Spring, 2015 ECEN 4003, CSCI 4830

Concurrent Programming

Homework 3

Assigned: 03/02/2015; Due: 03/16/2015, before class.

Please submit your homeworks through Moodle.

Problem 1

In this problem, you will write three different implementations of concurrent list-based sets, and test their performance.

1. Implement the Coarse-grained, Fine-grained, and Lazy algorithms for the list-based set (Chapter 9 of the textbook).

2. Implement another version of the Lazy algorithm, where the `remove` method just marks the node to be removed, without physically removing it. You will need to add another method, `cleanUp()`, that is called from time to time to physically remove marked nodes from the list.

3. Compare the performance (running time) of the four implementations in various settings (number of threads, ratio of calls to `contains`, `add`, `remove` (and `cleanUp`); length of the list; whether or not the method access the same region of the list; etc). If possible, for each of the four algorithm find a setting where it is fastest.

4. Propose a method to automatically test for the correctness of your implementations. (You do not need to implement it.) One such method might be based on comparing a list produced by running a number of operations in parallel to a list produced by the same operations running sequentially. What is the difficulty here?

Problem 2

You learn that your competitor, the Acme Atomic Register Company, has developed a way to use Boolean (single-bit) atomic registers to construct an efficient write-once single-reader single-writer atomic register. Through your spies, you acquire the code fragment shown, which is unfortunately missing the code for read(). Your job is to devise a read() method that works for this class, and to justify (informally) why it works. (Remember that the register is write-once, meaning that your read will overlap at most one write.)

```
1   class AcmeRegister implements Register {

2     // N is the total number of threads
3     // Atomic multi-reader single-writer register
4     private BoolRegister[] b = new BoolMRSWRegister[3 * N];

5     public void write(int x) {
6        boolean[] v = intToBooleanArray(x);

7        // copy v[i] to b[i] in ascending order of i
8        for (int i = 0; i < N; i++)
9           b[i].write(v[i]);

10       // copy v[i] to b[N+i] in ascending order of i
11       for (int i = 0; i < N; i++)
12          b[N+i].write(v[i]);

13       // copy v[i] to b[2N+i] in ascending order of i
14       for (int i = 0; i < N; i++)
15          b[(2*N)+i].write(v[i]);
16    }

17    public int read() {
18       // missing code
19    }
20  }
```

Problem 3

The AtomicInteger class (in the java.util.concurrent.atomic package) is a container for an integer value. One of its methods is boolean compareAndSet(int expect, int update). This method compares the objects current value to expect. If the values are equal, then it atomically replaces the objects value with update and returns true. Otherwise, it leaves the objects value unchanged, and returns false. This class also provides int get() which returns the objects actual value. Consider the FIFO queue implementation shown in Fig. 3.15. It stores its items in an array items, which, for simplicity, we will assume has unbounded size. It has two AtomicInteger fields: tail is the index of the next slot from which to remove an item, and head is the index of the next slot in which to place an item. Give an example showing that this implementation is not linearizable.

```
1   class IQueue<T> {
2       AtomicInteger head = new AtomicInteger(0);
3       AtomicInteger tail = new AtomicInteger(0);
4       T[] items = (T[]) new Object[Integer.MAX_VALUE];
5       public void enq(T x) {
6         int slot;
7         do {
8             slot = tail.get();
9         } while (! tail.compareAndSet(slot, slot+1));
10        items[slot] = x;
11      }
12      public T deq() throws EmptyException {
13        T value;
14        int slot;
15        do{
16           slot = head.get();
17           value = items[slot];
18           if (value == null)
19           throw new EmptyException();
20       } while (! head.compareAndSet(slot, slot+1));
21      return value;
22      }
23  }
```