap latency (ns/swap) vs. array size. The line in blue is the new implementation, while the green line is the old one. This graph demonstrates that our implementation has consistantly better latency on varying array sizes.

(note that the test script will never pass identical elements).

```
boolean swap(int i, int j) {
    if (i < j) {
        synchronized (array[i]) {
            synchronized (array[j]) {
                // do the swap
            }
        }
    } else {
        synchronized (array[j]) {
            synchronized (array[i]) {
                // do the swap
            }
        }
    }
}
```

Let's try and sketch out why this makes deadlocks impossible. First, define a deadlock as a state of a process that contains a set of threads and resources locked by those threads in such a way that not a single resource can be unlocked.

Assume the contrary, *i.e.* assume that a deadlock can occur for some set of threads and associated locks. Let the j^{th} element be the one with the biggest index that is currently locked. If such j does not exist, then no elements are locked and there is no deadlock. So, we have two options:

1. j is locked by a thread that also holds a lock on another element (i). In this case, the thread can finish it's swap without waiting and there is no deadlock.

2. j is locked by a thread that is waiting for another element to get unlocked (let this be the i^{th} element). Then $j < i$ (since a thread always locks the smaller index element first). However, the i^th element cannot be locked, since j is the biggest locked element. Contradiction.

These two cases exhaust all options. So, we are done.

Why is this program faster than locking the entire `swap()` function? Obviously, if two threads call swap with pairs (i, j) and (k, l) such that all of i, j, k, and l are distinct, we can run two calls to `swap()` concurrently. The probability that all of i, j, k, and l will be distinct assuming a uniform probability of choosing either index is $\frac{1}{N^4}\binom{N}{4}$, where N is the size of the state array. Since this number is pretty close to 1 for even smaller values of N, we know that most calls to `swap()` will be concurrent. This was not the case with the previous implementation, where each call to `swap()` locked all other threads from calling that function with probability 1.

Table 1: **Performance Results.** Performance measurements on various array sizes (Length), swap numbers (Swaps), threads running (Thrds), and implementations (Class). The Null implementation just measures the overhead of the testing harness. The Synchronized implementation measures the performance of the old code, and BetterSafe measures the performance of the new code (both with the testing harness overhead).

Class	Thrds	Swaps	Length	ns/swap
Null	2	10^5	10^2	465.6326
Null	4	10^5	10^2	731.2623
Null	8	10^5	10^2	1435.772
Null	2	10^6	10^2	108.45987
Null	4	10^6	10^2	400.7666
Null	8	10^6	10^2	1998.479
Null	2	10^5	10^3	435.0688
Null	4	10^5	10^3	667.5677
Null	8	10^5	10^3	1472.597
Null	2	10^6	10^3	104.25525
Null	4	10^6	10^3	413.3709
Null	8	10^6	10^3	2067.881
Sync	2	10^5	10^2	855.2863
Sync	4	10^5	10^2	2336.024
Sync	8	10^5	10^2	5383.923
Sync	2	10^6	10^2	353.9067
Sync	4	10^6	10^2	1409.992
Sync	8	10^6	10^2	2846.775
Sync	2	10^5	10^3	830.4144
Sync	4	10^5	10^3	2535.192
Sync	8	10^5	10^3	5412.758
Sync	2	10^6	10^3	335.4249
Sync	4	10^6	10^3	1237.7328
Sync	8	10^6	10^3	2968.002
BetterSafe	2	10^5	10^2	819.7651
BetterSafe	4	10^5	10^2	1468.635
BetterSafe	8	10^5	10^2	2318.607
BetterSafe	2	10^6	10^2	363.6185
BetterSafe	4	10^6	10^2	733.4225
BetterSafe	8	10^6	10^2	1412.724
BetterSafe	2	10^5	10^3	886.18
BetterSafe	4	10^5	10^3	1521.752
BetterSafe	8	10^5	10^3	2309.512
BetterSafe	2	10^6	10^3	274.4405
BetterSafe	4	10^6	10^3	752.3692
BetterSafe	8	10^6	10^3	1430.531