Exercises week 10
Mandatory handin 5
Friday 6 November 2015

Goal of the exercises

The goal of this week's exercises is to make sure that you can write concurrent programs using the transactional memory approach to mutable shared state, and that you can assess the advantages and pitfalls of (optimistic) transactional concurrency compared to the lock-based pessimistic concurrency.

Do this first

Get and unpack this week's example code in zip file pcpp-week10.zip on the course homepage.

Download the Multiverse library `multiverse-core-0.7.0.jar` from the public course homepage, and put it in a suitable place such as `~/lib/multiverse-core-0.7.0.jar`.

To compile and run a Java file such as `TestAccounts.java` with Multiverse use the following commands:

```
javac -cp ~/lib/multiverse-core-0.7.0.jar TestAccounts.java
java -cp ~/lib/multiverse-core-0.7.0.jar:. TestAccounts
```

Exercise 10.1 **(Optional)** Test and time the queue implementation in TestStmQueues.java.

1. This week's lecture presented a bounded queue class StmBoundedQueue implemented using transactional memory. Test it using the test suite developed in week 8's lecture; see file TestBoundedQueueTest.java. Does the new queue pass those tests?
2. Measure the overall time to run the above-mentioned test, on week 8's lock-based SemaphoreBoundedQueue implementation as well as on this week's StmBoundedQueue implementation. Use the simple Timer class directly to measure the time from right after passing the start CyclicBarrier to right after passing the end CyclicBarrier; you do not need the Mark6 or Mark7 timing infrastructure. How well does the transactional queue implementation perform compared to the lock-based one from week 8?

Exercise 10.2 Implement a histogram class `StmHistogram` using transactional memory. You should use the Multiverse library for Java and *not* `synchronized`, locks, or Java atomics. Build on the partial solution in file stm/TestStmHistogram.java.

The basic histogram functionality is as in the week 2 exercises, but the interface now is slightly different:

```
interface Histogram {
  void increment(int bin);
  int getCount(int bin);
  int getSpan();
  int[] getBins();
  int getAndClear(int bin);
  void transferBins(Histogram hist);
}
```

Method `getAndClear(bin)` should atomically return the count in bin `bin` of the histogram and also reset that count to zero.

Method `transferBins(hist)` should transfer all the counts from `hist` to this histogram, by adding each bin count in `hist` to this histogram and atomically also setting that count to zero. Thus if `this.getCount(7)` is 20 and `hist.getCount(7)` is 30 before the call, and there are no other calls going on, then after the call `this.getCount(7)` is 50 and `hist.getCount(7)` is 0.

(The `transferBins` operation is more meaningful and useful than the `addAll` implemented in previous exercises. In an application where multiple threads update their own histograms, `transferBins` can be used to periodically aggregate the thread-local histograms into a common global histogram, without losing or duplicating any counts.)

1. Implement the basic methods `increment`, `getCount` and `getSpan` in your StmHistogram class. The bins of the histogram should be held as transactional integer variables, that is:

   ```
   class StmHistogram implements Histogram {
     private final TxnInteger[] counts;
     ...
   }
   ```

2. The file stm/TestStmHistogram.java contains code to run 10 threads in parallel to count the number of prime factors in all the numbers in the range 0. . . 3 999 999. It uses your transactional histogram implementation to maintain the counts.

   The correct result should look like this:

   ```
   0:         2
   1:    283146
   2:    790986
   3:    988651
   4:    810386
   5:    524171
   6:    296702
   7:    155475
   8:     78002
   9:     38069
   ... and so on
   ```

   showing that 283 146 numbers in 0. . . 3 999 999 have 1 prime factor (those are the prime numbers), 790 986 numbers have 2 prime factors, and so on. (The 2 numbers that have 0 prime factors are 0 and 1). And of course the numbers in the second column should add up to 4 000 000. Run this code. Does it produce the correct result with your histogram implementation?

3. Implement method `getBins` so that it returns an array of the counts in the bins of the histogram.

4. Implement method `getAndClear(bin)` so that it atomically returns the count in bin `bin` of the histogram and also resets that count to zero.

5. Implement method `transferBins(hist)` so that it transfers all the counts from `hist` to this histogram, by adding each bin count in `hist` to this histogram and atomically also setting that count to zero. The `getAndClear` method should be useful in doing this.

   Note that `transferBins(hist)` can be implemented in at least two ways: (a) use one large transaction that transfers all the bins; or (b) use a transaction for each bin that atomically transfers that bin. Since (b) gives shorter transactions and therefore less likelihood that the transaction will fail and be retried, it is probably much preferable to (a). With (b), if a concurrent thread increments the `hist` counts, there may be no point in time at which all bins of `hist` are actually zero. This is acceptable so long as no counts are lost and no counts are duplicated during the transfer.

6. Now extend the code from subquestion 2 so that the main thread creates a new StmHistogram instance called `total` and occasionally calls `total.transferBins(histogram)` where `histogram` is the one the prime counting threads write to.

   The main thread may do this 200 times, say every 30 milliseconds by calling `Thread.sleep(30)` in between the calls to `transferBins`. It should start doing this only after the prime counting threads have been started. When all the threads have terminated, the main thread should call `dump(total)` to print the `total` histogram.

   At the end `histogram`'s counts should be all zero, and `total`'s counts should be what `histogram`'s used to be, regardless when and how many times `transferBins` has been called. Is this the case?

7. What effect would you expect `total.transferBins(total)` to have? What effect does it have in your implementation? Explain.

Exercise 10.3 Implement a concurrent hash map `StmMap` using transactional memory. Start from the sketch in file stm/TestStmMap.java. You should use the Multiverse library for Java and *not* `synchronized`, locks, or Java atomics. Thanks to the visibility effects of the Multiverse `atomic` transactions, there is no need for subtle `volatile` tricks or similar.

The basic map functionality should be as in the week 6 and 8 exercises, but you can ignore the internal `reallocateBuckets` method — although it is quite easy to implement it *correctly* using transactional memory, it is not clear how to implement it *efficiently*.

Use immutable ItemNode<K,V> nodes as in the StripedWriteMap implementation; you can reuse the item node class exactly as it is.

The entries in the `buckets` array are mutable and updates to them must be under transactional control, so each entry must have type TxnRef<ItemNode<K,V>>. Moreover, the `buckets` field itself is mutable and must be a TxnRef<. . . >, so in total `buckets` should have this somewhat impressive declaration:

```
class StmMap<K,V> implements OurMap<K,V> {
  private final TxnRef<TxnRef<ItemNode<K,V>>[]> buckets;
  ...
}
```

That is, a transactional reference to an array of transactional references to ItemNode<K,V> objects.

1. Implement the `get` method. You can either (a) enclose the entire method body in `atomic`, or (b) use `atomic` only around the code that accesses the `buckets` array reference and indexes into the array. Doing (a) seems simplest, but (b) better separates the transactional `buckets` accesses from the subsequent readonly search of the immutable item node lists and so makes the transaction shorter.

2. Implement the `forEach(consumer)` method. Use approach (b) outlined in the previous subquestion to keep the transaction short. The call to `consumer(node.k, node.v)` should clearly not be inside a transaction, because the transaction may fail and be restarted, in which case `consumer` may be called an arbitrary number of times for each entry in the hash map — very unlikely to be what is expected.

3. Implement the `put`, `putIfAbsent` and `remove` methods. Do not worry about updating the size count for now. Briefly explain why you believe your implementations are correct.

4. Implement the `size` method. The simplest approach is to use a single TxnInteger to hold the total number of entries in the hash map, and update that field inside the `put`, `putIfAbsent`, and `remove` transactions. That will probably be a concurrency bottleneck and lead to poor scalability on a manycore machine, but this is acceptable here.

5. Discuss the problems involved in implementing `reallocateBuckets` efficiently using transactions and optimistic concurrency. There seems to be at least two problems: (1) It makes no sense to transfer only half the hash table buckets from the old `buckets` to the new one, so the reallocation should be in a transaction. But that might be a very long transaction, with a high likelihood that some concurrent `put`, `putIfAbsent` or `remove` transaction causes the reallocate transaction to abort and then restart, again and again, wasting much computation. Moreover (2) by reallocating optimistically, many threads could start overlapping reallocations, and in the end only one of those transactions will succeed, again wasting all the computation performed in the failing transactions.

   So it seems that one needs a protocol by which all other updating (`put`, `putIfAbsent`, `remove`) threads block when one thread has started a reallocation. Could one have a transactional field `newBuckets` alongside `buckets`, with the convention that `newBuckets` is non-`null` exactly while one thread is reallocating, and all other mutating threads wait for `newBuckets` to be `null`? How does a thread block using transactions? (Hint: see the lecture's bounded queue implementation).

   Discuss this idea in approximately 15 lines of text; you do not need to implement it.