The Art of Multiprocessor Programming
Solutions to Exercises
Chapter 13

July 14, 2009

Exercise 158. Modify the StripedHashSet to allow resizing of the range lock array using read/write locks.

Solution. See Figure **??**

Exercise 159. For the LockFreeHashSet, show an example of the problem that arises when deleting an entry pointed to by a bucket reference, if we do not add a *sentinel* entry, which is never deleted, to the start of each bucket.

Solution. Consider a situation where node a has predecessor node p and successor node s in the list, and a is also an entry in the bucket array. An add() call intending to insert an entry b between a and s can find a in two ways: either through a's bucket entry, or by scanning directly through the list.

To remove a, we need to redirect both p next field and the bucket entry to s. Suppose we first redirect p's next field. A concurrent add(b) call that finds the bucket entry for a will link b between a and s. The resulting list is incorrect, however, because neither a nor b are now reachable from the start of the list

Suppose instead we first redirect the bucket entry from a to s. A concurrent add(b) call that finds a by scanning through the list will link b between a and s. The resulting list is incorrect, however, because the bucket entry should refer to b instead of to b's successor.

Exercise 160. For the LockFreeHashSet, when an uninitialized bucket is accessed in a table of size N, it might be necessary to recursively initialize (i.e., split) as many as $O(\log N)$ of its parent buckets to allow the insertion of a new bucket. Show an example of such a scenario. Explain why the expected length of any such recursive sequence of splits is constant.

Solution.

Part 1 Grow the table to size 2^i while adding only keys with least signficant bit 0. If we then add a key consisting of all 1s, we will need to initialize i dummy nodes of the form $1 \cdots 10$, where the length of the contiguouis sequence of 1s ranges from 1 to $i - 1$.

Part 2 See the Shalev-Shavit paper in the notes for a proof that the expected number of initialized buckets is constant.

Exercise 161. For the LockFreeHashSet, design a lock-free data structure to replace the fixed-size bucket array. Your data structure should allow an arbitrary number of buckets.

```
public class DynamicBuckets<T> {
  int capacity;
  BucketList[] table;
  AtomicReference<DynamicBuckets<T>> overflow;
  public DynamicBuckets(int myCapacity) {
    capacity = myCapacity;
    table = (BucketList<T>[]) new BucketList[capacity];
    for (int i = 0; i < capacity; i++) {
      table[i] = null;
    }
    overflow = new AtomicReference<DynamicBuckets<T>>(null);
  }
  public BucketList<T> get(int i) {
    if (i < capacity) {
      return table[i];
    } else {
      grow();
      DynamicBuckets<T> theRest = overflow.get();
      return theRest.get(i - capacity);
    }
  }
  public void set(int i, BucketList<T> bucketList) {
    if (i < capacity) {
      table[i] = bucketList;
    } else {
      grow();
      DynamicBuckets<T> theRest = overflow.get();
      theRest.set(i - capacity, bucketList);
    }
  }
  void grow() {
    if (overflow.get() == null) {
      DynamicBuckets<T> theRest = new DynamicBuckets(2 * capacity);
      overflow.compareAndSet(null, theRest);
    }
  }
}
```

Figure 1: Lock-Free, arbitrary-size bucket container

Solution. See Figure 1. The simplest approach is to use an array with an overflow field. The overflow field is an AtomicReference<> to another, larger instance of the same class. Getting or setting an array element within the array bounds accesses the array directly. If the index is larger than the array index, the get or set method checks whether an overflow container has been allocated. If not, it allocates one with twice the size of the current container. Multiple threads may try to allocate an overflow container simultaneously, so we use compareAndSet() to ensure that only one is installed. Once we have confirmed that an overflow container exits, we forward the get or set call there (after adjusting the index).

Exercise 162. Outline correctness arguments for LockFreeHashSet's add(), remove(), and contains() methods.

Hint: you may assume the LockFreeList algorithm's methods are correct.

Solution. See the Shalev-Shavit paper for a complete proof.