Exercises week 7
Friday 10 October 2014

Goal of the exercises

The goal of this week's exercises is to make sure that you can achieve good performance and scalability of lock-based concurrent software, using finer-grained locks, lock striping, the Java class library's atomically updatable numbers, immutability and the visibility effects of volatiles and atomics.

Due to the fall break, the handin deadline for these exercises is Thursday 23 October 2014.

Do this first

Get and unpack this week's example code in zip file pcpp-week07.zip on the course homepage.

File TestStripedMap.java contains implementations of several thread-safe hash map classes:

(A) A complete implementation of SynchronizedMap<K,V> which follows the Java monitor pattern: all mutable fields are private, all public methods are synchronized, and no internal data structures escape.

(B) A partial implementation of StripedMap<K,V> which does not follow the Java monitor pattern, but divides the buckets table into stripes, locking each stripe both on read and write accesses. This is the subject of Exercise 7.1.

(C) A partial implementation of StripedWriteMap<K,V> which also divides the buckets table into stripes, but locks each stripe only on write accesses. Read accesses do not take locks at all, but their correctness is assured — we hope — by (1) working on immutable item nodes, and (2) ensuring visibility of writes by careful use of atomics and volatiles. This is the subject of Exercise 7.2.

(D) A simple wrapper WrapConcurrentHashMap<K,V> around Java's ConcurrentHashMap<K,V>, for comparison.

Exercise 7.1 The SynchronizedMap<K,V> implementation scales (and therefore performs) poorly on a multicore computer because of all the locking: only one thread at a time can read or write the hash map.

The lecture showed that scalability can be considerably improved by *lock striping*. Instead of locking on the entire table of buckets, one divides it into a number of stripes (here 32), and locks only the single stripe that is going to be read or updated.

This is the idea in the StripedMap<K,V> class, whose implementation in file TestStripedMap.java contain only methods `containsKey` and `put` and some auxiliary methods.

Your task below is to implement the remaining public methods, as described by interface OurMap<K,V>. They are very similar to the method implementations in class SynchronizedMap<K,V>, except that they do not lock the entire hash map. only the relevant stripe.

1. Implement method `V get(K k)` using lock striping. It is similar to `containsKey`, but returns the value associated with key `k` if it is in the map, otherwise `null`. It should use the `ItemNode.search` auxiliary method.

2. Implement method `int size()` using lock striping; it should return the total number of entries in the hash map. The size of stripe `s` is maintained in `sizes[s]`, so the `size()` method should simply compute the sum of these values, locking each stripe in turn before accessing its value.

   Explain why it is important to lock stripe `s` when reading its size from `sizes[s]`?

3. Implement method `V putIfAbsent(K k, V v)` using lock striping. It is very similar to `putIfAbsent` in class SynchronizedMap<K,V> but should of course only lock on the stripe that will hold key `k`. It should use the `ItemNode.search` auxiliary method. Remember to increment the relevant `sizes[stripe]` count if any entry was added.

4. Implement method `V remove(K k)` using lock striping. Again very similar to SynchronizedMap<K,V>. Remember to decrement the relevant `sizes[stripe]` count if any entry was removed.

5. Implement method `void forEach(Consumer<K,V> consumer)`. This may be implemented in two ways: either (1) iterate through the buckets as in the SynchronizedMap<K,V> implementation, locking the corresponding stripe before accessing the bucket; or (2) iterate over the stripes, and for each stripe iterate over the buckets that belong to that stripe. The latter takes each stripe lock only once, instead of potentially thousands of time. Explain your implementation.

   In both cases, since `forEach` reads the volatile `buckets` field several times but locks only stripe-wise, it must first obtain a reference `theBuckets` and then use that in the rest of the method, like this:

   ```
   public void forEach1(Consumer<K,V> consumer) {
     final ItemNode<K,V>[] bs = buckets;
     for (int hash=0; hash<bs.length; hash++) {
       synchronized (locks[hash % lockCount]) {
         ItemNode<K,V> node = bs[hash];
         ...
       }
   }
   ```

   Otherwise another thread may call `reallocateBuckets` and hence replace the `buckets` array with one of a different size between observing the length of the array and accessing its elements.

6. You may use method `testMap(map)` for very basic single-threaded functional testing while making the above method implementations. See how to call it in method `testAllMaps`. To actually enable the `assert` statements, run with the `-ea` option:

   ```
   java -ea TestStripedMap
   ```

7. Measure the performance of SynchronizedMap<K,V> and StripedMap<K,V> by timing calls to method `exerciseMap`. Report the results from your hardware and discuss whether they are as expected.

8. What advantages are there to using a small number (say 32 or 16) of stripes instead of simply having a stripe for each entry in the buckets table? Discuss.

9. Why can using 32 stripes improve performance even if one never runs more than, say, 16 threads? Discuss.

10. (Subtle, but answered in the source code) Why is it important for thread-safety that the number of buckets is a multiple of the number of stripes?

Note that method `reallocateBuckets` has been provided for you. Its auxiliary method `lockAllAndThen` uses recursion to take all the stripe lock; this is the only way in Java to take a variable number of intrinsic locks.

Exercise 7.2 The striped hash map in class StripedMap<K,V> scales better with more threads than the SynchronizedMap<K,V> hash map. However, it can be further improved by locking a stripe only when writing, not when reading, so that many reads can proceed concurrently without locking. This idea is outlined in class StripedWriteMap<K,V>, which is a somewhat subtle undertaking, based on several ideas that are different from both SynchronizedMap<K,V> and StripedMap<K,V>.

First, the item nodes are made immutable, all fields of class ItemNode<K,V> are `final`. That means that as soon as a read access (`containsKey`, `get` or `forEach`) has obtained a reference to a list of item nodes in a bucket, it need not be concerned with atomicity or visibility: nothing it accesses can be affected by other threads.

Second, the slice sizes will now be represented by an AtomicIntegerArray so that no locking is needed when incrementing and decrementing the stripe sizes. It also ensures that a thread executing the `size()` method can see the increments and decrements made by threads that `put`, `putIfAbsent` and `remove` entries.

Third, the writes to and reads from the `sizes` array are (ab)used to ensure visibility of updates to the `buckets` array. After any write to an element of `buckets`, `sizes` is written also, and before any read of an element of `buckets`, `sizes` is read. This ensures that `containsKey`, `get`, `forEach` will see any writes performed by `put`, `putIfAbsent` and `remove`.

Fourth, making class ItemNode<K,V> immutable means that `put` and `remove` may need to copy part of the list of item nodes in a bucket, but those lists should in any case hold at most a few items (otherwise the hash map

is slow), the allocation of a new item node is fast, the cost appears to be outweighed but the time saved on not locking, and parts of the code become much neater this way.

Some ideas in StripedWriteMap<K,V> are inspired by the implementation of Java's ConcurrentHashMap, which however uses many more sophisticated techniques.

1. Implement method `int size()`. This is very straightforward: simply compute the sum of the stripe sizes. Since these are represented in an AtomicIntegerArray, all writes are visible to this method's reads; no locking is needed.

2. Implement method `V putIfAbsent(K k, V v)`. You must lock on the relevant stripe. Use auxiliary method `ItemNode.search(bl, k, old)` to determine whether `k` is already in the hash map, where `bl` is the bucket list reference obtained from `buckets[hash]`. If yes, then do nothing; else create a new item node from `k`, `v` and `bl`, and update the `buckets` table with that. Remember to update the stripe size if an entry was added.

   Why do you not need to write to the stripe size if nothing was added?

3. Implement method `V remove(K k)`. Lock on the relevant stripe. Use `ItemNode.delete(bl, k, old)` to delete the entry with key `k`, if any, from bucket list `bl`, and update the buckets table with the result. Remember to update the stripe size if an entry was removed.

4. Implement method `void forEach(Consumer<K,V> consumer)`. Same comments apply as Exercise 7.1.5, but additionally you must read the stripe's size before iterating over its buckets, for visibility of writes.

5. Measure the performance of SynchronizedMap<K,V>, StripedMap<K,V> StripedWriteMap<K,V> and WrapConcurrentHashMap<K,V> using method `exerciseAllMaps`. Report the results and discuss whether they are as expected.

6. (Optional, only really interesting if you have access to a computer with many cores) Measure the scalability of the four hash map implementations by running method `timeAllMaps`. Report the results, in tabular or graphical form, and discuss the results.

Exercise 7.3 File TestLongAdders.java contains several implementations of a long (64-bit) integer with thread-safe `add` and `get` operations: (a) Java's AtomicLong; (b) Java 8's LongAdder; (c) a simple `long` field with synchronized operations; (d) a number represented as the sum of multiple "stripes" densely allocated in an AtomicLongArray; and (e) a number represented as the sum of multiple "stripes" allocated as scattered AtomicLong objects.

If you do not have Java 8, delete or comment out the code (b) that uses class LongAdder.

1. Compile the file and run the code to measure, on your own hardware, the performance of the various atomic long implementations. Report the numbers and discuss whether they are plausible, eg. relative to the number of cores in your machine and the number of threads trying to access the thread-safe long integer.

2. Create a new class NewLongAdderLessPadded as a variant of the NewLongAdderPadded class, where you remove the strange `new Object()` creations in the constructor. Create a suitable version of the method `exerciseNewLongAdderPadded` where you measure the performance of this new class along with the others.

   Do those `new Object()` allocations make any difference, positive or negative, on your hardware?