Recap: optimistic retries

We previously discussed the idea of optimistic retries, where a copy of the object was made, the desired operation was performed on the copy, and then there was a “small” commit method that was synchronized and called with the old and new values such that if the old value was still the current value, it would become the new value.

The general idea was to try to minimize the number of tests and operations that had to be performed within a synchronized critical section of code.

One risk was that if there was high contention for that critical section, there would be many (expensive) failed attempts, and many threads queuing up to gain access to that critical section over and over.

What if you could use the same general approach seen in optimistic retries but do so without any synchronized critical section?
Recap: java.util.concurrent.atomic.AtomicInteger

Similar to how ConcurrentHashMap has compound methods, the AtomicInteger class provides several compound methods. These provide for effectively atomic read-modify-write operations. However, these atomic compound methods do not use locking to achieve atomicity.

The key method to this is:

\[ \text{compareAndSet} \text{ (expectedVal, newVal) } \]

which will only set the value to \text{newVal} if the current value is \text{expectedVal} and returns a boolean value based on whether or not it was successful.

For example, \text{getAndSet} \text{ (newVal) } is a method which calls \text{compareAndSet} to set the value to \text{newVal} and return the previous value at the time that the “set” was actually successful.

Recap: The compareAndSet method

At the heart of the AtomicInteger is the compareAndSet method. The current version of the Java concurrency library has the following in the body of this method:

\[
\text{return unsafe.compareAndSwapInt(this, valueOffset, expect, update);}
\]

Based on the platform this uses hardware-level instructions to accomplish the atomic operation (for example, instructions like either the CMPXCHG8B or CMPXCHG16B compare and exchange instruction for Intel chips).
Recap: Only effectively atomic...

It is possible that some other thread or even threads have performed an action during something some `ThreadX` sees as an effectively atomic operation if the other thread or threads ends up setting the value to what it had been when the initial “get” was done by `ThreadX` before `ThreadX` actually has its `compareAndSet` performed.

An issue with pseudo-random number generators...

Something that I didn’t consider in some earlier examples where for “busy work” I had (for example) a consumer generating random numbers is that a typical pseudo-random number generator will base the next value on the previous value(s).

How it does this can impact the runtime of the number generator itself if all of the threads are sharing a single number generator.

Consider the following examples from Java Concurrency in Practice…
Example number generator:

```java
public class PseudoRandom {
    int calculateNext(int prev) {
        prev ^= prev << 6;
        prev ^= prev >>> 21;
        prev ^= (prev << 7);
        return prev;
    }
}
```

Guarding the seed with a lock:

```java
public class ReentrantLockPseudoRandom extends PseudoRandom {
    private final Lock lock = new ReentrantLock(false);
    private int seed;
    ReentrantLockPseudoRandom(int seed) {this.seed = seed;}
    public int nextInt(int n) {
        lock.lock();
        try {
            int s = seed; seed = calculateNext(s);
            int remainder = s % n;
            return remainder > 0 ? remainder : remainder + n;
        } finally {lock.unlock();}
    }
}
```
Guarding the seed with an atomic integer:

```java
public class AtomicPseudoRandom extends PseudoRandom {
    private AtomicInteger seed;
    AtomicPseudoRandom(int seed) {
        this.seed = new AtomicInteger(seed);
    }
    public int nextInt(int n) {
        while (true) {
            int s = seed.get();
            int nextSeed = calculateNext(s);
            if (seed.compareAndSet(s, nextSeed)) {
                int remainder = s % n;
                return remainder > 0 ? remainder : remainder + n;
            }
        }
    }
}
```

What happens as more threads are sharing a generator?
Invariants over multiple variables

While classes such as `AtomicInteger` (and `AtomicBoolean` and `AtomicLong`) are provided, some are not. For example, there is no `AtomicFloat` (though clever use of `floatToIntBits` can give you one).

However, what if you have two or more variables involved in an invariant and want to set/test them all atomically without locking? One option is to take advantage of `AtomicReference` and create a new class that holds the related variables in its own structure. The class itself can then have a setter for each of the variables.

For example, if we wanted a range of integers that was always non-empty, the first value would always have to be less than the second, we could create something like `AtomicNonEmptyRange` (see next slide).

```java
public class AtomicNonEmptyRange {
    private static class Pair<T> {
        final T first, second;
        public Pair(T firstIn, T secondIn) {
            first = firstIn; second = secondIn;
        }
    }

    private final AtomicReference<Pair<Integer>> range
        = new AtomicReference<Pair<Integer>>(new Pair<Integer>(0, 0));

    public boolean setLower(int newFirst) {
        Pair<Integer> currentRange, newRange;
        while (true) {
            currentRange = range.get();
            if (newFirst > currentRange.second) {return false;}

            newRange = new Pair<Integer>(newFirst, currentRange.second);
            if (range.compareAndSet(currentRange, newRange)) {
                return true;
            }
        }
    }

    // same basic idea for setUpper
}
```
Some non-blocking algorithm terminology

Within the umbrella of “non-blocking” algorithms are the categories wait-free, lock-free, and obstruction free.

**wait-free**: every thread is making forward progress

**lock-free**: something is always making forward progress

**obstruction free**: if a thread is running without any other threads competing for the same resources then it will complete within a fixed number of steps