Exercises lesson 7

Last update 2020-10-05

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.

Exercise 7.1 *Green*

The file LongArrayListUnsafe.java contains a simple implementation of an ArrayList of Longs.

In this exercise you must make it thread-safe, and minimize locking to make it as efficient as possible.

In particular, it will not be considered a solution simply to make all methods synchronized.

The design choices is entirely up to you. But, you are expected to consider at least the following aspects:

- Use of a striped locking strategy
- Use of one or more of classes from java.util.concurrent.atomic.
- Use of synchronizers - semephore, readers-writers lock, or something similar

You are expected to back up the quality of your solution by performance measurements on at least the operations get() and add(). You should consider how to find a meaningful performance measure "under heavy load". Including how to define "heavy load". For example, trying to see if your solution scales with the number of calls to get(), calls to add(), and the number of threads doing it at the same time.

This exercise is deliberately open-ended. We have been through a lot of issues, and you should be able to put it together in this exercise.

Exercise 7.2

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 7.2.

(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 **??**.

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

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.

Green

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 could 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 must work as in Java's ConcurrentHashMap, that is, `atomically` do the following: check whether key `k` is already in the map; if it is, return the associated value; and if it is not, add `(k,v)` to the map and return null. The implementation can be 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. Ignore `reallocateBuckets` for now.

Yellow

4. Extend method `putIfAbsent` to call `reallocateBuckets` when the bucket lists grow too long. For simplicity, you can test whether the size of a stripe is greater than the number of buckets divided by the number of stripes, as in method `put`.
5. 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.
6. Implement method `void forEach(Consumer<K,V> consumer)`. Apparently, this may be implemented in two ways: either (1) iterate through the buckets as in the SynchronizedMap<K,V> implementation; or (2) iterate over the stripes, and for each stripe iterate over the buckets that belong to that stripe.

   It seems that (1) requires locking on all stripes before iterating over the buckets. Otherwise an intervening `reallocateBuckets` call on another thread may replace the `buckets` array with one of a different size between observing the length of the array and accessing its elements. It does not work to just obtain a copy `bs` of the `buckets` field, because `reallocateBuckets` destructively redistributes item node lists from the old buckets array to the new one.

   It seems that (2) can be implemented by locking only stripe-wise and then iterating over the stripe's buckets. While holding the lock on a stripe, no `reallocateBuckets` can happen.

   Explain what version you have implemented and why.
7. 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.
8. Why can using 32 stripes improve performance even if one never runs more than, say, 16 threads? Discuss.

Red

9. 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
   ```
10. 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.
11. A comment in the example source code says that it is important for thread-safety of StripedMap and StripedWriteMap that the number of buckets is a multiple of the number of stripes.

Give a scenario that demonstrates the lack of thread-safety when the number of buckets is `not` a multiple of the number of stripes. For instance, use 3 buckets and 2 stripes, and consider two concurrent calls `put(k1,v1)` and `put(k2,v2)` where the hash code of `k1` is 5 and the hashcode of `k2` is 8.

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.