CS131 HW3 Java Memory Performance Races
Jun Zhou 804179311

BetterSafe:

BetterSafe is better than Synchronized, because it uses a reentrant lock, which ensures correct performance (like synchronized), but the implementation of lock in Java was designed to be an improvement on simply using the 'synchronized' keyword. Synchronized works by locking the whole object when a synchronized member function is called on it. Reentrant lock is similar but has added functionality, and is supposed to be more scalable and perform better under 'heavy contention,' as explained in http://www.ibm.com/developerworks/java/library/j-jtp10264/index.html.

BetterSorry:

My BetterSorry implementation is similar to GetNSet, since it also uses atomic access. It is at least as good as GetNSet because of atomicity, but it is also better, simply by using getAndDecrement() and getAndIncrement(), which makes the entire retrieval and update action atomic, rather than performing a a volatile fetch and then an update separately. This atomicity ensures better reliability than Unsynchronized, which provides no protection at all. Volatile access is helpful because it ensures that a thread will see updates performed by other threads immediately and will fetch the newest data, rather than simply using and updating its own internal cache. Unsynchronized will be vulnerable to threads reading and writing out old values. The threads will have an inconsistent view of the data. In some test cases, BetterSorry also achieves better performance/speed than BetterSafe, because it does not require the locking overhead that BetterSafe needs. Threads do not need to wait for each other. For example, with 20 threads and 1000 swaps, or with 30 threads, BetterSorry is faster. See test results summary.

BetterSorry does not ensure perfect correctness and is still vulnerable to race conditions. It does not use locking, so it is still the case that two threads could read in a value at almost the same time, perform updates on that value, and both try to write out to memory. Whichever update is performed last will be the final result, but not both. For example, if the threads were trying to increment, only one incrementation would be performed rather than both.

GetNSet:
is designed to be halfway between synchronized and unsynchronized. It does not lock, so it does not ensure 100% correctness. But it does use volatile access with get() and set(), which will access memory each time rather than loading from cache. The result may still be wrong however, because while one thread had fetched a value and is updating it, another thread could try to get that same value. After the first thread writes its update back out, the second thread would finish its own update (but an update on the previous value), and it's write would overwrite thread one's update. Ie, both threads try to increment an array element, but the end result is +1 rather than +2 as expected.

Measurements and Problems:

Some classes which had data races were hard to measure, because they could end up in an infinite loop. The swap function checks bounds between 0 and maxval, so if the bound is breached, the function would return false and the thread would not be able to increment its counter and continue working.

Only Synchronized and BetterSafe are fully DRF. GetNSet and BetterSorry are both compromises. Unsynchronized is completely unsafe.

These are some tests that are likely to fail:
```
java UnsafeMemory Unsynchronized 8 1000000 6
5 6 3 0 3
java UnsafeMemory GetNSet 8 1000000 6 5 6 3
0 3
```

Conclusion:
Based on the test results, the most unreliable (and slow due to deadlocks) are Unsynchronized and GetNSet. I would say those should not be used. I would recommend to use either BetterSafe or BetterSorry. BetterSafe has a better lock implementation than Synchronized, which gives it perfect reliability and better speed. BetterSorry may still have data races because it does not perform locking. However, it is a compromise using volatile access, and with a higher number of threads, it can sometimes perform faster than BetterSafe because threads do not need to wait for each other.

Test Results:
```
java UnsafeMemory Synchronized 8 1000000 6 5
6 3 0 3
Threads average 4608.40 ns/transition
java UnsafeMemory Synchronized 8 1000000 6 5
6 3 0 3
Threads average 3985.77 ns/transition
java UnsafeMemory Synchronized 8 1000000 6 5
6 3 0 3
```

```
Threads average 4300.38 ns/transition

 java UnsafeMemory Synchronized 20 1000000 6
5 6 3 0 3
Threads average 10068.7 ns/transition
java UnsafeMemory Synchronized 20 1000000 6
5 6 3 0 3
Threads average 9444.64 ns/transition
java UnsafeMemory Synchronized 20 1000000 6
5 6 3 0 3
Threads average 6621.53 ns/transition

java UnsafeMemory Synchronized 20 1000 6 5 6
3 0 3
Threads average 162016 ns/transition

............................................
java UnsafeMemory Unsynchronized 8 1000000 6
5 6 3 0 3
//deadlock, does not finish

java UnsafeMemory Unsynchronized 20 1000 6 5
6 3 0 3
Threads average 179240 ns/transition
sum mismatch (17 != 21)
java UnsafeMemory Unsynchronized 20 1000 6 5
6 3 0 3
Threads average 167979 ns/transition
sum mismatch (17 != 18)
java UnsafeMemory Unsynchronized 20 1000 6 5
6 3 0 3
Threads average 170903 ns/transition
sum mismatch (17 != 25)

............................................
java UnsafeMemory GetNSet 8 1000000 6 5 6 3
0 3
//infinite loop, doesn't finish

java UnsafeMemory GetNSet 20 1000 6 5 6 3 0
3
Threads average 212380 ns/transition
sum mismatch (17 != 12)
 java UnsafeMemory GetNSet 20 1000 6 5 6 3 0
3
Threads average 195957 ns/transition
sum mismatch (17 != 18)
java UnsafeMemory GetNSet 20 1000 6 5 6 3 0
3
Threads average 187647 ns/transition
sum mismatch (17 != 21)
java UnsafeMemory GetNSet 20 1000 6 5 6 3 0
3
Threads average 207728 ns/transition
sum mismatch (17 != 20)

............................................
java UnsafeMemory BetterSafe 8 1000000 6 5 6
3 0 3
Threads average 2576.38 ns/transition
 java UnsafeMemory BetterSafe 8 1000000 6 5
6 3 0 3
Threads average 1727.23 ns/transition
java UnsafeMemory BetterSafe 8 1000000 6 5 6
3 0 3
Threads average 2174.88 ns/transition

java UnsafeMemory BetterSafe 20 1000000 6 5
6 3 0 3
Threads average 7401.36 ns/transition
java UnsafeMemory BetterSafe 20 1000000 6 5
6 3 0 3
Threads average 7056.77 ns/transition
 java UnsafeMemory BetterSafe 20 1000000 6 5
6 3 0 3
Threads average 7688.78 ns/transition

java UnsafeMemory BetterSafe 20 1000 6 5 6 3
0 3
Threads average 246350 ns/transition
java UnsafeMemory BetterSafe 20 1000 6 5 6 3
0 3
Threads average 247598 ns/transition

java UnsafeMemory BetterSafe 30 1000 6 5 6 3
0 3
Threads average 383113 ns/transition

............................................
java UnsafeMemory BetterSorry 8 1000000 6 5
3 0 3
Threads average 3929.92 ns/transition
java UnsafeMemory BetterSorry 8 1000000 6 5
6 3 0 3
Threads average 3895.70 ns/transition
java UnsafeMemory BetterSorry 8 1000000 6 5
6 3 0 3
Threads average 3436.86 ns/transition

java UnsafeMemory BetterSorry 20 1000000 6 5
6 3 0 3
Threads average 14910.6 ns/transition
java UnsafeMemory BetterSorry 20 1000000 6 5
6 3 0 3
Threads average 11086.2 ns/transition
java UnsafeMemory BetterSorry 20 1000000 6 5
6 3 0 3
Threads average 9430.48 ns/transition

java UnsafeMemory BetterSorry 20 1000 6 5 6
3 0 3
Threads average 188717 ns/transition
java UnsafeMemory BetterSorry 20 1000 6 5 6
3 0 3
Threads average 201937 ns/transition

java UnsafeMemory BetterSorry 30 1000 6 5 6
3 0 3
Threads average 312045 ns/transition
```