Homework 3: Java Shared Memory Performance Races

Introduction

Java provides many different ways to achieve synchronization in a multithreaded program. In this lab, we consider the performance and reliability of several synchronization methods including the synchronized keyword and ReentrantLock, as well as the tradeoff between performance and reliability when not using synchronization. Our multithreaded program maintains an array of bytes, and each iteration involves decreasing one of the numbers in the array and increasing another.

1. Test Conditions

All testing was done on the SEASNet Linux Server 6, running RedHat 7. The server is running two Intel Xeon Processor E5620, each of which supports 8 threads, for a total of 16 threads. The server also has approximately 64 GB of memory and runs Java version 10.0.1, with Java SE Runtime Environment 18.3 (build 10.0.1+10).

2. Synchronization Methods

The synchronization methods we tested for performance and reliability were the "synchronized" keyword and the ReentrantLock from java.util.concurrent.locks. These are the only true synchronization methods we tested; both eliminating the synchronized keyword entirely and declaring each byte in the array to be volatile do not provide reliability. We consider the benefits and disadvantages of the ReentrantLock and alternate methods to achieve synchronization and justify our decision to use ReentrantLock to implement BetterSafe.

2.1. java.util.concurrent

This package provides low level structures that can help achieve concurrency, such as Semaphore, queues, and ThreadFactory[1]. Although this package could be used to implement BetterSafe, its low-level nature makes it more difficult to use, compared to the simplicity of just locking and unlocking the ReentrantLock. However, using a synchronized queue to schedule threads gives a lot of freedom to the programmer to achieve the exact behavior that he wants, and would likely run much faster than the ReentrantLock.

2.2. java.util.concurrent.atomic

This package provides classes that update their values atomically, providing a simple lock-free way of ensuring that values are properly updated[1]. Using the classes provided in this package to implement BetterSafe would be faster than a lock-based approach, since there is less overhead associated with an atomic update than there is with locking or unlocking. However, because our program requires both reading and writing to the array, the lack of locks can cause problems. Since the program checks that the values in the array are valid and then increments them, if execution changes to another thread after the check is done but before the increment, then the program will potentially increment a value past its maximum value. Because we want to avoid such reliability errors, we would not choose these classes to implement BetterSafe.

2.3. java.util.concurrent.locks

This package provides many different types of locks that can be used to ensure mutual exclusion in critical sections. In particular, the main implementation of the Lock interface is the ReentrantLock[1], which we use in BetterSafe to ensure mutual exclusion. Locks are simple to use, only requiring a lock operation before the critical section and an unlock operation after the critical section. However, because locks force mutual exclusion, this reliability comes at the cost of performance, as if one thread has the lock, no other threads can update the array.

2.4. java.lang.invoke.VarHandle

This class provides a reference to a particular variable, and grants access to the variable through different access modes. The variable can be changed atomically to a new value, similar to the atomic package[1]. Using this class associated with the byte array would be faster than a lock-based approach, since there is less overhead associated with atomic updates than with locks. However, just as in the case of the atomic package, our program requires both reads and writes, meaning that execution switches can cause the program to fail even when atomic updates are performed. Because we want to avoid these reliability errors, we should not use this class to implement BetterSafe. ReentrantLock is the best way to implement BetterSafe, as it combines ease of use with the mutual exclusion that we need.

3. Test Results

3.1. Performance Tests

The performance of Null, Synchronized, Unsynchronized, GetNSet, and BetterSafe was measured in terms of average time per iteration in nanoseconds. Null is included to help measure the overhead associated with the function calls, thread creation, and loops; because it instantly returns from functions, almost all of its time is spent in one of these three areas. Null is able to consistently outperform Synchronized by a significant margin

regardless of the thread count, number of iterations, initial sum, and size of array. We therefore want another synchronization method that is able to provide the reliability of Synchronized but run faster.

For each of these measurements, we chose to include the data point even if the final sum did not match the initial sum, as we are looking for a metric on the performance, not the reliability. However, we did not include the data point if the program ended up looping, which can occur if data races cause all of the values to increase to the maximum value or decrease to the minimum value, making it impossible to perform another valid swap. Since the program will never finish if this occurs, the average time will be infinite, which is not helpful for our analysis.

The test program was run on these five classes, measuring the performance as a function of the number of threads. The average time per iteration in nanoseconds was measured five times for each class and thread count and averaged to reduce variance in the final measurement. The number of iterations for these tests was 10,000, the maximum value was 100, and the starting array was [50, 50, 50, 50, 50]. The results are seen in Figure 1 below.

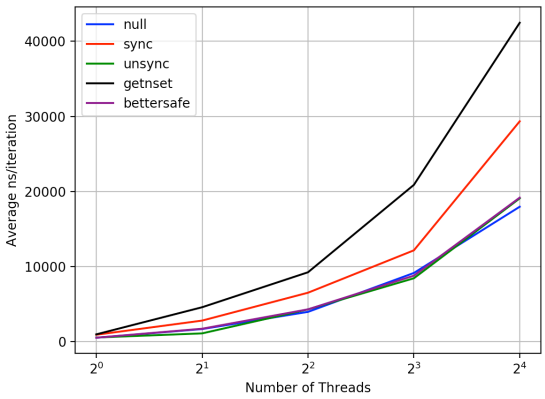

Figure 1: Average time per iteration as a function of the number of threads. In each of the five classes, the average time increases roughly linearly with the thread count.

We also measured the performance as a function of the number of iterations. We obtained the average time per iteration five times for each class and number of iterations, and averaged the times to reduce the variance in the final measurement. The thread count for each of these tests was 8, the maximum value was 100, and the starting array was [50, 50, 50, 50, 50]. As the number of iterations increases, the number of infinite loops in the unreliable classes also increases. Consequently, we were unable to get good data for the GetNSet class when the number of iterations reached 10,000. The results of this test are shown in Figure 2.

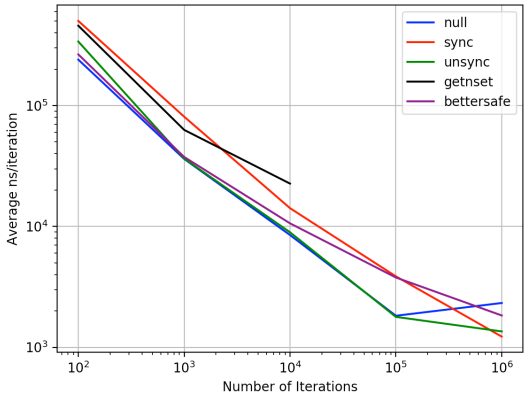

Figure 2: Average time per iteration as a function of the number of iterations. As the number of iterations increases, the average time per iteration decreases, since the overhead due to synchronization/thread creation becomes less significant the longer the program runs.

3.2. Reliability Tests

We define a test as successful if the final sum of the array elements is the same as the initial sum, and define reliability as the proportion of successful tests. This condition does not imply that synchronization was maintained the entire time; one error could have simply corrected an earlier error. The Synchronized, BetterSafe, and Null classes all maintain 100% reliability regardless of the number of threads, iterations, initial sum, or array size. Synchronized and BetterSafe use mutual exclusion to achieve this reliability, while Null simply returns immediately and therefore cannot fail. We tested the reliability of GetNSet and Unsynchronized as a function of the number of iterations. Because these two classes do not use locks or other synchronization methods, we see many infinite loops as the number of iterations increases. In addition, reliability for these classes is low, with either zero or one successful tests in each case. The thread count for each of these tests was 8, the maximum value was 100, and the starting array was [50, 50, 50, 50, 50]. A total of 30 tests were run for each number of iterations. Tables 1 and 2 show the reliability of Unsynchronized and GetNSet.

Iterations	Success	Mismatch	Loop	Reliability
1,000	1	29	0	1/30
10,000	0	27	3	0/30
100,000	0	9	21	0/30

Table 1: Reliability of Unsynchronized

Iterations	Success	Mismatch	Loop	Reliability
1,000	0	30	0	0/30
10,000	1	27	2	1/30
100,000	0	2	28	0/30

Table 2: Reliability of GetNSet

4. Analysis

4.1. Synchronized

Synchronized is a data-race free class, since every update to the array is done through a synchronized method. Consequently, two threads cannot simultaneously call the same method, so two updates to the array cannot occur concurrently. However, although Synchronized is 100% reliable, we see in Figures 1 and 2 that it is consistently the slowest.

4.2. Unsynchronized

Unsynchronized is not a data-race free class, and makes no effort to ensure that two updates to the array do not simultaneously occur. Consequently, two updates can overlap with each other, causing a value to be incremented just once. The following command is extremely likely to fail, and in fact fails all 30/30 tests:

java UnsafeMemory Unsynchronized 8 100000 4 2 2

Although Unsynchronized is very fast, running almost at the same speed as Null, this performance increase comes at the cost of reliability. We see in Table 1 that Unsynchronized fails 89/90 test cases. Unsynchronized's unreliability is too big of a factor to justify its use for any practical purpose, even though it runs extremely fast.

4.3. GetNSet

GetNSet is also not a data-race free class, although it at least attempts to achieve synchronization by using the AtomicIntegerArray class to make each read and write to the array atomic. Though this helps avoid data races in theory, we see in practice that GetNSet is just as unreliable as Unsynchronized. Although volatile forces a memory access each time a variable is read or written, if execution switches to another thread right after the read is performed and a write is done in the other thread, the value in the read will be outdated, so the test fails. Note that this unreliability is partially due to the fact that we were not allowed to use the methods getAndIncrement() and getAndDecrement(), relying only on get() and set() instead. The same command also fails for GetNSet in 30/30 test cases.

java UnsafeMemory GetNSet 8 100000 4 2 2

GetNSet is not very fast, performing the worst of the five classes in most cases. Because of all of the memory accesses that GetNSet is forced to do (up to 6 in one iteration), as the number of iterations increases, we expect to see GetNSet's performance decrease relative to the other classes. As shown in Figure 2, as the number of iterations rises, GetNSet starts to perform worse than the other classes. In addition, since there are so many memory accesses, as the number of threads increases, GetNSet's performance will also decrease.

4.4. BetterSafe

BetterSafe is a data-race free class, using the ReeentrantLock class to ensure that the array cannot be accessed by two different threads. Before the contents of the array are examined, BetterSafe locks the ReentrantLock, and unlocks it after the values of the two bytes has been changed. By making it impossible for two threads to access the array simultaneously, BetterSafe becomes data-race free.

BetterSafe is also able to run faster that Synchronized while still maintaining 100% reliability. We see from Figures 1 and 2 that BetterSafe is faster than Synchronized in almost all cases. BetterSafe secures the critical section using a finer-grained lock, locking only the sections that read or write from the array, instead of locking the entire method like Synchronized does[2]. However, BetterSafe is consistently slower than GetNSet and Unsynchronized. GetNSet uses atomic integers, which requires less overhead than locks but does not ensure reliability, while Unsynchronized runs faster since it does not make any effort to ensure reliability.

5. Conclusion

In this lab we measured the performance and reliability of various synchronization methods. We see that the synchronization keyword is able to provide 100% reliability, but at the cost of performance. There exist faster ways to achieve 100% reliability, such as using ReentrantLock. Finally, we see that certain methods such as atomic arrays and not attempting synchronization trade performance for reliability; while GetNSet and Unsynchronized both run almost as fast as Null, they provide 0% reliabilitySince GDI wants a class that is fast and reliable, the most suitable class is BetterSafe.

6. References

1. Java Standard Edition Version 10 API Specification. Oracle and/or its affiliates. Available: https://docs.oracle.com/javase/10/docs/api/overview-summary.html
2. Lea, Doug. Using JDK 9 Memory Order Modes. Available: gee.cs.oswego.edu/dl/html/j9mm.html