Computer Science 131, Programming Languages

Homework 3, Java Shared Memory Performance Races

Name: Zhouyang Xue

ID: 104629708

TA: Saketh Kasibatla

Server: lnxsrv09.seas.ucla.edu

Java Version: 1.8.0_161

CPU: 2.3 GHz Intel Core i5

Memory: 8 GB 2133 MHz LPDDR3

Date: 02-09-2018

1 BetterSafe Implementation

1.1 *java.util.concurrent*

java.util.concurrent is a package used for developing concurrent applications. One functionality this package includes that is useful for Homework 3 is semaphores. The semaphore blocks thread-level access to critical sections to achieve mutual exclusion. Normally, a semaphore holds a set of permits, and before a thread could access the critical section, it has to check for available permits in the semaphore. This approach would definitely increase the reliability of the program, yet in terms of performance, it is approximately the same as the keyword Synchronized, if not slower. Based on the spec, we want *BetterSafe* to have better performance than *Synchronizedstate*. Therefore, there are apparently better choices.

1.2 *java.util.concurrent.atomic*

This package supports lock-free, thread-free programming, and is very powerful in terms of concurrency. However, given that we would expect multiple threads running at the same time in this homework, and that we want *BetterSafe* class to be 100% reliable *java.util.concurrent.atomic* is not a good choice, either.

1.3 *java.lang.invoke.VarHandle*

This package allows users to use Java constructs to atomic or ordered operations on fields or individual classes. In terms of reliability, *java.lang.invoke.VarHandle* appears to have similar problems with *java.util.concurrent.atomic*. That is, when it uses the volatile keyword to access shared variables when multiple threads are running, it still couldn't avoid race condition.

1.4 *java.util.concurrent.locks*

Finally, *java.util.concurrent.locks* is my final choice. I chose this package for two major reasons. The first one is that it's reliability-proof, because it utilizes ReentrantLock to achieve mutual exclusion. The second reason is that it has a way superior performance than Synchronized methods. The *java.util.concurrent.locks* package was designed for performance optimization under multi-thread environment with JDK 5.0. Therefore, I landed on using *java.util.concurrent.locks* to implement *BetterSafe*.

2 BatterSafe Performance

2.1 Performance

As discussed above, the ReentratLock was designed for performance optimization under multi-thread environment with JDK 5.0. For example, ReentrantLock supports lock polling and interruptible lock waits, and it's also unstructured. These functionalities largely improve *BetterSafe* class' performance, and allows it to have a better performance than *Synchronized.*

2.2 Reliability

Essentially, BatterSafe is using locks to prevent race condition, which is to say, when one thread is operating on a critical section, other threads would not have the permission to access the same section. Therefore, BetterSafe class is guaranteed to be reliable.

3 Challenges

The logic of the implementing the four classes are fairly easy, yet testing them turns out to be difficult.
The hardest part of testing is that for all the unreliable classes (e.g. *GetNRun* and *Unsynchronized*), it is possible to get into infinite loop when run over multiple threads. This feature forbids me to run shell scripts for testing.

Moreover, it seems like these infinite loops also took over some portion of the CPU resources, that makes other test results slower.
DRFs discussed below.

4 Performance and Reliability

4.1 Unsynchronized

This class is the fastest yet the most unreliable class among the four. During tests of 10000 transactions, the average time/transaction is 1997.5ns on 4 threads, and 3294.46ns on 8 threads.
Unsynchronized's unreliability actually makes sense, because it holds no protection over critical sections, and that all threads can access within any order.
This class is not DRF, the reliability shows almost 0% when the number of transactions is extremely large. A good example would be:
java UnsafeMemory Unynchronized 12 1000000 6 5 6 3 0 3

4.2 Synchronized

When the transaction number is small, Synchronized's performance outcasts BetterSafe's. Its average time/transaction of 10,000 transactions is 8038.72ns over 8 threads, and 3131.99 over 4 threads. However, for 1,000,000 transactions, its average time/transaction becomes 1643.61ns over 12 threads, 911.907ns over 8 threads, and 384.115ns over 4 threads.
In terms of reliability, Synchronized is very reliable indeed. The synchronized keyword assures that no two threads would access swap() at the same time.

4.3 GetNSet

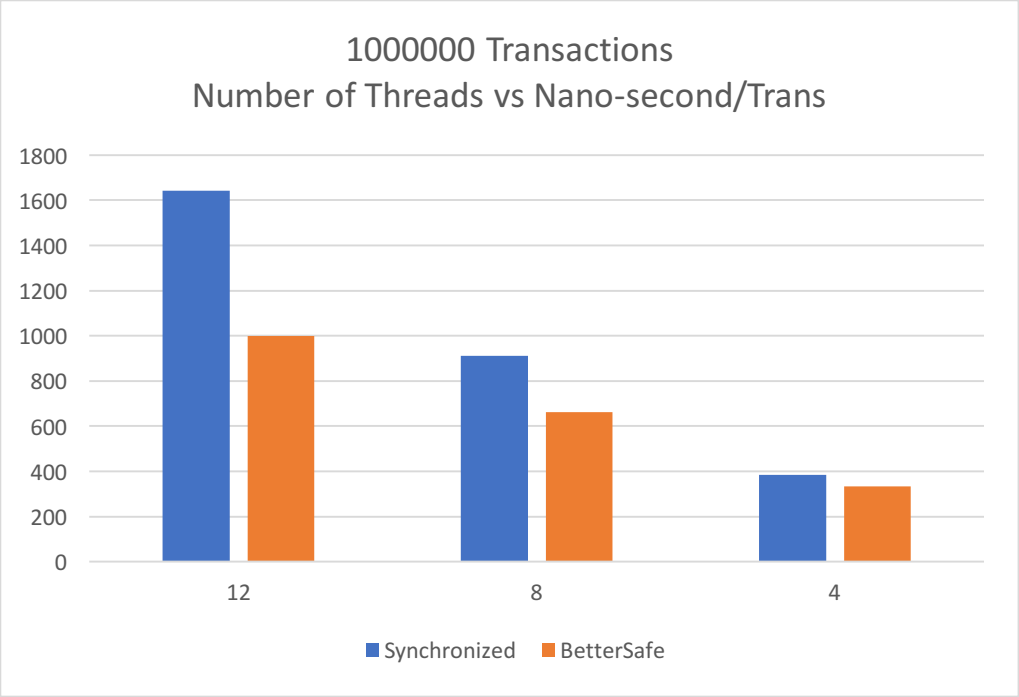
The GetNSet approach is between Unsynchronized and Synchronized. It outperforms Synchronized in terms of efficiency, but not as reliable. In fact, it's reliability is only slightly better than Unsynchronized. The reason is that get() and set() make sure that no two methods would access one variable through get() or set() at the same time. However, the typical write-load sequence problem still exits. In other words, get() and set() are essentially two separate operations and while one thread is accessing a variable through set(), the variable can still be get() by other threads. Therefore, this approach is as not reliable as unsynchronized, maybe a little bit better.
This class, similar to Unsynchronized, is also not a DRF class. The infinite loop result occurs very often on GetNSet, and its reliability is also near 0% when it comes to large number of transactions: e.g.
java UnsafeMemory GetNSet 12 1000000 6 5 6 3 0 3

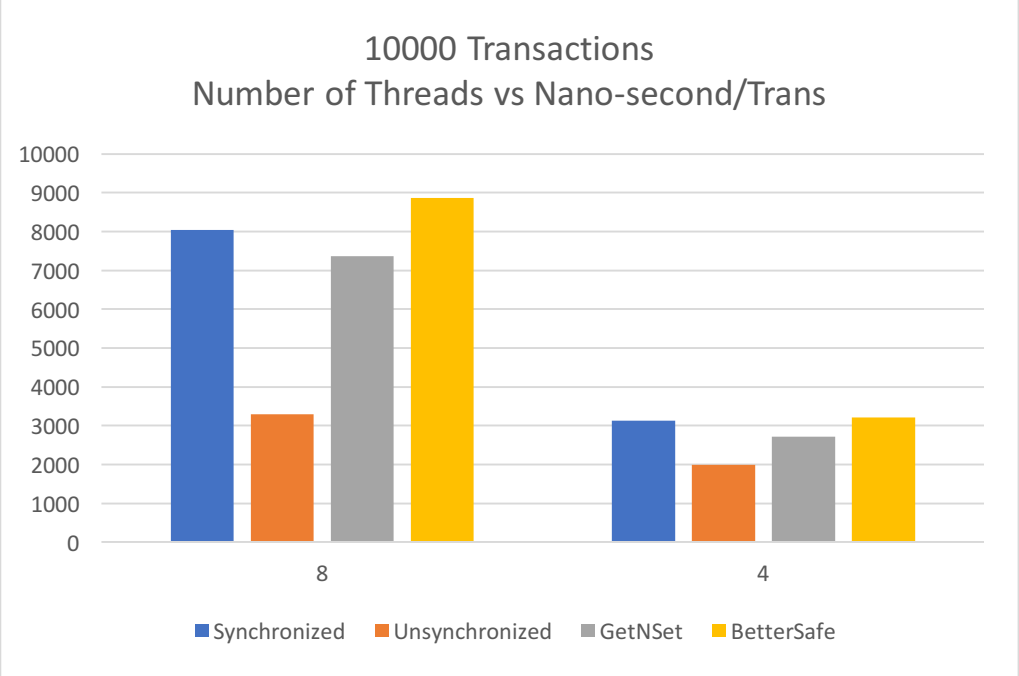
4.4 BetterSafe

When it comes to large number of transactions, BetterSafe is apparently the winner among the four. As discussed above, first and foremost, it's 100% reliable. In terms of efficiency, its time/transaction of 1,000,000 transactions is 999.621ns over 12 threads, 661.739 over 8 threads, and 332.258 over 4 threads. However, then there are only 10,000 transactions, its performance lost to Synchronized. BetterSafe utilizes ReentratLock, thus is perfectly reliable.

5 Conclusion

For GDI's application, the best choice would be BetterSafe, given that GDI specializes in finding patterns in large amount of data. Therefore, BetterSafe is an efficient and also reliable class in this case.


1000000 Transactions
Number of Threads vs Nano-second/Trans
1800
1600
1400
1200
1000
800
600
400
200
0
12
8
4
Synchronized
BetterSafe


10000 Transactions
Number of Threads vs Nano-second/Trans
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
8
4
Synchronized
Unsynchronized
GetNSet
BetterSafe