Computer Science 131 Programming Languages
Homework 3 Report

Name:	Jerry Liu
ID:	404474229
Lab Section:	1B
TA:	Seunghyun Yoo

Server:	`lnxsrv09`
CPU:	Intel®Xeon® CPU E5-2640 v2
Frequency:	2.00GHz
Cores / Threads:	16 Cores / 32 Threads
Main Memory:	64,216,540 kB
Java Version:	1.8.0_112

Date: October 24, 2016

1 Better Safe

1.1 Performance

BetterSafe is faster than *Synchronized* since it utilizes `ReentrantLock` in the `java.util.concurrent` package. The `java.util.concurrent` package was introduced in JDK 5.0 to optimize performance in multithread environment as specified in the Java API [1]. In other words, *ReentrantLock* was designed to give superior performance than *synchronized* keyword. This is why our *BetterSafe* implementation is faster than *Synchronized*.

1.2 Reliability

Since in our implementation, we lock the whole function body as a critical section and we won't unlock it until we return, our implementation serves the same thread-safety purpose as the `synchronized` keyword. Thus, our implementation for *BetterSafe* is as reliable as *Synchronized*.

2 BetterSorry

2.1 Performance

As we can see from Appendix A, when the thread number is 8, *BetterSorry* beat *BetterSafe* in 3/4 tests.

BetterSorry is faster when the *number of transitions*, N, is not very large because *BetterSorry* utilizes `AtomicIntegerArray` and its builtin atomic `getAndIncrement()`, `getAndDecrement()` methods.

According to the Java API [1],
`java.util.concurrent.atomic` is both lock-free and thread-safe on single variables. Since this method does not impose any lock, it is expected to run faster than *BetterSafe* when N is not very large.

However, when N gets large ($\approx 10^6$), *BetterSafe* performs better than *BetterSorry*, which might be caused by data-race conditions on *BetterSorry*. Data-race conditions will slow down the program significantly if we take a look at the `SwapTest` implementation. I will discuss this in detail in Appendix B.

2.2 Reliability

BetterSorry is much more reliable than *Unsynchronized* since *Unsynchronized* imposes a vast number of race conditions that will never get the result right. Our test data also show the same trait. When *number of transitions* is large, *Unsynchronized* cannot even finish its calculation due to the implementation of `SwapTest`. Here is a table comparing the two classes' reliability when $N = 10^4$:

Class	BS	Unsync
Wrong Ans	0	14
Infinite Loop	0	15
% of Wrong Ans	0%	93.33%

2.3 Race Condition

Here is the source code of the `swap` function:

```
public boolean swap(int i, int j)
{
        if (value.get(i) <= 0 ||
            value.get(j) >= maxval)
                return false;

        value.getAndDecrement(i);
        value.getAndIncrement(j);
        return true;
}
```

As we can see, between the if statement and `value.getAndDecrement(i); ...`, other threads might just step in and modify the i^{th} and j^{th} position before the current thread try to execute `value.getAndDecrement(i); ...`. This is where the race condition comes from. If we increase the number of threads and / or increase the number of transitions required, race conditions will happen more frequently. In fact, as we can see from Appendix A, *BetterSorry* is already on its way to surpass the time required by *Synchronized*.

However, it is very hard to make *BetterSorry* fail due to the system's configuration. The CPU is very powerful, and the time between the `if` statement and the update statements is too small for another thread to update the same positions at the same time. Also, the CPU also has advanced branch prediction rule. Since most of the evaluations are true, the CPU won't even need to execute the conditions in the `if` statement at all. This implies that only update statement, an atomic operation, is executed most of the time. When the CPU mispredicts a branch, all later executions will be stopped and the CPU has to restore what actually happened at the point of misprediction. This makes other threads hard to step on the integer array.

If we really want to make this function fail, we probably need much more threads and number of transitions to achieve the result. We can also make the number of transitions really really small and fit everything into each Core's cache. This can occasionally fail the test. In other words, it is very difficult to create such scenario in SEASNet's servers.

3 Difficulties

I have difficulties figuring out why *GetNSet* does not work when we increase the thread numbers and / or the number of transitions.
Also, I had some troubles writing the test scrip to automate the testing process. I originally tried to use a for loop to iterate through different threads, number of transitions, but given the unreliability of *Unsynchronized* and *GetNSet*, I give up and start to separate these two classes with others in testing.
Last but not least, the graph plotting is tedious.

3.1 Unsynchronized

This one is not DRF since it has no synchronization at all. Test case is very simple:

```
java UnsafeMemory Unsynchronized \\
8 100000 6 6 3 0 3
```

This one gives us $25/30 = 83.33\%$ infinite loops and $5/5 = 100\%$ wrong answers. Note: for all tests I try to run 30 times to satisfy *Central Limit Theorem*.

3.2 GetNSet

This class is not DRF. We use the same reasoning from the last section: between the evaluation of the `if` statement and the update clause, other threads might step on the same exact position as the current thread. Since we were first calculating the values of the parameter to the `set()` method, other threads have a high probability of using that piece of storage and thus make the program unreliable. The same test case that fails *Unsynchronized* can also fail this one:

```
java UnsafeMemory GetNSet \\
8 100000 6 6 3 0 3
```

This one gives us $28/30 = 93.33\%$ infinite loops and $2/2 = 100\%$ wrong answers.

3.3 BetterSafe

This class is DRF since we used `ReentrantLock` to implement it in the critical section: inside the `swap` function.

3.4 BetterSorry

This class is theoretically not DRF but it is very hard to make it fail just because of race conditions. This one is actually very reliable. The following test case can make *BetterSorry* to fail 1 of 30 instances:

```
java UnsafeMemory GetNSet \\
32 100 6 6 3 0 3
output too large (7 != 6)
```

4 Conclusion

After performance and reliability tests, we have the following conclusion:

1. If the number of transitions and / or the number of threads is not particularly large, we should use *BetterSorry* since it gives the best performance with substantial, though not 100%, reliability. Although when the number of transitions is very small, the *BetterSorry* method gives the wrong result, but the result is not far away from the correct one. Of course, when the transition is very small, we do not need any multithread application anyway.

2. If the number of transitions and / or the number of threads is very large, say $> 10^6, > 16$ respectively, we should really consider *BetterSafe* because it is not only safer but also faster in such scenarios.

A Performance Measurement

For performance measurement, we need to control the number of variables. I first controlled the number of transitions and then the number of threads.

A.1 Time vs Number of Transitions

Here I utilize 8 threads, the same number as most modern high-end PCs. I vary the number of transitions from 10^3 to 10^6. Figure 1 and Figure 2 show the result:

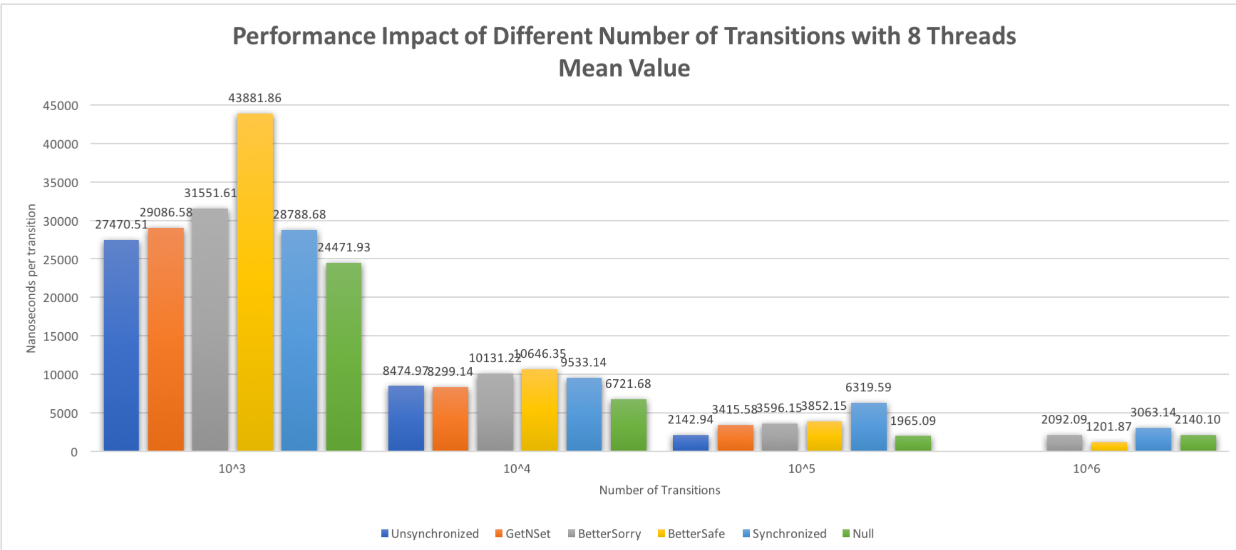

Figure 1: Plot of Mean Values of Performance Impact of Different Number of Transitions with 8 Threads

As we can see, *Null* almost always run ahead of other classes since it does nothing at all.
Synchronized is getting slower and slower relatively when the number of transitions increase. This is why the Java people invented the *concurrent* package to fix the performance issue caused by `synchronized` keyword.
Unsynchronized might be the fastest besides *Synchronized* due to its lightweight (no locks, no concurrent package uses). However, it is not usable due to the number of times it caused *SwapTest* to run into infinite loops and the high percentage of getting a wrong answer.
Table 1 shows the number of wrong answers and infinite loops out of 30 runs:

Number of Transitions	10^3	10^4	10^5
Infinite Loops	0	15	25
% of Wrong Answers	90%	93.33%	100%

Table 1: Number of Transitions vs Number of Infinte Loops & % of Wrong Answers

GetNSet has better performance compared to safer classes (defined in the spec), but it also has terrible reliability. It is almost as unreliable as *Unsynchronized*.
Table 2 shows the number of wrong answers and infinite loops out of 30 runs:

Number of Transitions	10^3	10^4	10^5
Infinite Loops	1	5	28
% of Wrong Answers	100%	100%	100%

Table 2: Number of Transitions vs Number of Infinte Loops & % of Wrong Answers

BetterSafe uses the `ReentrantLock` from the concurrent package. As we can see, as the number of transitions increase, *BetterSafe* gets faster and faster compared to other classes. When the number of transitions is 10^6, it actually outperforms *Null*. This performance test proves why `ReentrantLock` was developed to replace synchronized.

BetterSorry uses `AtomicIntegerArray` and its builtin `getAndIncrement(); getAndDecrement()` atomic methods to update values. It is very reliable and fast. It is faster than *BetterSafe* most of the time except when $N = 10^6$, the reason of which is discussed in Section 2.3 and Appendix A.2.

We also have a plot of standard deviation to get a sense of the variability of the average time required to finish all transitions.

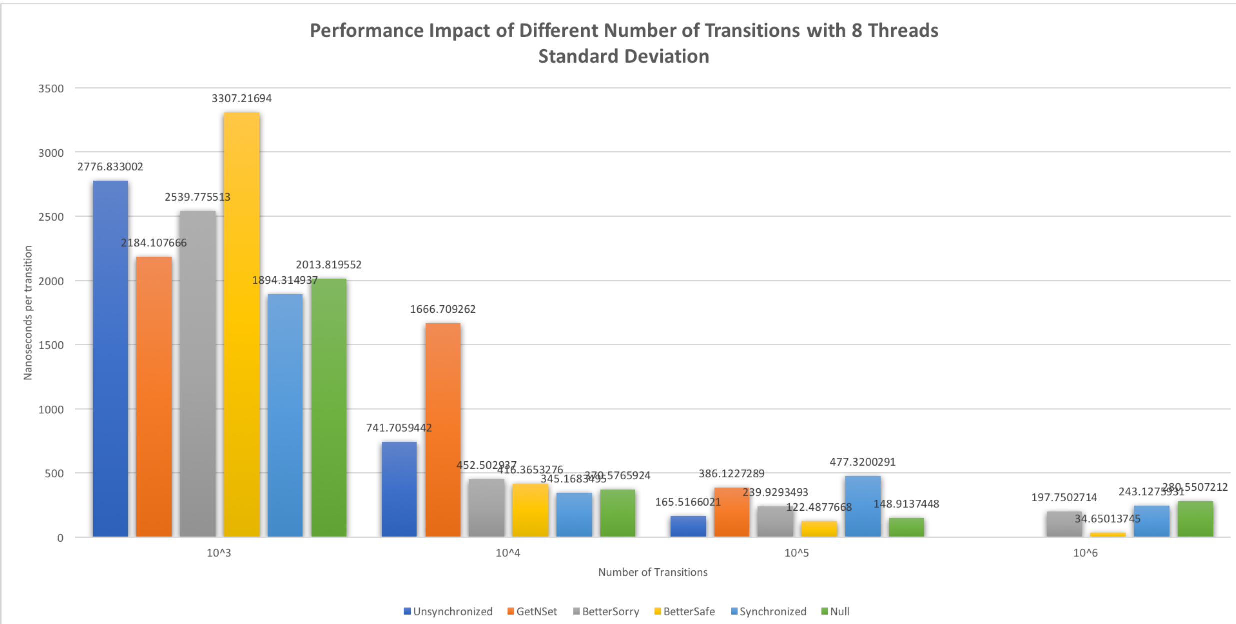

Figure 2: Plot of Standard Deviation of Performance Impact of Different Number of Transitions with 8 Threads

A.2 Time vs Number of Threads

We chose 10^6 transitions and tested on all safer classes. Figure 3 and Figure 4 show the result:

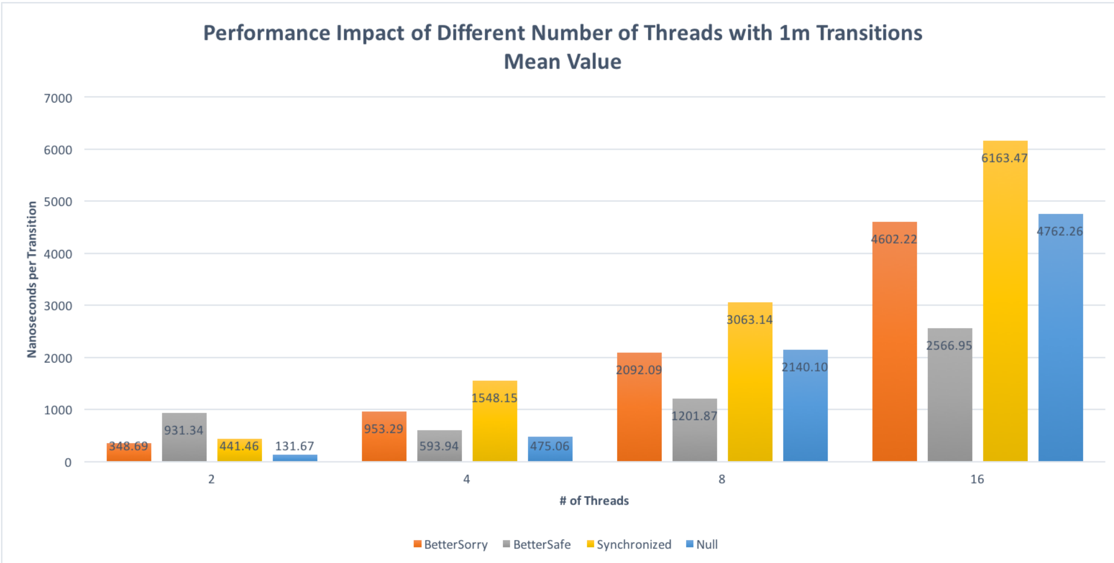

Figure 3: Plot of Mean Values of Performance Impact of Different Number of Threads with 10^6 Transitions

As we can see, *Synchronized* is the slowest method when the number of threads is larger than 2. For a normal server, it is almost impossible to only have 2 threads. as the number of threads increases, *BetterSorry* actually performs worse than *BetterSafe* due to potential data races. This proves how incredible the `ReentrantLock` is. It is not only safer, but also faster when the number of threads & the number of transitions go up.
The variability is shown in Figure 4. This also gives us idea about reliability of each method. If potential data races exist, the variance will go up due to inconsistent time spent choosing two *eligible* swap positions.

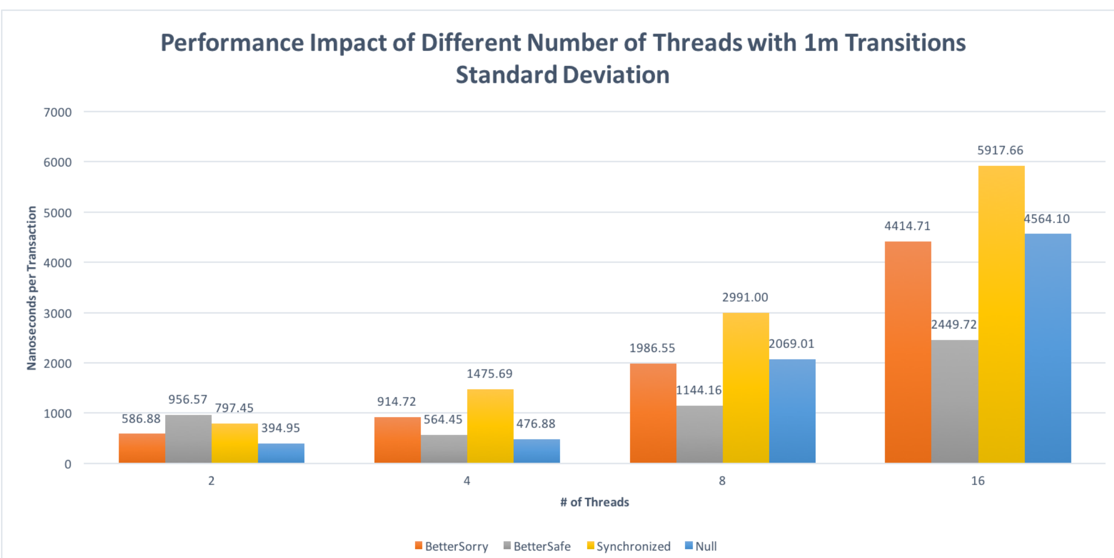

Figure 4: Plot of Standard Deviation of Performance Impact of Different Number of Threads with 10^6 Transitions

B SwapTest details

The reason why we have infinite loop can be shown below:

```
public void run() {
    int n = state.size();
    if (n != 0)
        for (int i = 0; i < nTransitions; ) {
            int a = ThreadLocalRandom.current().nextInt(0, n);
            int b = ThreadLocalRandom.current().nextInt(0, n - 1);
            if (a == b)
                b = n - 1;
            if (state.swap(a, b))
                i++;
        }
}
```

We can see that if the swap is not successful, the for loop will not increment the variable `i`. Thus, if we have data race conditions, and the threads that increment a position updates its value before the threads decrementing that position, or vice versa, we will see an array full of 0's and `maxval`'s. Finally, the swap will always fail, and the program stuck in the infinite loop forever.

References

[1] `http://docs.oracle.com/javase/8/docs/api/`, *Java™Platform, Standard Edition 8 API Specification*, Oracle and/or its affiliates, 2016