Multiprocessor Programming Course

Dario Bahena Tapia - Ricardo Zavaleta Vazquez - Isai Barajas Cicourel
November 2015

1 LockFreeQueueTest

1.1 Particular Case (problem)

This is a particular case of the mutual exclusion problem, where the shared resource is a queue.

1.2 Solution

In order to guarantee the correctness of the multi-threaded access to the queue, it implements a lock free scheme on its *enq* and *deq* methods by putting waits before modifying the state of the queue. It does it incorrectly though, as we will see later.

```
1   class LockFreeQueue {
2     public int head = 0;      // next item to dequeue
3     public int tail = 0;      // next empty slot
4     Object[] items; // queue contents
5     public LockFreeQueue(int capacity) {
6       head = 0;  tail = 0;
7       items = new Object[capacity];
8     }
9
10   public void enq(Object x) {
11       // spin while full
12       while (tail - head == items.length) {}; // spin
13       items[tail % items.length] = x;
14       tail++;
15     }
16
17     public Object deq() {
18       // spin while empty
19       while (tail == head) {}; // spin
20       Object x = items[head % items.length];
21       head++;
22       return x;
23     }
24   }
```

1.3 Experiment Description

The test program LockFreeQueueTest includes the following individual test cases; the parallel degree is two threads for each operation (queue or dequeue), with a TEST_SIZE of 512 (number of items to enqueue and dequeue) which is spread evenly among the two threads on each group:

- *testSequential*: calls *enq* method as many times as TEST_SIZE, and later calls *deq* method the same number of times checking that the FIFO order is preserved.

- *testParallelEnq*: enqueues in parallel (two threads) but dequeues sequentially.

- *testParallelDeq*: enqueues sequentially but dequeues in parallel (two threads)

- *testParallelBoth*: enqueues and dequeues in parallel (with a total of four threads, two for each operation).

The test program was run on two types of machines; one with two cores only and another with 24 cores.

1.4 Observations and Interpretations

The bottom line of this exercise is most likely, to show several types of problems that can occur if we do not use mutual exclusion; the queue implementation fails implementing it, making the test vulnerable to several cases of race conditions. Below we explain some of them.

1.4.1 Non atomic dequeue: referencing invalid registers

The symptom for this problem is a NullPointerException, and it occurred on both test machines:

```
1) testParallelEnq(mutex.LockFreeQueueTest)java.lang.NullPointerException
at mutex.LockFreeQueueTest.testParallelEnq(LockFreeQueueTest.java:67)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
```

where the offending line is a cast to an Integer from the value returned by the *deq* method; this implies that such function is returning *null*. A possible scenario to produce this outcome is as follows. Prefixes of T1 or T2 indicate the thread running the action, and they refer to the line numbers of the code of method *deq* posted above.

- Assume that queue holds a single element which lives in *items* array at position 0; the array has *null* on position 1 (per initialization).

- T1: Executes method up to line 19.

- T2: Executes method up to line 19.

- T1: Executes lines 20, getting *items[0]*.

- T1: Executes lines 21, increasing *head* to 1.

- T2: Executes line 20, but as *head* was changed already, it gets *items[1]*.

- T2: Executes lines 21, increasing *head* to 2.

- T1: Executes line 22 returning *items[0]*

- T2: Executes line 22 returning *items[1]*, which was *null*.

The problem with above interlacing derives from the fact that the *deq* method is not an atomic operation, hence it allowed both threads to enter into the critical section and compete for updating the shared variables.

1.4.2 Non atomic dequeue: returning duplicate values

The symptom for this problem is a duplicate pop warning, and it occurred on both test machines:

```
.parallel deq
Exception in thread "Thread-5" junit.framework.AssertionFailedError: DeqThread: duplicate pop
at junit.framework.Assert.fail(Assert.java:57)
at junit.framework.TestCase.fail(TestCase.java:227)
at mutex.LockFreeQueueTest$DeqThread.run(LockFreeQueueTest.java:129)
```

where the offending line is an assertion that validates that nobody else has pop such value from the queue; as the threads which populate do have non overlapping ranges of values, the pop operations (*deq*) shall never return a duplicate one. But duplication is possible indeed, if we have a sequence like the one below between the two threads:

- Assume that queue holds a single element which lives in *items* array at position 0.

- T1: Executes method up to line 19.

- T2: Executes method up to line 19.

- T1: Executes line 20, getting *items[0]*.

- T2: Executes line 20, getting as well *items[0]*.

- T1: Executes line 21, setting *head* to 1.

- T2: Executes line 21, setting *head* to 2.

- T1: Executes line 22 returning *items[0]*.

- T2: Executes line 22 returning *items[0]*.

Not only we left the queue in an inconsistent state (head has incorrect value), but we also returned the same element twice, triggering then the violation on the test. The underlying problem is the same as previous case: lack of atomicity of the *deq* method.

1.4.3 Non atomic auto-increment: loosing values

The symptom for this problem is a never ending program, hanging on either the test *testParallelBoth* or *testParallelEnq*; this issue occurred on both test machines (though it was easier to reproduce on the one with two cores). When produced several thread dumps of the Java program, we can see either two hanging threads (*testParallelBoth* case):

```
"Thread-7" #16 prio=5 os_prio=0 tid=0x00007f3140102000 nid=0x3f51
          runnable [0x00007f31226e9000]
   java.lang.Thread.State: RUNNABLE
        at mutex.LockFreeQueue.deq(LockFreeQueue.java:33)
        at mutex.LockFreeQueueTest$DeqThread.run(LockFreeQueueTest.java:141)

"main" #1 prio=5 os_prio=0 tid=0x00007f3140009800 nid=0x3f3c in Object.wait()
         [0x00007f3148382000]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x00000000d6effea8> (a mutex.LockFreeQueueTest$DeqThread)
        at java.lang.Thread.join(Thread.java:1245)
        - locked <0x00000000d6effea8> (a mutex.LockFreeQueueTest$DeqThread)
        at java.lang.Thread.join(Thread.java:1319)
        at mutex.LockFreeQueueTest.testParallelBoth(LockFreeQueueTest.java:123)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:497)
        at junit.framework.TestCase.runTest(TestCase.java:164)
        at junit.framework.TestCase.runBare(TestCase.java:130)
```

The main thread is waiting for a dequeue thread to finish; but that one
is on an infinite loop at line 19 of *deq* method (line numbers per our listing
in this document, not in the file). This means that we have lost some of the
inserted elements in queue, and that one of the dequeue threads will never
finish; as they expect each to pop a fixed amount of elements per the test.

The other hanging scenario that comes out of (*testParallelEnq*), is a single
hanging thread, which was actually the main thread doing a serial dequeue.
The common factor for both scenarios was a parallel enqueue.

But the amount of times we request a dequeue operation, among all
threads, is the same as the number of elements we queued; how come we
end up loosing some of those? One possible explanation is again, the lack
of atomicity but this time of the *enq* method; to be more specific, the lack
of atomicity of its auto-increment operation *tail++* (line 14). Let us re-
member that the auto-increment operator in Java is nothing but syntactic
sugar for the following sequence of operations (when applied to *tail* variable):

```
tmp = tail;
tmp = tmp + 1;
tail = tmp;
```

5

If two threads execute the lower level operations above, we can see how they can end up loosing increments in the shared variable *tail*; the following sequence is on example of such scenario:

- T1: executes $tmp = tail$.

- T2: executes $tmp = tail$.

- T1: executes $tmp = tmp + 1$.

- T2: executes $tmp = tmp + 1$.

- T1: executes $tail = tmp$.

- T2: executes $tail = tmp$.

We can appreciate that in the above interlacing, the final value of the shared variable is $tail + 1$; instead of the expected value of $tail + 2$. It would be enough to loose a single value this way, in order to make the enqueue threads think that they inserted the total of 512, while they really inserted 511; as each dequeue thread will try to pop 256 each, only one of them will be able to finish while the other will get blocked after having removed 255 entries. That is most likely the explanation for the hung threads we pasted above [1]; the solution is again to really implement mutual exclusion around the methods *deq* / *enq*; in such a way that they become atomic operations.

1.5   Proposed changes to fix the problems

In class it was mentioned that the *head* and *tail* variables were not declared as *volatile*; which in Java jargon means they are susceptible to caching on each core. This means, that a write to any of those two variables (which control the queue size) are not meant to be reflected immediately to the other cores, unless we declare the variable as *volatile*. We did several attempts to achive lock-free implementation of this queue, by making the variables *volatile* but it was enough; then we tried by making atomic only portions of the *enq* and *deq* methods; but it did not work either.

---

[1]Note that it does not matter that the array *items* has populated all the correct entries, because the flow is controlled by the counters *tail* and *head*.

All the three scenarios described before can be eliminated, if we make the methods *enq* and *deq* atomic; this can be easily achieved in Java by making them *synchronized*. However, by doing that, we will loose parallelism among the two groups of threads (those calling *enq* and those calling *deq*); this is because the *synchronized* keyword uses as lock the whole object, so at any moment in time, only one synchronized method can actually run within any object. In order to overcome this limitation, we can use synchronized blocks against two different lock objects (one for each operation).

Even with the changes above, we can still have issues; what if the very first thread running is one calling *deq* method? It will find the queue empty and loop forever. In order to prevent that, we should remove the waiting operation out of the queue methods, and put them in the test code itself. This is because, on the cited scenario that we try to dequeue with an empty queue, we would expect to simply try again (giving the chance to parallel enqueue threads to produce something for us to pop). The final code which incorporates these fixes is listed below:

```java
class LockFreeQueue {
  private static Object enqLock = new Object();
  private static Object deqLock = new Object();

  public int head = 0;    // next item to dequeue
  public int tail = 0;    // next empty slot
  Object[] items; // queue contents
  public LockFreeQueue(int capacity) {
    head = 0; tail = 0;
    items = new Object[capacity];
  }

  public  boolean enq(Object x) {
     synchronized(enqLock)
        {
            if (tail - head == items.length) {
                return false;
            }
            items[tail % items.length] = x;
            tail++;
            return true;
        }
  }
```

```java
public Object deq() {
    synchronized(deqLock)
    {
        if (tail == head) {
            return null;
        }
        Object x = items[head % items.length];
        head++;
        return x;
    }
}
}
```

The test code was also modified, to make the enqueue and dequeue threads to iterate until they have successfully called their respective methods 256 times (total size of queue divided by number of threads). A successful call is one that does not return *false* nor *null*. The modified code was executed ten thousand times and it did not produce any of the original problems we explained. Though not a formal proof of its correctness, it is a good indication of the same (the original program produced one of the cited problems quite often, usually within 10 executions).

A probably more elegant solution we came out with, after reading about Java facilities for locks and conditions, was the following; it has the advantage that it does not require us to change the test case:

```java
class LockFreeQueue {
  int head = 0;    // next item to dequeue
  int tail = 0;    // next empty slot
  Object[] items;  // queue contents

  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();

  public LockFreeQueue(int capacity) {
    head = 0;  tail = 0;
    items = new Object[capacity];
  }

  public void enq(Object x) {
    lock.lock();
    try {
      // spin while full
      while (tail - head == items.length)
        notFull.await();
      items[tail % items.length] = x;
      tail++;
      notEmpty.signalAll();
    }
    catch(InterruptedException e) {
      throw new RuntimeException(e);
    }
    finally {
      lock.unlock();
    }
  }

  public Object deq() {
    lock.lock();
    try {
      // spin while empty
      while (tail == head)
        notEmpty.await();
      Object x = items[head % items.length];
      head++;
      notFull.signalAll();
      return x;
    }
    catch(InterruptedException e) {
      throw new RuntimeException(e);
```

```
    }
    finally {
      lock.unlock();
    }
  }
}
```

This second option is interesting on its usage of the *await* call to the condition objects; such call provokes that a thread which got granted a lock releases it to give a change to other threads to come in. For example, an enqueuer may give a chance to a dequeuer to make room for it; or, a dequeuer may give an enqueuer a chance to produce something for it. Without those calls to *await*, we could end up falling into never ending loops.

Another relevant detail of this second solution is the usage of *signalAll* method on conditions; this is to awake the threads which were waiting on that particular condition. In our case, it corresponds to the enqueuers or dequeuers which called *await* on their respective *notFull* or *notEmpty* conditions. Thus, as soon as there is room an enqueuer is awakened and as soon as there is data a dequeuer is awakened. As there are tests where we can have more than one thread on each condition, we prefer to use *signalAll* method rather than *signal* (which will awake a single thread). This is to prevent the lost awakes problem that is mentioned on the book (on the chapter about monitors, if we recall correctly).

2 AtomicMRMWRegisterTest

2.1 Particular Case (problem)

The problem we are trying to solve here, is that of implementing an atomic multi-valued [2] multiple-reader multiple-writer register. Let us briefly recall that atomic registers are the most powerful ones, when compated with the other two categories: safe and regular. Informally, atomic registers behave like one would expect when reads overlap writes: the reads always "see" the last written value; while regular registers allows to see either previous or latest value. Finally "safe" registers allow reads to see any value (not safe at all then!).

2.2 Solution

The solution from the book uses as primitive blocks the atomic multi-valued multi-reader single-writer registers; we build an array of them whose size equals the number of threads we want to support (assuming number of writers is same as readers). When a given thread A wants to write a value, it needs to read all the values on the array and writes to its own cell an stamped value that is bigger than any one observed. When same thread wants to read a value, it reads again whole array and returns the one with biggest stamped value (resolving ties with thread ids). This is essentially the same as Lamport's Bakery algorithm from mutual exclusion chapter; the code is small enough to fit here, so there it goes for reference:

---

[2]As opposed to boolean values, which have only a couple of values; while multi-valued can range over a big set, like integers.

```
1  public class AtomicMRMWRegister<T> implements Register<T>{
2    // array of multi−reader single−writer registers
3    private StampedValue<T>[] a_table;
4    public AtomicMRMWRegister(int capacity, T init) {
5      a_table = (StampedValue<T>[]) new StampedValue[capacity];
6      StampedValue<T> value = new StampedValue<T>(init);
7      for (int j = 0; j < a_table.length; j++) {
8        a_table[j] = value;
9      }
10   }
11   public void write(T value) {
12     int me = ThreadID.get();
13     StampedValue<T> max = StampedValue.MIN_VALUE;
14     for (int i = 0; i < a_table.length; i++) {
15       max = StampedValue.max(max, a_table[i]);
16     }
17     a_table[me] = new StampedValue<T>(max.stamp + 1, value);
18   }
19   public T read() {
20     StampedValue<T> max = StampedValue.MIN_VALUE;
21     for (int i = 0; i < a_table.length; i++) {
22       max = StampedValue.max(max, a_table[i]);
23     }
24     return max.value;
25   }
26 }
```

2.3   Experiment Description

The test program AtomicMRMWRegisterTest includes the following individual test cases:

- *testSequential*: calls *write* method first with a value of 11, then calls *read* method expecting read 11. A single thread is used (main thread).

- *testParallel*: calls twice method *write*, putting first 11 then 22 value. Then proceeds to create 8 reader threads, which expect all to read 22. This test is not really exercising multi-write capability of the register. sequentially.

2.4 Observations and Interpretations

The test performs well on 2 and 24 core machines, without any race conditions nor abnormal behaviors. This partly because, the test is not really exercising the multi-write capacility, nor the interlacing of writes and reads. We believe the authors just copied paste a previous test (probably one that just cared about single reader and multiple writers), and forgot to update. Due time constraints we did not modify the test to exercise those missing features; a sample execution is shown below:

```
$ junit register.AtomicMRMWRegisterTest
.sequential read and write
.parallel read

Time: 0.004

OK (2 tests)
```

3 RegMRSWRegisterTest

3.1 Particular Case (problem)

The problem we face here is that of building a regular multi-valued multiple-reader single-writer register; out of simpler constructions, like the rest of chapter 4.

3.2 Solution

The solution, though supporting multi-values (integers), is bounded in the range of values. We use as primitive blocks regular boolean multiple-reader single-writer registers; and create an array of them (as many as values in supported range), and use it to represent numbers in unary way. That is, each position on the array that is set to true represents the number corresponding to that position (the index itself is the number).

Initially the boolean array is initialized to zero value, indicated by having true the cell at index 0. The write method of value x sets to true array position x, and updates to false the lower value positions (so it updates from right to left). In order to guarantee the regular property, the read method operates on the opposite direction: it goes from left to right, starting at zero position, and returns the first position whose value is true (an invariant of the array should be that there is at least one cell in with true value). The code from the book is presented below, for further reference (the range of values picked by the book's authors was that of byte Java type):

```
1   public class RegMRSWRegister implements Register<Byte> {
2     private static int RANGE = Byte.MAX_VALUE - Byte.MIN_VALUE + 1;
3     // regular boolean mult-reader single-writer register
4     boolean[] r_bit = new boolean[RANGE];
5     public RegMRSWRegister(int capacity) {
6       r_bit[0] = true; // least value
7     }
8     public void write(Byte x) {
9       r_bit[x] = true;
10      for (int i = x - 1; i >= 0; i--)
11        r_bit[i] = false;
12    }
13    public Byte read() {
```

```
14        for (int i = 0; i < RANGE; i++)
15          if (r_bit[i]) {
16            return (byte)i;
17          }
18        return -1;  // never reached
19      }
20  }
```

3.3   Experiment Description

The test program *RegMRSWRegisterTest* includes the following individual
test cases (which is pretty much the same thing as test *AtomicMRMWReg-isterTest*):

- *testSequential*: calls *write* method first with a value of 11, then calls
  read method expecting read 11. A single thread is used (main thread).

- *testParallel*: calls twice method *write*, putting first 11 then 22 value.
  Then proceeds to create 8 reader threads, which expect all to read 22.

3.4   Observations and Interpretations

Just like *AtomicMRMWRegisterTest*, this does not exhibit any issue on two
nor in 24 core machines. Again, part of that reason is that the test is not
really interlacing writes with reads. A sample successful execution is shown
below:

```
$ junit register.RegMRSWRegisterTest
.sequential read and write
.parallel read

Time: 0.004

OK (2 tests)
```

4 SimpleSnapshotTest

4.1 Particular Case (problem)

The problem we want to solve is that of atomic snapshots: we want to read atomically a set of registers. For this problem we assume the registers are themselves atomic, and that each one supports multiple-readers but just one writer. The problem is better defined in terms of the Java interface we want to implement:

```
public interface Snapshot<T> {
  public void update(T v);
  public T[] scan();
}
```

While the method *update* allows each thread to modify its own register (single-writer), the *scan* method allows any thread to read all the registers as a single atomic operation (multiple-readers). Ideally, we would like to have a wait-free implementation of these two methods.

4.2 Solution

The solution presented in the book is a natural evolution to the sequential implementation (which uses *synchronized* methods for both *update* and *scan* methods). The solution is called *SimpleSnapshot* and it uses stamped values for the *update* method; there is no need to seek for the maximum, just to increase the current stamp per thread (each thread only writes to its own register). This makes the *update* method wait-free, which is the ideal (it was not very hard to achieve indeed, given the single-writer restriction).

For the *scan* method we call an auxiliary function *collect*, which represents a non atomic read of all the registers (which are copied into a new array and returned); if two consecutive *collect* calls return same values, it means that between those two calls there were no writes. When we reach that condition, we return the array of values; otherwise we repeat the iteration. Please note that *scan* method is not wait-free (we do not know how many times we will need to iterate), but at least is obstruction-free (we are not blocking other

threads from trying their own *scan* calls).

The main parts of the code are presented below for reference:

```java
1  public class SimpleSnapshot<T> implements Snapshot<T> {
2     // array of atomic MRSW registers
3     private StampedValue<T>[] a_table;
4     ...
5     public void update(T value) {
6       int me = ThreadID.get();
7       StampedValue<T> oldValue = a_table[me];
8       StampedValue<T> newValue =
9           new StampedValue<T>((oldValue.stamp)+1, value);
10      a_table[me] = newValue;
11    }
12    private StampedValue<T>[] collect() {
13      StampedValue<T>[] copy = (StampedValue<T>[]) new StampedValue[a_table.length];
14      for (int j = 0; j < a_table.length; j++)
15        copy[j] = a_table[j];
16      return copy;
17    }
18    public T[] scan() {
19      StampedValue<T>[] oldCopy, newCopy;
20      oldCopy = collect();
21      collect: while (true) {
22        newCopy = collect();
23        if (! Arrays.equals(oldCopy, newCopy)) {
24          oldCopy = newCopy;
25          continue collect;
26        }
27        // clean collect
28        T[] result = (T[]) new Object[a_table.length];
29        for (int j = 0; j < a_table.length; j++)
30          result[j] = newCopy[j].value;
31        return result;
32      }
33    }
34  }
```

4.3   Experiment Description

The test program *SimpleSnapshotTest* consists of two individual test cases:

- *testSequential*: with a single thread we update its value first (11), and then obtain an snapshot (*scan*); expectation is to have a single value in array (the one we wrote).

- *testParallel*: we create a couple of threads, and each one writes its own register twice (first putting 11, then 22). Then each thread proceeds to call *scan* method and saves its own returned values into a global results table. At the end of test, main thread check that both threads got the same results (which should be a two element array, both with value 22).

4.4   Observations and Interpretations

The test runs fine, both in 2 and 24 core machines; sample output below:

```
$ junit register.SimpleSnapshotTest
.sequential
.parallel

Time: 0.002

OK (2 tests)
```

5 QueueTest

5.1 Particular Case (problem)

The problem we want to solve is again mutual exclusion around a shared bounded queue; just like chapter 2. The difference here is that, instead of just using locks the authors try to explain the facilities provided by Java *Condition* interface (part of package *java.util.concurrent.locks*). Such interface, along with the *Lock* one, offer a finer grain control over the Monitors [3] capabilities offered by Java language; when compared to merely using the *synchronized* feature.

5.2 Solution

The solution to the mutual exclusion problem around the bounded queue is solved by using a single Lock of type *ReentrantLock*, to control exclusive access to the data structure (reentrant is important to ensure that a thread which has gotten the lock already, can request it again as many times as it want). In addition, the solution uses a couple of *Condition* objects to represent the conditions which enqueuers and dequeuers wait on:

- *notFull*: condition enqueuers wait on, prior daring to push.

- *notEmpty*: condition dequeuers wait on, prior daring to pop.

The implementations of methods *enq* and *deq* are similar to those from the flawed *LockFreeQueue* that we analyzed before; there are a couple of crucial differences though:

- We use a lock to implement mutual exclusion, so we eliminate all the issues seen before due race conditions.

- The potentially infinite loops for both *enq* and *deq* now do something: they call the *await* method on their respective conditions. This guarantees that we do not have deadlocks: if a thread does not find the

---

[3]Monitors, as explained in the book, are an structured way to integrate synchronization with an object's data and methods (speaking about OO paradigm).

queue on the expected state to perform the operation, it releases the lock and allow the others to proceed. In that way we allow enqueuers to put the data we are waiting for, or the dequeuers to make the space we need; after which they "wake-up" the waiting threads with a call to condition's *signal* method.

The code looks quite similar to our second proposed modification of previous exercise *LockFreeQueueTest*; and actually, it also looks quite similar to the sample code listed on the Javadoc API of *Condition* interface (on which we also based our proposal). Anyway, the book solution is listed below for further reference:

```
1   class Queue<T> {
2       final Lock lock = new ReentrantLock();
3       final Condition notFull  = lock.newCondition();
4       final Condition notEmpty = lock.newCondition();
5       final T[] items;
6       int tail, head, count;
7       public Queue(int capacity) {
8           items = (T[])new Object[capacity];
9       }
10      public void enq(T x) throws InterruptedException {
11          lock.lock();
12          try {
13              while (count == items.length)
14                  notFull.await();
15              items[tail] = x;
16              if (++tail == items.length)
17                  tail = 0;
18              ++count;
19              notEmpty.signal();
20          } finally {
21              lock.unlock();
22          }
23      }
24      public T deq() throws InterruptedException {
25          lock.lock();
26          try {
27              while (count == 0)
28                  notEmpty.await();
29              T x = items[head];
30              if (++head == items.length)
```

```
31              head = 0;
32          −−count ;
33          notFull . signal ();
34          return x;
35       } finally {
36          lock . unlock ();
37       }
38    }
39  }
```

5.3   Experiment Description

The test is pretty much the same than *LockFreeQueueTest*, except for the number of threads (8 instead of 2) and data (64 instead of 512).

5.4   Observations and Interpretations

Although the test does not exhibit any issue on 2 nor in 24 cores, we would feel safer replacing the *signal* calls by *signalAll*, just to totally discard the potencial problem of lost wake-ups explained on the book (this is what we did for our second proposal to fix *LockFreeQueueTest*, indeed). A sample successful execution is shown below:

```
$ junit monitor.QueueTest
.sequential push and pop
.parallel both
.parallel deq
.parallel enq

Time: 0.009

OK (4 tests)
```

6 LazyListTest

6.1 Particular Case (problem)

The problem is that of gradually improving what coarse grain locking offers for concurrent data structures like sets (implemented with linked lists).

6.2 Solution

The *LazyList* solution is a refinement of the *OptimisticList* solution which does not lock while searching, but locks one it finds the interesting nodes (and then confirms that the locked nodes are correct). As one drawback of *OptimisticList* is that the most common method *contains* method locks, the next logical improvement is to make this method wait-free while keeping the *add* and *remove* methods locking (but reducing their transversings of the list from two to just one).

The refinement mentioned above is precisely that of *LazyList*, which adds a new bit to each node to indicate whether they still belong to the set or not (this prevents transvering the list to detect if the node is reachable, as the new bit introduces such invariant: if a transversing thread does not find a node or it is marked in this bit, then the corresponding item does not belong to the set. This behaviour implies that *contains* method does a single wait-free transversal of the list.

For adding an element to the list, *add* method traverses the list, locks the target's predecessor and successor nodes, and inserts the new node in between. The *remove* method is lazy (hece the name of the solution), as it splits its task in two parts: first marks the node in the new bit, logically removing it; and second, update its predecessor's next field, physically removing it.

The three methods ignore the locks while transversing the list, possibly passing over both logically and physically deleted nodes. The *add* and *remove* methods still lock *pred* and *curr* nodes as with *OptimisticList* solution, but the validation reduces to check that *curr* node has not been marked; as well as validating the same for *pred* node, and that it still points to *curr* (validating a couple of nodes is much better than transversing whole list

though). Finally, the introduction of logical removals implies a new contract
for detecting that an item still belongs to set: it does so, if still referred by
an unmarked reachable node.

The most relevant methods, *remove* and *contains*, are listed below:

```
1    public boolean remove(T item) {
2      int key = item.hashCode();
3      while (true) {
4        Node pred = this.head;
5        Node curr = head.next;
6        while (curr.key < key) {
7          pred = curr; curr = curr.next;
8        }
9        pred.lock();
10       try {
11         curr.lock();
12         try {
13           if (validate(pred, curr)) {
14             if (curr.key != key) {      // present
15               return false;
16             } else {                    // absent
17               curr.marked = true;       // logically remove
18               pred.next = curr.next;    // physically remove
19               return true;
20             }
21           }
22         } finally {                     // always unlock curr
23           curr.unlock();
24         }
25       } finally {                       // always unlock pred
26         pred.unlock();
27       }
28     }
29   }
30
31   public boolean contains(T item) {
32     int key = item.hashCode();
33     Node curr = this.head;
34     while (curr.key < key)
35       curr = curr.next;
36     return curr.key == key && !curr.marked;
37   }
```

6.3 Experiment Description

The test is pretty much the same described for *LockFreeQueueTest*, with a few differences.

- The data structure here is a set, rather than a queue.

- The exceptiont that it uses 8 threads instead of two for each operation (*add* / *remove*).

- The threads that remove elements do not care only in successfully removing certain number of times (like with the queue); here they expect to remove a particular subset of the values.

6.4 Observations and Interpretations

The test works as expected on a two cores machine, sample output below:

```
.parallel deq
.parallel both
.sequential push and pop
.parallel enq

Time: 0.03

OK (4 tests)
```

Interestingly, on a 24 cores machine, sometimes the test case *testParallelBoth* fails with exceptions like the one below:

```
junit.framework.AssertionFailedError: RemoveThread: duplicate remove
at junit.framework.Assert.fail(Assert.java:57)
at junit.framework.TestCase.fail(TestCase.java:227)
at lists.LazyListTest$RemoveThread.run(LazyListTest.java:142)
```

While debugging the error above, we found that the message of "duplicate remove" is a bit misleading; is not really that someone else tried to delete that value (as each *RemoveThread* cares about a unique set of values). The

real problem is that the removing threads just try once to remove each value, and fail if they did not find any of them. Since both the adder and remover threads are started concurrently, there is no guarantee that the adders will come first than the removers; so it could be that the removers try to pull out something that has not been inserted yet (leaving to the exception shown above).

Since the *LazyList* solution does not include an error-and-retry approach (as with our second rewrite of the *LockFreeQueue*, which used Java condition's await methods), the only way to fix this would be to rewrite the test program itself. Each remover thread will need to indefinitely try to remove all its items until completion, rather than expecting that all of them are available by the time they are to be removed. We tried that approach and made the proper adjustments to the test program, after which the problem got solved as expected.

7   BoundedQueueTest

7.1   Particular Case (problem)

The problem is to implement a bounded partial queue (that is, one which
finite capacity which allows waits inside its methods).

7.2   Solution

The solution proposes to allow concurrency among enqueuers and dequeuers,
by using two different locks (one for each group of threads). The code is small
enough to be placed and explained in a bit more detail here, so let us give it
a try:

```
1     public void enq(T x) {
2        if (x == null) throw new NullPointerException();
3        boolean mustWakeDequeuers = false;
4        enqLock.lock();
5        try {
6           while (size.get() == capacity) {
7              try {
8                 notFullCondition.await();
9              } catch (InterruptedException e) {}
10          }
11          Entry e = new Entry(x);
12          tail.next = e;
13          tail = e;
14          if (size.getAndIncrement() == 0) {
15             mustWakeDequeuers = true;
16          }
17       } finally {
18          enqLock.unlock();
19       }
20       if (mustWakeDequeuers) {
21          deqLock.lock();
22          try {
23             notEmptyCondition.signalAll();
24          } finally {
25             deqLock.unlock();
26          }
27       }
28    }
```

The main points of algorithm above are as follows (the code above was previously corrected, as it had several things inverted):

- It does not allow null values.

- There is a lock dedicated to *enq* method, which guarantees mutual exclusion to the critical section for enqueuers.

- It spins while the queue is full (the try/catch block was added just to allow compilation). This line was incorrectly asking whether queue was empty, while it needs to ask whether is still full.

- Once we know queue is room, we allocate the new node and update *tail* pointer.

- We atomically get and increment the atomic counter for the size (this was previously a decrement, which was incorrect).

- If the atomically retrieved previous value of size was zero, it means the queue was empty and there may had been waiting dequeuers; therefore we acquire the lock for *deq* method and inform all dequeuers that the queue has not at least an item.

The *deq* method is symmetric so we do not repeat same explanation (but worth to say that it had same inverted conditions and actions than *enq* method, in the code; so we needed to apply same corrective actions ... perhaps that was the purpose of the exercise?).

7.3   Experiment Description

The test is pretty much the same described for *LockFreeQueueTest.* (with the exceptiont that it uses 8 threads instead of two).

7.4   Observations and Interpretations

The original code for the test made no sense, as conditions and actions for *enq* and *deq* methods were inverted; we did not even care testing such strange version that did not match at all the one published on the book (we also needed to fix the initialization of the *size* variable, which shall be zero instead

of *capacity*). After the corrections mentioned on the Solution section above, the test ran normally. Sample output below:

```
.parallel both
.sequential push and pop
.parallel enq
.parallel deq

Time: 0.028

OK (4 tests)
```

Even if the original version of this test actually ran (we did not test it), it turns out quite counter-intuitive. The modified code worked fine on both 2 and 24 core machines.

8 LockFreeQueueRecycleTest

8.1 Particular Case (problem)

From the generic problem of improving coarse grained lock approaches, the particular approach followed on this exercise corresponds to the extreme case: lock-free data structure. The particular case is for a queue, and it has the additional bonus of recycling memory.

8.2 Solution

The solution is based on an 1996 ACM article from Maged M. Michel and Michael L. Scott, who based on previous publications, suggest a new way of implementing a lock-free queue with recycles. They implement the queue with a single-linked list using *tail* and *head* pointers; where *head* always points to a dummy or sentinel node which is the first in the list, and *tail* points to either the last or second to last node in the list. The algorithm uses "compare and swap" (*CAS*) with modification counters to avoid the ABA problem.

Dequeuers are allowed to free dequeued nodes by ensuring that *tail* does not point to the dequeued node nor to any of its predecessors (that is, dequeued nodes may be safely reused). To obtain consistent values of various pointers the authors relied on sequences of reads that re-check earlier values to ensure they have not changed (these reads are claimed to be simpler than snapshots).

8.3 Experiment Description

The test is pretty much the same described for *LockFreeQueueTest* (with the exceptiont that it uses 8 threads instead of two).

8.4 Observations and Interpretations

The test does not exhibit any pitfall, which suggests that the theory works just fine on the tested machines (we tried the one with 2 and with 24 cores). Sample output below:

```
.parallel enq
.parallel deq
.parallel both
.sequential push and pop

Time: 0.023

OK (4 tests)
```

9 SynchronousDualQueueTest

9.1 Particular Case (problem)

The problem is to reduce the synchronization overhead of a synchronous queue.

9.2 Solution

The solution is given by using what is called a dual data structure; which splits the *enq* and *deq* operations in two parts. When a dequeuer tries to remove an item from the queue, it inserts a reservation object indicating that is waiting for an enqueuer. Later when an enqueuer thread realizes about the reservation, it fulfills the same by depositing an item and setting the reservation's flag. On the same way, an enqueuer thread can make another reservation when it wants to add an item and spin on its reservation's flag.

This solution has some nice properties:

- Waiting threads can spin on a locally cached flag (scalability).

- Ensures fairness in a natural way. Reservations are queued in the order they arrive.

- This data structure is linearizable, since each partial method call can be ordered when it is fulfilled.

9.3 Experiment Description

The test creates 16 threads, grouping them on enqueuers and dequeuers; for each group it divides the worlkoad (512) evenly. Then each thread proceeds to either enqueue or dequeue, as many times as its share of the workload indicated (512/8). There are some assertions to ensure that we do not repeat values (each dequeuer marks an array at the index of the value it got). among them.

9.4   Observations and Interpretations

The test runs normally most of the times, but in some occasions it hangs (in the test machine with two cores is a rare error, but on the one with 24 cores it almost always occurs) . During those cases we can see a null pointer exception, and some never ending enqueuers:

```
.parallel both
Exception in thread "Thread-9" java.lang.NullPointerException
        at queue.SynchronousDualQueueTest$DeqThread.run(SynchronousDualQueueTest.java:67)
2015-10-18 23:20:41
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.60-b23 mixed mode):

"Thread-12" #21 prio=5 os_prio=0 tid=0x00007f4314112800 nid=0x75cf runnable [0x00007f42fc980000]
   java.lang.Thread.State: RUNNABLE
        at queue.SynchronousDualQueue.enq(SynchronousDualQueue.java:49)
        at queue.SynchronousDualQueueTest$EnqThread.run(SynchronousDualQueueTest.java:60)

"Thread-8" #17 prio=5 os_prio=0 tid=0x00007f431410e800 nid=0x75cd runnable [0x00007f42fcb82000]
   java.lang.Thread.State: RUNNABLE
        at queue.SynchronousDualQueue.enq(SynchronousDualQueue.java:49)
        at queue.SynchronousDualQueueTest$EnqThread.run(SynchronousDualQueueTest.java:60)
```

The *NullPointerException* and the hanging enqueuers are related; the former appears cause the *deq* method returns null hence it fails to cast into integer for a dequeuer thread (leaving the overall situation unbalanced, as now nobody will consume some of the items being pushed into the queue).

The queue is populated with integers, so the *null* value returned must be either a defect of the tested class *SynchronousDualQueue*; or that the test case is badly designed. On either case, is shall be related with concurrency, as error occurs much more often on the machine with 24 cores. The easiest way to handle this is to modify the test to handle *null* values on dequeuers thread class:

```
class DeqThread extends Thread {
  public void run() {
    for (int i = 0; i < PER_THREAD; i++) {
      Object v = null;
      while ( (v = instance.deq()) == null );
      int value = (Integer) v;
      if (map[value]) {
        fail("DeqThread: duplicate pop");
      }
      map[value] = true;
    }
  }
}
```

With above modification, the test does not hang anymore (tried in both test machines, with 2 and 24 cores). Given more time, we would have preferred to dig further into root cause of this problem; instead of just putting a patch on the test program.

10 TreeTest

10.1 Particular Case (problem)

This problem belongs to chapter 12, where the idea is to present problems which look inherently serial but that surprisingly accept interesting concurrent solutions (though not always easy to explain). The particular problem this test is exercising is that of shared counting, where we have N threads to increase a shared numeric variable up to certain value; but with the goal of producing less contention than a serial solution would generate.

10.2 Solution

The java program is called *Tree.java*, although it corresponds to the *CombiningTree* from the book, which is a binary tree structure where each thread is located at a leaf and at most two threads can share a leaf. The shared counter is at the root of the tree, and the rest of the nodes serve as intermmediate result points. When a couple of threads collide in their attempt to increment, only one of them will serve as the represtant and go up in the tree with the mission of propagating the combined increment (2) up to the shared counter at the root of the tree. In its way up, it may encounter further threads that collide again with it, making it wait or continue its journey to the root.

When a thread reaches the root it adds its accumulated result to the shared counter, and propagates down the news that the job is done to the rest of the threads that waited along the way. This solution to the counting problem has worse latency than lock-based solutions ($O\left(1\right)$ vs $O\left(log(p)\right)$, where p is the number of threads or processors; but it offers a better throughput.

10.3 Experiment Description

The program consists of a single unit test *testGetAndIncrement*, which creates 8 threads to perform each $2^2 0$ increments. The program uses an auxiliary array called *test* to record the individual increments done by each thread; assertions are made to ensure that none of the attemtps results in a duplicated value, as well as to ensure that all threads completed their task.

34

10.4 Observations and Interpretations

The test performs without much controversy in an elapsed time between 3 and 4 seconds (laptop computer with i5 processor). Below a sample output:

```
.Parallel, 8 threads, 1048576 tries

Time: 3.641
```

To make the test a little more interesting, we created another couple of tests reusing sample template of existing one; difference was the Thread class used on each case.

The first additional test uses a thread class based on Java provider *java.util.concurrent.atomic.AtomicInteger*:

```
cnt2 = new AtomicInteger();

class MyThread2 extends Thread {
  public void run() {
    for (int j = 0; j < TRIES; j++) {
      int i = cnt2.getAndIncrement();
      if (test[i]) {
        System.out.printf("ERROR duplicate value %d\n", i);
      } else {
        test[i] = true;
      }
    }
  }
}
```

while the second additional test was based on a custom class that used synchronized *getAndIncrement* method:

```
class MyThread3 extends Thread {
  public void run() {
    for (int j = 0; j < TRIES; j++) {
      int i = cnt3.getAndIncrement();
      if (test[i]) {
        System.out.printf("ERROR duplicate value %d\n", i);
      } else {
        test[i] = true;
```

```
      }
    }
  }
}

class MyCounter {
  private int cnt = 0;

  public synchronized int getAndIncrement()
  {
    return cnt++;
  }
}
```

The expectation was that the test test based on a synchronized method would be the slowest (*Parallel3* label on output), the one based on the *CombineTree* would become second (*Parallel1* label on output) and the fastest would be the one based on *AtomicInteger* (*Parallel2*); as the latest is likely to take advantage of hardware atomic instruction of 32bits. Surprisingly, the test based on *CombineTree* was the worst of all, by far:

```
.Parallel1, 8 threads, 1048576 tries took 3878
.Parallel2, 8 threads, 1048576 tries took 162
.Parallel3, 8 threads, 1048576 tries took 723

Time: 4.827

OK (3 tests)
```

Most likely we are not comparing apples with apples, as the theory does not match the experiment; perhaps the *CombineTest* class is not meant for real usage, so this comparison is not fair. Anyway, the test does not hang nor fails on neither the 2 cores nor the 24 machines.