Homework 3

Ian Ker-Seymer

March 2015

Instructions for running

1. `cd` in to directory
2. Run the tests: `make test`

Problem 1

Below are some tables of the results of running the various concurrent list implementations found in the `ProblemOne/` folder. Each question slowly reduces the granularity of the locking in order to maximized the time the threads run in parallel.

Table 1: Number of threads: 8, Array size: 200

Method	CoarseList	FineGrainList	LazierList	LazyList
Add	6ms	59ms	14ms	16ms
Remove	27ms	20ms	31ms	29ms
Contains	12ms	86ms	8ms	16ms

Table 2: Number of threads: 16, Array size: 200

Method	CoarseList	FineGrainList	LazierList	LazyList
Add	8ms	152ms	48ms	62ms
Remove	60ms	54ms	71ms	4ms
Contains	27ms	188ms	48ms	19ms

Table 3: Number of threads: 32, Array size: 200

Method	CoarseList	FineGrainList	LazierList	LazyList
Add	40ms	186ms	193ms	29ms
Remove	135ms	198ms	57ms	11ms
Contains	92ms	253ms	46ms	68ms

Table 4: Number of threads: 8, Array size: 2000

Method	CoarseList	FineGrainList	LazierList	LazyList
Add	119ms	6323ms	1322ms	223ms
Remove	667ms	1020ms	3222ms	237ms
Contains	2325ms	1430ms	260ms	2532ms

Table 5: Number of threads: 16, List length: 2000

Method	CoarseList	FineGrainList	LazierList	LazyList
Add	341ms	1479ms	495ms	638ms
Remove	1175ms	3998ms	673ms	602ms
Contains	3078ms	2398ms	3151ms	466ms

Table 6: Number of threads: 32, Array size: 2000

Method	CoarseList	FineGrainList	LazierList	LazyList
Add	2306ms	8903ms	846ms	1850ms
Remove	14990ms	8849ms	15075ms	1251ms
Contains	7615ms	22043ms	3202ms	3251ms

Coarse list often (counter-intuitively) performs better than FineGrainList despite having more of its execution time spent in locks. This is more of a practical concern and is due to the overhead that is required to obtain and release locks. With small arrays and small number of threads, there is not enough gained by adding parallelism when you take into account the overhead caused by dealing with locks.

LazierList begins to shine when adding to a new list, however, it suffers tremendously when attempting to remove values because it must a) traverse more values and b) occasionally loops through the list to remove all of the marked nodes.

Testing the list

In order to test the implementations, you could create some fairly complicated test cases which see to exploit all of the edge cases you can imagine, with as many threads as possible. Then, you can compare that list index by index with a list that was created purely sequentially. By creating tests cases with contention amongst threads, hopefully you can find a situation which 'breaks' the implementation.

However, this is problematical in general. The number of possible permutations with respect to list manipulation is nearly infinite. You simply cannot test every case; and thus is fails as a method for proving correctness.

A better method is using a formal proof to show that the list is linearizable, then use test cases to ensure you don't accidentally write bugs in your implementation.

Problem 2

The implementation relies on the state of multiple registers to determine if a register is currently being written to. During the read phase, reads the 3 separate indices (from 0, N -> N, 2N -> 2N, 3N). It then checks whether whether 1N and 2N are equal. If they are, that means it is not being overwritten and we can return 2N. Otherwise, 0N represents the correct state as because N1 can't be written before N1.

Problem 3

A linearizable object is an object in which a single linearization point can be found. In the IQueue example, there is not a single point where "the effects of the method call become visible to other method calls." In order for something to be considered enqueued, two separate instructions are required:

```
tail.compareAndSet(slot, slot+1);
items[slot] = x;
```

Because of this, the List can enter an inconsistent state from the viewpoint of other methods. This is best exemplified with an example.

Imagine you have N threads which call enqueue. All of the threads manage to call `tail.compareAndSet(slot, slot+1)`; however, only N-1 threads call `items[slot] = x`. At this point, one slot is filled with `null` while the others have some value

1	8	*null*	9	2

Now, imagine that `dequeue()` is called twice before `items[slot] - x` changes the `null` value.

null	9	2

If `dequeue()` were called again, it would return an `EmptyException()`, despite the queue containing values. This is the type of inconsistency that can happen without linearization guarantees.