Sequential Consistency in Java

Jeffrey Xu
University of California, Los Angeles

1 Test Results

I used lnxsrv06 to perform my tests, which has Java version 9.0.1, 15 Intel Xeon E5620 CPUs running at 2.4 GHz, and 65.8 GB of memory.
All tests were run on a list of four elements in an attempt to maximize the number of threads competing to access a single element while not guaranteeing that every thread conflicts with another. Every test used 16 threads to maximize the effect of synchronization overhead and 10,000,000 iterations to amortize any constant overhead. Ten trials were run for each synchronization method.

Null	460 ± 50 ns / iteration
Synchronized	3070 ± 80 ns / iteration
Unsynchronized	N/A
GetNSet	3940 ± 30 ns / iteration
BetterSafe	1740 ± 30 ns / iteration

No inconsistencies in the results indicative of a data race were detected in any test that completed.
Unsynchronized cannot be compared using this test, because with this many threads and iterations, Unsychronized causes so many data races that it is nearly guaranteed for every value in the array to go to the maximum value, preventing any further swap operations from taking place. Thus, repeating the tests with 10 trials, 4 array elements, 16 threads, and only 10,000 iterations:

Null	14100 ± 500 ns / iteration
Synchronized	15100 ± 300 ns / iteration
Unsynchronized	14300 ± 500 ns / iteration
GetNSet	26300 ± 400 ns / iteration
BetterSafe	31400 ± 300 ns / iteration

Unsynchronized failed on every test, even getting stuck with all of its values at maximum in one out of the ten trials, while the others succeeded on every one.
Note that the measured times are much higher because the overhead is less amortized, as shown by the high cost per iteration for NullState, which should have no cost per iteration. These results are consequently less useful than the ones resulting from 1,000,000 iterations.

2 Analysis

2.1 Synchronized

The Synchronized implementation making use of the synchronized keyword was used as a baseline against which the other results were measured. While it was not the fastest implementation, Synchronized is intuitive and guarantees safety. There is no reason to use Synchronized in place of BetterSafe in any actual application, but if a concurrent program must be created quickly, it is perfectly reasonable to use the synchronized keyword on functions that alter state to guarantee that the program is data-race free.
The Synchronized implementation is data-race free because the synchronized keyword prevents other threads holding a reference to the object from executing methods that could interfere with the first executor.

2.2 Unsynchronized

As can be seen from the tests, the time saved by removing the synchronized keyword is generally not worth it. It was nearly impossible for the Unsynchronized implementation to produce a correct answer, and at the low numbers of threads and iterations required to get it to work, time saves mean little. Furthermore, as a result of corrupt state transitions, many more values in the array than expected went to the maximum value, greatly reducing the number of eligible state transitions and slowing the program down. Using a wholly unsynchronized concurrent program is highly inadvisable, but if little conflict between threads is expected (such as in an embarassingly parallel problem) or if the consequences of data races are not concerning for some reason, it may occasionally be done.

The Unsynchronized implementation is not data-race free because although all memory accesses in one thread are guaranteed to stay in order, it is completely possible for another thread to update a value in between the first thread reading and updating that same value. An example of a test that Unsynchronized is very likely to fail is

```
java UnsafeMemory Unsynchronized \
16 10000 100 20 20 20
```

2.3 GetNSet

Though an instructor on Piazza said this implentation should suffer from data races, and there is no obvious protection against them, I was unable to cause one. Even using System.out.print to perform I/O to force threads to yield between getting the old value and setting the new one did not result in any data races in more than twenty trials with the largest number of threads possible on lnxsrv06 and highest number of iterations possible in a practical amount of time (ten seconds or so). However, this failure to find data races does not mean that they are guaranteed not to exist, and in any event GetNSet was found to be slower than Synchronized across a large range of numbers of iterations. Due to its theoretical inability to guarantee data races will be avoided and its experimentally tested lack of performance, there is not much reason to use GetNSet for this application. However, the wide assortment of functionality provided by java.util.concurrent.atomic should not be discounted; in more complex objects where many separate variables are being manipulated concurrently and locking the entire object to protect only a few sections of memory would be wasteful, this library would be more efficient than the highly coarse-grained locking of the synchronized keyword and easier to use than the more granular locking of java.util.concurrent.locks.

In theory, GetNSet is not data-race free because although using volatile accesses guarantees that no reads or writes will be cached and thus accidentally shielded from other threads' reads or writes, it does not guarantee that all the reads and writes performed by one state change in one thread take place in a contiguous block. It should be possible for another thread to write to a value in between the first thread getting and setting its value, thus discarding the change created by the second thread. However, I was unable to find a test case that caused this to happen. As stated, I even tried making threads yield in hopes of increasing the delay between getting and setting a value to allow another thread to sneak in.

2.4 BetterSafe

There were four packages or classes suggested for implementing BetterSafe: java.util.concurrent, java.util.concurrent.atomic, java.util.concurrent.locks, and java.lang.invoke.VarHandle. I chose to use java.util.concurrent.locks for my implementation.

2.4.1 java.util.concurrent

Though java.util.concurrent offers a wide range of synchronization and concurrency options, none of them are easily usable for this application. The data being manipulated is simple enough that more complex concurrent data structures are unnecessary. Most of the five synchronizer classes deal with getting multiple threads to run simultaneously, but the opposite effect is desired; we want to exclude all other threads from affecting the part of data we are working with while we are working with it. The one exception, Semaphore, does not offer much that is useful to this application that locks from java.util.concurrent.locks do not offer.

2.4.2 java.util.concurrent.atomic

java.util.concurrent.atomic could be usable for this application. In particular, the atomic incrementAndGet and decrementAndGet operations offered by AtomicIntegerArray seem perfect. However, while this eliminates the possibility that the result of one increment or decrement will be overwritten and discarded by the result of another, there is still one small chance for failure in that java.util.concurrent.atomic offers no way to include the check that one value is greater than zero and the other value is less than the maximum value. This means that theoretically, two threads could check the same value, find that it is equal to one and thus can be decremented, then each decrement the value atomically without rechecking and discovering that this would put the value at -1. This could be solved by using compareAndSet to make sure that the value being incremented or decremented is still the same as the one that was found to be greater than zero or less than the maximum, but then there is an issue if one compareAndSet succeeds and the other fails, because then either the successful compareAndSet must be undone or the failed compareAndSet must be retried, checking again to ensure that the value is still eligible to be incremented or decremented. While this is not impossible, it is difficult, and so I opted not to use java.util.concurrent.atomic. Were it not for these issues, java.util.concurrent.atomic would likely be the best choice due to avoiding the overhead of mutexes.

2.4.3 java.util.concurrent.locks

java.util.concurrent.locks provides locks, which are designed to solve the problem of mutual exclusion we are dealing with. The trivial solution of using one lock for the whole object shows that this package can be used to achieve the same reliability as the synchronized keyword. However, locks are far from a magic bullet; the actual operations of acquiring and releasing a lock can be fairly expensive in comparison to using atomic operations, and attempting to use multiple locks can result in deadlock. Nevertheless, I chose to use java.util.concurrent.locks.

2.4.4 java.lang.invoke.VarHandle

java.lang.invoke.VarHandle offers similar functionality to java.util.concurrent.atomic; indeed, the memory effects of many functions in java.util.concurrent.atomic were defined in relation to those of functions in java.lang.invoke.VarHandle. As such, using VarHandle has similar pros and cons. Using atomic operations rather than locks to ensure safety has significantly lower overhead, but the complex testing and setting of two variables together stretches the limits of the provided atomic operations. I chose not to use java.lang.invoke.VarHandle for the same reasons I chose not to use java.util.concurrent.atomic.

2.4.5 Notes on my implementation

My implementation of BetterSafe is faster than Synchronized because it uses more fine-grained locking. Each value in the array has its own lock associated with it, rather than sharing one per-object lock. Thus, two swap operations that work on completely different values can execute at the same time, which cannot be done in the Synchronized implementation, which fully serializes all swaps. It is still completely safe and reliable because if any two operations conflict, they must be manipulating the same value, and as such they must each wait for the same lock before executing, so it is guaranteed that one operation will fully complete and release the lock before the other one begins. The biggest problem I had with this implementation was that of deadlock, because each swap operation requires two locks to ensure it can test and change two values. As a result, occasionally two threads would acquire one lock each and then attempt to acquire the one the other one was holding, preventing either thread from continuing and releasing the lock the other one was waiting for. I solved this by implementing a total ordering. By ensuring that the lock corresponding to the higher-indexed array element was obtained first, I removed the possibility of a circular dependency.
The other problem I overcame when attempting to implement BetterSafe was my failed attempt to use java.util.concurrent.atomic. This is why my BetterSafe class is called BetterSafe3State, which I assumed was acceptable because a Piazza post stated that any naming scheme was acceptable so long as the shell commands worked as specified. This means there are three unused classes in my jar: BetterSafeState and BetterSafe2State, which were previous attempts to use atomic operations to implement BetterSafe, and GetNSetUnsafeState, which was how I tried inducing data races in GetNSet by waiting between volatile accesses. The classes I implemented that should be graded are UnsynchronizedState, GetNSetState, and BetterSafe3State.