Recap: optimistic retries (Set 12, Slide 19)

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?
Using `Integer` objects without locks at all?

An `Integer` object can have many methods accessing it without worrying about locking due to the atomic nature of the code within the method and the fact that it is immutable.

Let’s assume a mutable version where the actual `int` is held in an instance variable named `value`.
- create a new `Integer` without needing locks.
- get a value from an `Integer` without needing locks since the body would be `return value;`
- set a value in an `Integer` without needing locks since the body would be `value = valueIn;`

However, performing compound actions like either increment or decrement (either by 1 or by an arbitrary value) is a problem since the only technique we have so far to make those atomic is through some form of locking.

`java.util.concurrent.atomic.AtomicInteger`

Similar to how `ConcurrentHashMap` has compound methods, the `AtomicInteger` class provides several compound methods. There 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: `compareAndSet(expectedVal, newVal)` which will only set the value to `newVal` if the current value is `expectedVal` and returns a `boolean` value based on whether or not it was successful.

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

```java
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).

---

Compound Operation: Get and Increment

```java
public final int getAndIncrement() {
  for (;;) {
    int current = get();
    int next = current + 1;
    if (compareAndSet(current, next))
      return current;
  }
}
```

If there is high contention for a particular `AtomicInteger` object’s value to be incremented, what happens?
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.

Consider how the following might play out:

• Starting value: 1
• Thread 1 increments
• Thread 2 increments
• Thread 3 decrements

One possible trace...

Starting value: 1
Thread 1 increments
Thread 2 increments
Thread 3 decrements

T1 reads a 1
T2 reads a 1
T1 increments to 2 and tries to store (value is “still 1” so succeeds)
T3 reads a 2
T3 decrements to 1 and tries to store (value is “still 2” so succeeds)
T2 increments to 2 and tries to store (value is “still 1” so succeeds)

However, what T2 doesn’t realize is this is a “different” value of 1 than it had seen when it did its initial read!