Homework 3: Java shared memory performance races

1 Testing

1.1 Testing Environment

Following are the specifications of my testing environment:

Java Version: 1.8.0_112
JVM: Java HotSpot(TM) 64-Bit Server VM (build 25.112-b15, mixed mode)
CPU Type: Intel(R) Xeon(R) CPU E5-2640 v2
Clock Speed: 2.00GHz
Cache Memory Size: 20480 KB
Main Memory Size: 65758316 kB

1.2 Test Cases

In order to measure the performance of the following classes, benchmark tests were performed on the `Null` and `Synchronized` classes. I modified `UnsafeMemory.java` so that the average transitions were saved to a text file after each call. I used the following bash command to gather results:

```
for run in {1..100}; do
java UnsafeMemory2 Synchronized 1 1000 126
50 50 50 50 50 50 50 50;
done;
```

then wrote a java class which took the average of 100 tests for my data, unless otherwise noted.

Table 1: Null

# Threads	1000 swaps (ns/transition)	1mil swaps
1	3132.92	37.67
2	6446.62	138.25
8	27478.21	2168.57
16	57486.13	4429.74
32	145632.12	6733.57

2 Performance and Reliability

2.1 *Synchronized*

Table 2: Synchronized

# Threads	1000 swaps (ns/transition)	1mil swaps
1	3177.88	64.35
2	6637.10	430.10
8	31276.35	2812.54
16	65502.57	5564.48
32	149203.81	12218.45

`Synchronized` guarantees 100% reliability because it prevents all other processes from accessing the array once one process has entered the critical block. As a result, transitions are much slower, but the array always has the correct sum even when there are multiple threads. `Synchronized` works well with a large number of threads so long as a large enough amount of swaps are taking place; in the case of increasing threads for only 1000 swaps, the amount of contention that occurs between processes waiting for each other and the overhead of threading actually diminishes performance greatly; it is thus better used with a larger amount of swaps.

2.2 *Unsynchronized*

Table 3: Unsynchronized

# Threads	1000 swaps (ns/transition)	1mil swaps
1	3078.60	46.63
2	6227.76	174.44
8	27361.58	1677.91
16	58860.71	4762.40
32	146044.88	7979.26

The `Unsynchronized` class is the most unreliable, as it provides no way to avoid race conditions between threads. It performs roughly the same as `Synchronized` in the single-threaded case, which makes sense since `Synchronized` does not have to halt other processes when it enters a critical region, and is 100% reliable since there is no concurrent access.

It is clearly much faster, however, in the multi-threaded cases, since it does not implement any safety measures; it simply allows the processes to perform the swaps indiscriminately, with locally cached array values, which leads to the sum of the array being mismatched in almost all cases because of the overwrites from not updating variables. It ”hangs” in multithreaded runs when the array is small and the values in the array are too close to the min or max; this is because incorrect incrementing/decrementing of array elements will cause them to fall out of the $[0, 127]$ range, thus causing our range check to return false forever (since the code only terminates after n successful swaps).

2.3 *GetNSet*

Table 4: GetNSet

# Threads	1000 swaps (ns/transition)	1mil swaps
1	3730.70	58.99
8	30765.27	1981.84

`GetNSet` uses the `get()` and `set()` methods from `AtomicIntegerArray`. These two methods work similarly to the `volatile`, in which reads and writes are retrieved and stored from main memory in order to avoid race conditions between threads. Its not as fast as `Unsynchronized`, but does maintain the array sum in more cases, but is still a bit faster than `Synchronized`. However, as the number of swaps increases, the less likely that `GetNSet` returns the correct array sum. So around 5-10 swaps will maintain reasonable safety, but anything beyond that and the integrity of the sum is usually not upheld.

GetNSet is not DRF because while `get()` and `set()` are atomic and behave like `volatile`, there are still opportunities where the transitions can mess up. For example, the values of *i* and *j* are safely retrieved by `get()`, but then stored in local variables, which `swap` then uses for range comparison and then incremented or decremented in `set()`. When we update the values in `set()`, we need to write an increment/decrement to the current value to be set; this operation is not atomic, which causes the data race conditions.

`GetNSet` suffers from the same ”hanging” issue as `Unsynchronized` when the array elements are too close to the ends of the range.

2.4 *BetterSafe*

Table 5: BetterSafe

# Threads	1000 swaps (ns/transition)	1mil swaps
1	4154.28	65.66
8	43544.08	1075.92
32	202840.01	4854.18

The main feature of `BetterSafe` is `ReentrantLock` from the `java.util.concurrent.locks` library. According to the Oracle documentation, `ReentrantLock` works nearly the same as `synchronized` plus some functionality, but the main difference in this case is that the locking is explicitly declared.

Excluding the single-threaded case, *BetterSafe* suffers from much less overhead as the number of threads increases when there is a large amount of swaps. When there are 1 million swaps, it does significantly better than *Synchronized*, with much less difference in performance with the number of threads. It does not do as well (slightly poorer) with only 1000 swaps; this seems to be a common trend across all the classes, where the trade-off for more safety is not worth the overhead for smaller amounts of operations across multiple threads. At about 10000 swaps, the performance of the two classes is nearly the same.

`BetterSafe` is better than `Synchronized` because when there is high contention, as in the case of a huge amount of swaps, `Synchronized` begins to bottleneck. The block forces the other threads, wherever they are, to stop when one thread enters the critical block to access the array. The `ReentrantLock` implementation works fairly consistently even as the threads increase. Though the other threads must pause, they only do so when they run into the block which requires the lock. This aligns with the results discussed in Goetz’s Java article, which claims that `ReentrantLock` scales better than `synchronized` (Goetz, 2004).

It is still 100% reliable, however, because the locks still ensure that only one thread is reading and writing the array state at a given time. `finally` will always execute the unlocking, even if `try` returns or throws an exception prematurely, which prevents the issue of having one thread holding the lock forever.

3 *BetterSorry* Implementation

Table 6: BetterSafe and BetterSorry Comparison

# Threads	BetterSafe	BetterSorry
2	816.38	239.01
4	663.18	548.43
8	1347.60	1186.90

The performance of `BetterSorry` is improved over `BetterSafe` through the use of the `AtomicInteger` class. `AtomicInteger` methods implicitly work like the `volatile` keyword; instead of explicitly forcing threads to stop when another thread hits a critical section, threads are allowed to access the array values, but must fetch and update the values in main memory, rather than in caches or registers specific to a thread. Thus, there is no contention for locks and several threads may access the critical section at once. The benefits of `BetterSorry` are best seen in arrays larger than our previous tests; the table data comes from testing with an array of size 100, with 1 million swaps.

Once a volatile variable is accessed, any changes made prior are also made known to other threads; instead of using volatile, however, the `AtomicInteger` class uses a compare and set method that sets the new value when the value we access is the value we expect. It guarantees atomicity at the machine level, so the `getAndIncre Decrement()` functions allow us to atomically change *i* and *j*, preventing the overwrites that occur in `Unsynchronized()` when variables are not updated properly.

It is not data-race free, however, because the atomicity still doesn't prevent other threads from changing the values of *i* and *j* inbetween these methods. For example, say a thread finishes executing in the transition time between the `get()` call and `getAndDecrement`. Suppose the thread that finished decremented the value of *i* to 0; thus, the current thread should not decrement any further. However, we've already passed the *get()* check for this, and although the *i* value was correct at that time, it has since then changed. The `getAndDecrement` will still occur and now *i* would be out of range. If we perform our test with a small range (between [0, 10]) and values 0 and 1 for a large array, `BetterSorry` will probably fail with at least one *negative output* error.

References

Atomic Access, The Java Tutorials. Retrieved February 6, 2017, from Oracle Java Documentation: https://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html.

Goetz, B. Java theory and practice: More flexible, scalable locking in JDK 5.0, 2004. Retrieved February 6, 2017, from IBM developerWorks: http://www.ibm.com/developerworks/java/library/j-jtp10264/.

Peticolas, Dave.