Homework 3 Report: Java shared memory performance races

Eric Sehun Oh
304184918

Explain why your BetterSafe implementation is faster than Synchronized and why it is still 100% reliable.

The implementation for BetterSafe uses locks from the ReentrantLock class which has the same concurrency and memory semantics as the 'synchronized' keyword mechanism, which is the design choice used in the Synchronized class. The ReentrantLock class however adds additional functionalities such as its ability to try for lock interruptibly and its ability to timeout while waiting for the lock.

Another huge advantage of ReentrantLock is that it is much more scalable, since it offers far better performance when many threads are attempting to access a shared resource. As the number of threads increase per run, the average time decreases significantly and is less than those from the synchronized implementation.

Since both implementations offer mutual exclusion through protection of critical sections they are both guaranteed to be a 100% reliable.

Explain why your BetterSorry implementation is faster than BetterSafe. Explain why it is more reliable than Unsynchronized, by describing race conditions that Unsynchronized is liable to but BetterSorry avoids. Give a race condition that BetterSorry still suffers from.

The BetterSorry implementation uses an the AtomicInteger class versus the ReentrantLock class used in BetterSafe. AtomicInteger is much faster than using ReentrantLock because it does not use locks to protect critical sections but rather atomic instructions at the hardware level. When using locks, other threads must wait but when using a single CPU instruction there is less overhead.

In the Unsynchronized class, data is not protected and there is no mutual exclusion to protect the critical sections. However in the BetterSorry class updates are done atomically meaning that the update will definitely complete or not start at all. This atomicity provides a greater reliability than the Unsynchronized class.

However, the BetterSorry implementation suffers from a race condition in the swap function where the AtomicInteger array 'value' could be checked in the if statement and changed even before the getAndIncrement or getAndDecrement methods.

Can you write a test program that causes this race condition to occur with high probability and thus to make BetterSorry fail? If so, please supply it (and you can reuse it to answer questions below); if not, explain why not.

Yes, this race condition occurs after the if loop and before the increment and decrement methods. For a higher probability to make BetterSorry fail, there should be a greater amount of time between these sections of code. Perhaps more code between these lines of code would possibly cause BetterSorry to fail since there is a chance that the value would change after being checked.

```
public boolean swap(int i, int j) {
      if (value[i].get() <= 0
||value[j].get() >= maxval)
      //more line of code
            return false;
        // more lines of code
        value[i].getAndDecrement();
        value[j].getAndIncrement();
        return true;
    }
}
```

Discuss any problems you had to overcome to do your measurements properly. Explain whether and why the class is DRF; if it is not DRF give a reliability test (as a shell command "java UnsafeMemory model ...") that the class is extremely likely to fail on the SEASnet GNU/Linux servers.

For classes, Unsynchronized and GetNSet, there was trouble testing for the output because with high number of threads and iterations the process would not complete. For these two classes, the two calls below will be highly likely to fail because they are not DRF.

- `java UnsafeMemory Unsynchronized 8 100000 6 5 6 3 0 3`
- `java UnsafeMemory GetNSet 8 100000 6 5 6 3 0 3`

The BetterSorry class is not DRF either however it seems to pass most combinations that I ran it with. Since the result is nondeterministic it is not possible to say exactly which command would cause it to fail.

The other classes Null, Synchronized and BetterSafe are all DRF. Synchronized and BetterSafe are implemented by locks which protect the critical sections in which data updated.

Comparison of Implementations

```
java UnsafeMemory Null 4 1000 6 5 6 3 0 3
Threads average 6014.73 ns/transition
java UnsafeMemory Unsynchronized 4 1000 6 5 6 3 0 3
Threads average 6510.33 ns/transition
sum mismatch (17 != 19)
java UnsafeMemory Synchronized 4 1000 6 5 6 3 0 3
Threads average 8014.89 ns/transition
java UnsafeMemory GetNSet 4 1000 6 5 6 3 0 3
Threads average 9504.04 ns/transition
sum mismatch (17 != 14)
java UnsafeMemory BetterSafe 4 1000 6 5 6 3 0 3
Threads average 12161.7 ns/transition
java UnsafeMemory BetterSorry 4 1000 6 5 6 3 0 3
Threads average 7435.38 ns/transition

java UnsafeMemory Synchronized 6 100000 6 5 6 3 0 3
Threads average 1344.06 ns/transition
java UnsafeMemory BetterSafe 6 100000 6 5 6 3 0 3
Threads average 1432.25 ns/transition
java UnsafeMemory BetterSorry 6 100000 6 5 6 3 0 3
Threads average 795.726 ns/transition

java UnsafeMemory Synchronized 8 1000000 6 5 6 3 0 3
Threads average 858.229 ns/transition
java UnsafeMemory BetterSafe 8 1000000 6 5 6 3 0 3
Threads average 697.150 ns/transition
java UnsafeMemory BetterSorry 8 1000000 6 5 6 3 0 3
Threads average 644.473 ns/transition

java UnsafeMemory Synchronized 12 10000000 6 5 6 3 0 3
Threads average 948.365 ns/transition
java UnsafeMemory BetterSafe 8 1000000 6 5 6 3 0 3
Threads average 807.232 ns/transition
java UnsafeMemory BetterSorry 8 1000000 6 5 6 3 0 3
Threads average 676.423 ns/transition
```

For Unsynchronized and GetNSet, the process hangs when ran with 8 threads and > 10000 iterations.

As the number of threads and iterations increase, BetterSorry implementation becomes consistently the fastest. However, this would not be the preferred choice for GDI's implementation as it does not provide 100% reliability. Instead the choice is BetterSafe.