Homework 3 Report: Java Shared Memory Performance Races

Abstract

We measure and compare the performance and reliability of several models for modifying the values of an array using multiple threads. Alternative implementations using standard Java packages are also explored.

1. Performance and Reliability

	1	8	16
Null	20.90	149.81	490.33
Synchronized	50.53	1728.90	2838.38
Unsynchronized	40.83	DR	DR
GetNSet	56.90	DR	DR
BetterSafe	77.20	660.25	1386.72

Figure 1: Model vs. Thread Count: Table showing average ns/transition for a variety of models and thread counts. Each ns/transition value was averaged over 30 runs. Each run had 10,000,000 transitions. Unsynchronized and GetNSet have data races which cause infinite looping for 8 and 16 threads. DR (Data Race) has been substituted for those respective values.

Note: High variance with respect to the number of users on the SEASnet GNU/Linux servers has been noted; results may not be precisely reproducible, as testing circumstances may vary.

1.1. Characterization and Comparisons between Models

The implementation, performance, and reliability of each model varies substantially from the other models. We characterize and compare the performance and reliability of each model in the following paragraphs.

The Null model is a dummy class which does not actually implement transitions in a meaningful way. Although the transitions are not actually modifying any data, the measurements of the Null model may still be used to understand the overhead of the testing framework. Clearly, the ns/transition for each thread count of the Null model should be (and empirically is) substantially lower than any other; this is because no data mutation is actually occurring and no techniques to prevent contention are applied.

The Synchronized model uses the default Java keyword 'synchronized' to prevent contention between multiple threads. Unlike the Null model, the Synchronized model does implement transitions in a meaningful way. Because these transitions are performed by a synchronized method, the model is data race free and reliable. However, this comes at the expense of performance, as the Synchronized model can only average 2838 ns/transition for 16 threads.

The Unsynchronized model is similar to the Synchronized model, except that the method to implement transitions is not synchronized. This results in a higher performance, as threads are not concerned with acquiring or releasing locks. However, if there is more than 1 thread and a nontrivial number of transitions, the Unsynchronized model is extremely likely to loop infinitely. This behavior is caused when the data is incorrectly modified enough times to break the testing framework. Because of this, there is no measurement of 8 and 16 threads for the Unsynchronized model. For a single thread, the Unsynchronized model outperforms the Synchronized model, which makes sense, because it does not need to acquire/release any locks.

The GetNSet model relies on atomic get() and set() methods from the AtomicIntegerArray class. Although the get() and set() are atomic, meaning they are guaranteed to complete without interruption, the GetNSet model still has a data race. This is because another thread may modify the data between get() and set(). If this happens enough times, the testing framework breaks, similar to the Unsynchronized model. For a single thread, the GetNSet model performs worse than the Synchronized and Unsynchronized.

The BetterSafe model relies on locks from the ReentrantLock class. These locks are much faster than other locks because threads which do not acquire the lock will immediately return, rather than spin-locking. This model is data race free and outperforms the Synchronized model. For 16 threads, the BetterSafe model achieves 1386 ns/transition, which is even faster than the Synchronized model at 8 threads.

1.2. GDI's Application

The best model for GDI's application is BetterSafe because it has better performance than the Synchronized model and, as a bonus, has no data races, and so will be very reliable.

2. BetterSafe Implementation

2.1. Other Packages

The java.util.concurrent package is not necessary for the BetterSafe implementation because this package's top level synchronization techniques are specific to signaling, resettable multiway synchronization, control phased computation, or exchange points. These are not really relevant for the task at hand. The BetterSafe model simply needs to perform basic data modifications quickly with few or no data races.

The java.util.concurrent.atomic package can be used for the BetterSafe implementation because it supports the AtomicIntegerArray class. However, the getAndIncrement() and getAndDecrement() methods perform too slowly. That said, this approach does guarantee that there are no data races.

The java.lang.invoke.VarHandle package can also be used for the BetterSafe implementation. This is because it provides many atomic methods including getAndAdd(). Unfortunately, similar to the AtomicIntegerArray, this approach has been tested and was too slow compared to others. Many of these advanced atomic methods are not necessary for the current task, and therefore, this package was not used.

2.2. BetterSafe using ReentrantLock

The java.util.concurrent.lock package is the best package for implementing BetterSafe because of its performance and data race free nature. I compared this model against the other possibilities and empirically found that it was the fastest. This performance is explained by the fact that threads which fail to acquire the ReentrantLock will immediately return. On the otherhand, synchronized in Java relies on adaptive spinning, where a thread which fails to acquire the lock will spin before giving up and blocking. Moreover, this package is superior because of its simple syntax and methods – we only need to lock and unlock. The cons of this approach are that we still have to use a lock, which makes it slower than the unsynchronized; however, it has the benefit of working correctly. Moreover, the ReentrantLock has the advantage of potentially being more fine-grained than the synchronized keyword.

This BetterSafe implementation using the ReentrantLock from the java.util.concurrent.lock package is 100% reliable because threads acquire the lock for the entire duration of the critical section and release the lock only at the end of the critical section. This ensures that only one thread may access or modify the array at a time. No deadlock can occur because there is only one lock.

3. Measurement Techniques and Data Races

3.1. Measurement Techniques

As with most measurements, there is significant variance in the recorded ns/transition. This is caused by the varying activity levels on the SEASnet GNU/Linux servers. I have accounted for these inconsistencies as best as I can by taking the average of many repeated trials.

One source of error is that the startup costs and caching effects will vary depending on the other activities occurring on the machine at that particular instant. This cannot be entirely resolved in a reasonable and confirmable fashion, and thus I simply amortize the error by taking the average of many trials.

3.2. Data Races

The Unsynchronized model and GetNSet model both have data races. This is because neither implement synchronization. In particular, both the Unsynchronized model and GetNSet model just modify the values of the array as if there were only one thread. The following two shell commands are extremely likely to fail on the SEASnet GNU/Linux servers:

java UnsafeMemory Unsynchronized 16 10000000 2 0 0 0 0

java UnsafeMemory GetNSet 16 10000000 2 0 0 0 0