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.

```java
class LockFreeQueue {
  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 void enq(Object x) {
      // spin while full
      while (tail - head == items.length) {}; // spin
      items[tail % items.length] = x;
      tail++;
  }

  public Object deq() {
       // spin while empty
       while (tail == head) {}; // spin
       Object x = items[head % items.length];
       head++;
       return x;
  }
}
```

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 remember 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;
```

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:

```
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;
          }
  }
  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:

```
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.

```
public class AtomicMRMWRegister<T> implements Register<T>{
  // array of multi-reader single-writer registers
  private StampedValue<T>[] a_table;
  public AtomicMRMWRegister(int capacity, T init) {
    a_table = (StampedValue<T>[]) new StampedValue[capacity];
    StampedValue<T> value = new StampedValue<T>(init);
    for (int j = 0; j < a_table.length; j++) {
      a_table[j] = value;
    }
  }
  public void write(T value) {
    int me = ThreadID.get();
    StampedValue<T> max = StampedValue.MIN_VALUE;
    for (int i = 0; i < a_table.length; i++) {
      max = StampedValue.max(max, a_table[i]);
    }
    a_table[me] = new StampedValue<T>(max.stamp + 1, value);
  }
  public T read() {
    StampedValue<T> max = StampedValue.MIN_VALUE;
    for (int i = 0; i < a_table.length; i++) {
      max = StampedValue.max(max, a_table[i]);
    }
    return max.value;
  }
}
```

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):

```
public class RegMRSWRegister implements Register<Byte> {
  private static int RANGE = Byte.MAX_VALUE - Byte.MIN_VALUE + 1;
  // regular boolean mult-reader single-writer register
  boolean[] r_bit = new boolean[RANGE];
  public RegMRSWRegister(int capacity) {
    r_bit[0] = true; // least value
  }
  public void write(Byte x) {
    r_bit[x] = true;
    for (int i = x - 1; i >= 0; i--)
      r_bit[i] = false;
  }
  public Byte read() {
    for (int i = 0; i < RANGE; i++)
      if (r_bit[i]) {
        return (byte)i;
      }
    return -1;  // never reached
  }
}
```

3.3 Experiment Description

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

- *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:

```
public class SimpleSnapshot<T> implements Snapshot<T> {
  // array of atomic MRSW registers
  private StampedValue<T>[] a_table;
  ...
  public void update(T value) {
    int me = ThreadID.get();
    StampedValue<T> oldValue = a_table[me];
    StampedValue<T> newValue =
        new StampedValue<T>((oldValue.stamp)+1, value);
    a_table[me] = newValue;
  }
  private StampedValue<T>[] collect() {
    StampedValue<T>[] copy = (StampedValue<T>[]) new StampedValue[a_table.length];
    for (int j = 0; j < a_table.length; j++)
      copy[j] = a_table[j];
    return copy;
  }
  public T[] scan() {
    StampedValue<T>[] oldCopy, newCopy;
    oldCopy = collect();
    collect: while (true) {
      newCopy = collect();
      if (! Arrays.equals(oldCopy, newCopy)) {
        oldCopy = newCopy;
        continue collect;
      }
      // clean collect
      T[] result = (T[]) new Object[a_table.length];
      for (int j = 0; j < a_table.length; j++)
        result[j] = newCopy[j].value;
      return result;
    }
  }
}
```

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 BackoffLockTest

5.1 Particular Case (problem)

This is part of the chapter 7, where we are dealing with the generic issue of building real-life lock implementations that solve the multual exclusion problem. The particular subproblem is doing that with spinning locks (those which repeatedly try to acquire the lock); and the very concrete sub-subproblem here is how to reduce the contention out of the *TTASLock*.

5.2 Solution

The solution is based on a key observation, for which we need to cite the *TTASLock* code first:

```
public class TTASLock implements Lock {
  AtomicBoolean state = new AtomicBoolean(false);
  public void lock() {
    while (true) {
      while (state.get()) {}; // spin
      if (!state.getAndSet(true))
        return;
    }
  }
  public void unlock() {
    state.set(false);
  }
  ...
}
```

If between the calls to *get* and *getAndSet*, another thread took the look; then it must be that there is high contention for such resource (many threads trying to acquire it). Insisting on acquiring the lock on high contention scenarios is a bad idea; we will add a lot of traffic to the bus used by cores to communicate (as primitive *getAndSet* forces reads to be consistent) and the chances of getting it are scarce (too much competion).

The solution then, is to wait some time before retrying; how much? Well, the more we fail to acquire the lock the more we wait next time. The concrete solution for that, on the context of this test, is to use what is called "adaptive exponential backoff". Each time we fail to acquire the lock, a random number is generated within a given range and the thread sleeps that time [3]. Everytime we do this waiting, the range is enlarged twice (meaning that on average the random number is twice larged); this is done up to a maximum established (1024 in this case). Code is presented below for reference (note that we also have a lower bound for the minimum amount of time to wait; being 32 milliseconds here):

```
public class BackoffLock implements Lock {
  private Backoff backoff;
  private AtomicBoolean state = new AtomicBoolean(false);
  private static final int MIN_DELAY = 32;
  private static final int MAX_DELAY = 1024;
  public void lock() {
    Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);
    while (true) {
      while (state.get()) {};   // spin
      if (!state.getAndSet(true)) { // try to acquire lock
        return;
      } else {                  // backoff on failure
        try {
          backoff.backoff();
        } catch (InterruptedException ex) {
        }
      }
    }
  }
  public void unlock() {
    state.set(false);
  }
  ...
}

public class Backoff {
  ...

  /**
   * Backoff for random duration.
   * @throws java.lang.InterruptedException
```

[3]This sample uses milliseconds, a common unit when dealing with multi-thread programming

```
   */
  public void backoff() throws InterruptedException {
    int delay = random.nextInt(limit);
    if (limit < maxDelay) { // double limit if less than max
      limit = 2 * limit;
    }
    Thread.sleep(delay);
  }
}
```

By the way, the code for *Backoff* has an error (not shown above to save space in this report); the argument *max* passed to the constructor is not really used (there seems to be a typo there, where we used *min* parameter instead).

5.3 Experiment Description

The test consists in a single case *testParallel*, which creates 8 threads and puts each one to increase 1024 times a shared counter. The access to the counter is protected by the *BackoffLock* solution we described already. The assertion of the test is that the shared counter ends up with a final value, that reflects the fact that all the thread contributions are present (nobody lost increments, and we do not have more than expected).

5.4 Observations and Interpretations

With or without the fix mentioned about the *max* argument constructor in *Backoff* class, the test works fine on 2 and 24 cores. Below a sample execution:

```
.parallel

Time: 0.075

OK (1 test)
```

6 CompositeLockTest

6.1 Particular Case (problem)

The particular problem we are trying to solve, is that of looking for a good balance between the traceoffs imposed by queue locks vs backoff locks. On one hand, queue locks provide FIFO fairness, fast lock release, and low contention, but require nontrivial protocols for recycling abandoned nodes; on the other hand, backoff locks support trivial timeout protocols, but are not scalable, and may have slow lock release if timeout parameters are not well-tuned.

6.2 Solution

The solution proposed is to combine both type of locks, and is based on the observation that only the threads at the front of the queue need to do lock handoffs; while the rest may be happy doing backoffs. This solution occupies several pages at the book, and putting all the code and the details here may be cumbersome; let us just put the main high-level details.

The *CompositeLock* solution uses a short array of auxiliary lock nodes (the array is static, it can not be resized). Once a thread needs to acquire "the" lock it picks a random slot in the array; which would have a node. If the auxiliary lock protecting that node is used, the thread uses an adaptive backoff approach and retries (after sleeping the required time). Once the thread could acquire the lock over the picked node, it places that node into a *TOLock*-style queue. Then, the thread spins on the preceding node, and when that node's owner indicates it has finished, the thread can finally acquire "the" lock (and enter its critical section). When the thread leaves due completion or time-out, it releases ownership of the node, and another thread doing backoff can acquire it. There are far more details regarding the procedure to recycle the freed nodes of the array, while multiple threads attempt to acquire control over them; we omit those details here.

6.3 Experiment Description

There is a single test case *testParallel*, which creates a couple of threads and asks each to increment 2048 times a shared counter. The counter increment is the critical section, and is protected with a *CompositeLock*. At the end the shared counter shall have a value of 4096.

6.4 Observations and Interpretations

The test runs fine on our two and 24 core machines, sample execution below:

```
.
Time: 0.066

OK (1 test)
```

7 MCSLockTest

7.1 Particular Case (problem)

The problem is still how to build scalable spinning locks, and in particular, how to overcome the limitations from the backoff approaches. The queue-based approach introduced both *ALock* and *CLHLock*, the first is space inefficient and the second overcomes such limitation at the expense of not being suitable for cache-less NUMA architecture.

7.2 Solution

The solution or rather alternative to *CLHLock* approach is called *MCSLock*; and that is the algorithm tested in this section. Just like the *CLHLock* approach, the lock is modeled with a linked list of QNode objects, where each one represents either a lock holder or a thread waiting to acquire the lock; the difference though, is that the list is explicit, not virtual [4]. On the *MCSLock* approach, instead of embedding the list in thread-local variables, it is placed in the globally accessible QNode objects.

The *MCSLock* code is presented below for further reference:

```
public class MCSLock implements Lock {
  AtomicReference<QNode> queue;
  ThreadLocal<QNode> myNode;
  public MCSLock() {
    queue = new AtomicReference<QNode>(null);
    // initialize thread-local variable
    myNode = new ThreadLocal<QNode>() {
      protected QNode initialValue() {
        return new QNode();
      }
    };
  }
  public void lock() {
    QNode qnode = myNode.get();
    QNode pred = queue.getAndSet(qnode);
    if (pred != null) {
      qnode.locked = true;
```

[4]The author introduces the term “virtual”, when describing *CLHLock* because the list is implicit: each thread refers to its predecessor through a thread-local *pred* variable.

```
        pred.next = qnode;
        while (qnode.locked) {}      // spin
      }
    }
    public void unlock() {
      QNode qnode = myNode.get();
      if (qnode.next == null) {
        if (queue.compareAndSet(qnode, null))
          return;
        while (qnode.next == null) {} // spin
      }
      qnode.next.locked = false;
      qnode.next = null;
    }
    ...
    static class QNode {      // Queue node inner class
      boolean locked = false;
      QNode    next = null;
    }
}
```

This *MCSLock* solution is not a perfect replacement for *CLHLock*: on one side it has same advantages than *CLHLock*, in particular, the property that each lock release invalidates only the successor's cache entry, with the plus that it is better suited to cache-less NUMA architectures because each thread controls the location on which it spins. The space complexity is the same as *CLHLock*, $\mathrm{O}(L + n)$, as the nodes can be recycled.

On the other side, the disadvantages of *MCSLock* algorithm is that releasing a lock requires spinning; and that it requires more reads, writes, and compareAndSet() calls than the *CLHLock* algorithm.

7.3 Experiment Description

The test is exactly the same as that of *BackoffLockTest*: 8 threads try to increment 1024 times a shared counter, and they do that concurrently. At the end the counter shall have $8 * 1024$ as final value.

7.4 Observations and Interpretations

The test works fine in two-cores, but in 24 cores it hangs time to time; below a sample thread dump captured when hanging occurred (after trying the test a bit more than 300 times):

```
Full thread dump OpenJDK 64-Bit Server VM (23.7-b01 mixed mode):

"Thread-7" prio=10 tid=0x00007f3c4c0e0000 nid=0x6aaa runnable [0x00007f3bf6e4c000]
   java.lang.Thread.State: RUNNABLE
        at spin.MCSLock.lock(MCSLock.java:41)
        at spin.MCSLockTest$MyThread.run(MCSLockTest.java:52)

"Thread-6" prio=10 tid=0x00007f3c4c0de000 nid=0x6aa9 runnable [0x00007f3bf6f4d000]
   java.lang.Thread.State: RUNNABLE
        at spin.MCSLock.lock(MCSLock.java:41)
        at spin.MCSLockTest$MyThread.run(MCSLockTest.java:52)

"Thread-5" prio=10 tid=0x00007f3c4c0dc000 nid=0x6aa8 runnable [0x00007f3bf704e000]
   java.lang.Thread.State: RUNNABLE
        at spin.MCSLock.lock(MCSLock.java:41)
        at spin.MCSLockTest$MyThread.run(MCSLockTest.java:52)

"Thread-4" prio=10 tid=0x00007f3c4c0da000 nid=0x6aa7 runnable [0x00007f3bf714f000]
   java.lang.Thread.State: RUNNABLE
        at spin.MCSLock.lock(MCSLock.java:41)
        at spin.MCSLockTest$MyThread.run(MCSLockTest.java:52)

"Thread-3" prio=10 tid=0x00007f3c4c0d7000 nid=0x6aa6 runnable [0x00007f3bf7250000]
   java.lang.Thread.State: RUNNABLE
        at spin.MCSLock.lock(MCSLock.java:41)
        at spin.MCSLockTest$MyThread.run(MCSLockTest.java:52)

"Thread-2" prio=10 tid=0x00007f3c4c0d5800 nid=0x6aa5 runnable [0x00007f3bf7351000]
   java.lang.Thread.State: RUNNABLE
        at spin.MCSLock.lock(MCSLock.java:41)
        at spin.MCSLockTest$MyThread.run(MCSLockTest.java:52)

"Thread-1" prio=10 tid=0x00007f3c4c0d4000 nid=0x6aa4 runnable [0x00007f3bf7452000]
   java.lang.Thread.State: RUNNABLE
        at spin.MCSLock.lock(MCSLock.java:41)
        at spin.MCSLockTest$MyThread.run(MCSLockTest.java:52)

"Thread-0" prio=10 tid=0x00007f3c4c0d8800 nid=0x6aa3 runnable [0x00007f3bf7553000]
   java.lang.Thread.State: RUNNABLE
        at spin.MCSLock.lock(MCSLock.java:41)
        at spin.MCSLockTest$MyThread.run(MCSLockTest.java:52)
...
"main" prio=10 tid=0x0000000000c8a800 nid=0x6a79 in Object.wait() [0x00007f3c5ce0d000]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x000000058386adc8> (a spin.MCSLockTest$MyThread)
        at java.lang.Thread.join(Thread.java:1260)
        - locked <0x000000058386adc8> (a spin.MCSLockTest$MyThread)
        at java.lang.Thread.join(Thread.java:1334)
        at spin.MCSLockTest.testParallel(MCSLockTest.java:43)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:606)
        at junit.framework.TestCase.runTest(TestCase.java:176)
        at junit.framework.TestCase.runBare(TestCase.java:141)
        at junit.framework.TestResult$1.protect(TestResult.java:122)
        at junit.framework.TestResult.runProtected(TestResult.java:142)
        at junit.framework.TestResult.run(TestResult.java:125)
        at junit.framework.TestCase.run(TestCase.java:129)
        at junit.framework.TestSuite.runTest(TestSuite.java:252)
        at junit.framework.TestSuite.run(TestSuite.java:247)
        at junit.textui.TestRunner.doRun(TestRunner.java:116)
        at junit.textui.TestRunner.start(TestRunner.java:183)
        at junit.textui.TestRunner.main(TestRunner.java:137)
```

From the above stacks, we can observe that the main thread hangs because none of the launched 8 threads gets out of the spinning while of *lock* method (line 41):

```
        while (qnode.locked) {}        // spin
```

The reason of the above must be that the *locked* flag is not declared as *volatite*; without such keyword, the JVM does not guarantee any synchronization time limits between the *RAM* and the local caches of each core. Thus, the hanging shall occur when the write made by last thread to its local flag (*locked=false*, did not get propagated to any other core for a while; we do now know when they would get propagated, actually.

This is an interesting aspect of the *AtomicReference* class; while it does cover the "reference" to a *QNode*, it does not appear to affect the attributes of the object. Thus, changes to the *queue* reference will be atomic indeed; but they would point to attributes whose writes are not.

One may be tempted to think that the *volatile* keyword is only applicable to the *locked* boolean flag; but even with that chance the test may still hang. Below sample thread dump on the 24 cores machine, after retrying test 1118 times:

```
Full thread dump OpenJDK 64-Bit Server VM (23.7-b01 mixed mode):
```

```
"Thread-7" prio=10 tid=0x00007f55b00ef800 nid=0x9e07 runnable [0x00007f555c6e8000]
   java.lang.Thread.State: RUNNABLE
        at spin.MCSLock.unlock(MCSLock.java:49)
        at spin.MCSLockTest$MyThread.run(MCSLockTest.java:56)

"Thread-6" prio=10 tid=0x00007f55b00ed800 nid=0x9e06 runnable [0x00007f555c7e9000]
   java.lang.Thread.State: RUNNABLE
        at spin.MCSLock.lock(MCSLock.java:41)
        at spin.MCSLockTest$MyThread.run(MCSLockTest.java:52)

  ...

"main" prio=10 tid=0x00000000018eb800 nid=0x9dd6 in Object.wait() [0x00007f55c2736000]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x000000058386b910> (a spin.MCSLockTest$MyThread)
        at java.lang.Thread.join(Thread.java:1260)
        - locked <0x000000058386b910> (a spin.MCSLockTest$MyThread)
        at java.lang.Thread.join(Thread.java:1334)
        at spin.MCSLockTest.testParallel(MCSLockTest.java:43)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:606)
        at junit.framework.TestCase.runTest(TestCase.java:176)
        at junit.framework.TestCase.runBare(TestCase.java:141)
        at junit.framework.TestResult$1.protect(TestResult.java:122)
        at junit.framework.TestResult.runProtected(TestResult.java:142)
        at junit.framework.TestResult.run(TestResult.java:125)
        at junit.framework.TestCase.run(TestCase.java:129)
        at junit.framework.TestSuite.runTest(TestSuite.java:252)
        at junit.framework.TestSuite.run(TestSuite.java:247)
        at junit.textui.TestRunner.doRun(TestRunner.java:116)
        at junit.textui.TestRunner.start(TestRunner.java:183)
        at junit.textui.TestRunner.main(TestRunner.java:137)
```

In the above dump there are only hanging threads two; “Thread-6” is waiting on the already seen spinning loop of *lock* method, which should no longer be attributable to the lack of *volatile* attribute (we just fixed that). This spinning loop of the “Thread-6” hanging thread then, must be valid: the *locked* flag of the other thread has not changed.

The new part is the other hanging thread “Thread-7”, which is waiting on the spinning loop of the *unlock* method:

```
        while (qnode.next == null) {} // spin
```

We immediately notice same problem as with *locked* flag; the *next* pointer also needs immediate propagation to main memory, otherwise the threads will not detect then is being set to null. Let us also note that the hanging due *next* variable, triggers the hanging on the spinning loop for *lock* method; the *locked* flag is volatile, so a write to it would propagate right away and invalidate other's caches; but since the thread doing the *unlock* never exits the loop, it can not perform such write. Putting as volatile both attribues of class *Qnode* solves this issue; we ran the test 10,000 times without a hang.

An interesting observation made while discussing this exercise on the distribution list, was that the solution imposes a serialization, after all; but that is correct. Let us remember that the queue is just the internal implementation artifact of the high level lock construction. And for this particular test, we are trying to solve the mutual-exclusion problem for a counter; thus serialization is what we want, indeed.

Even if serialization is required, there are several ways of achieving it; some are more efficient at hardware level than others. The family of solutions that *MCSLock* belongs to, tries to minimize traffic on the core buses (as we only invalidate the cache of the core that owes the modified variables *locked* / *next*).

8 TTASLockTest

8.1 Particular Case (problem)

Here we are trying to address the shortcomings of the *TASLock* solution, which we know per the book causes a lot of bus traffic.

8.2 Solution

The solution is to loop on a locally cached variable, so that we do not generate traffic on the bus. The code is simple enough and is presented below:

```java
public class TTASLock implements Lock {
  AtomicBoolean state = new AtomicBoolean(false);
  public void lock() {
    while (true) {
      while (state.get()) {};  // spin
      if (!state.getAndSet(true))
        return;
    }
  }
  public void unlock() {
    state.set(false);
  }
  ...
}
```

Only the first time the *get* in the loop is called it generates bus traffic, but after that and until the moment where the state is changed by the owner of the lock, each thread uses its core cache. The drawback of this solution though, is that traffic comes once the owner releases the lock; in that moment the caches of all the cores waiting are invalidated, generating bus traffic. Furthermore, once unleashed all the threads/cores will call the synchronization primitive *getAndSet* (which again will generate traffic on the bus). After some fierce competition, once we have a winner, the system will reach again an stable configuration (looser threads spinning on local variables).

8.3 Experiment Description

The test is exactly the same as that of *BackoffLockTest*: 8 threads try to increment 1024 times a shared counter, and they do that concurrently. At the end the counter shall have $8 * 1024$ as final value.

8.4 Observations and Interpretations

The test runs fine on 2 and 24 cores, below a sample successful execution:

```
.parallel

Time: 0.024

OK (1 test)
```

We did not require to set to volatile any of the variables, as they were already atomic (*AtomicBoolean* implies same guarantees than *volatile*, and even more).

9 QueueTest

9.1 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 [5] capabilities offered by Java language; when compared to merely using the *synchronized* feature.

9.2 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

[5]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:

```java
class Queue<T> {
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();
  final T[] items;
  int tail, head, count;
  public Queue(int capacity) {
    items = (T[])new Object[capacity];
  }
  public void enq(T x) throws InterruptedException {
    lock.lock();
    try {
      while (count == items.length)
        notFull.await();
      items[tail] = x;
      if (++tail == items.length)
        tail = 0;
      ++count;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }
  public T deq() throws InterruptedException {
    lock.lock();
    try {
      while (count == 0)
        notEmpty.await();
      T x = items[head];
      if (++head == items.length)
        head = 0;
      --count;
      notFull.signal();
      return x;
    } finally {
      lock.unlock();
    }
  }
}
```

9.3 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).

9.4 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)
```

10 LazyListTest

10.1 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).

10.2 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:

```
public boolean remove(T item) {
  int key = item.hashCode();
  while (true) {
    Node pred = this.head;
    Node curr = head.next;
    while (curr.key < key) {
      pred = curr; curr = curr.next;
    }
    pred.lock();
    try {
      curr.lock();
      try {
        if (validate(pred, curr)) {
          if (curr.key != key) {     // present
            return false;
          } else {                   // absent
            curr.marked = true;      // logically remove
            pred.next = curr.next;   // physically remove
            return true;
          }
        }
      } finally {                    // always unlock curr
        curr.unlock();
      }
    } finally {                      // always unlock pred
      pred.unlock();
    }
  }
}

public boolean contains(T item) {
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key)
    curr = curr.next;
  return curr.key == key && !curr.marked;
}
```

10.3 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.

10.4 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.

11 BoundedQueueTest

11.1 Particular Case (problem)

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

11.2 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:

```
public void enq(T x) {
  if (x == null) throw new NullPointerException();
  boolean mustWakeDequeuers = false;
  enqLock.lock();
  try {
    while (size.get() == capacity) {
      try {
        notFullCondition.await();
      } catch (InterruptedException e) {}
    }
    Entry e = new Entry(x);
    tail.next = e;
    tail = e;
    if (size.getAndIncrement() == 0) {
      mustWakeDequeuers = true;
    }
  } finally {
    enqLock.unlock();
  }
  if (mustWakeDequeuers) {
    deqLock.lock();
    try {
      notEmptyCondition.signalAll();
    } finally {
      deqLock.unlock();
    }
  }
}
```

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?).

11.3 Experiment Description

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

11.4 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.

12 LockFreeQueueRecycleTest

12.1 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.

12.2 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).

12.3 Experiment Description

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

12.4 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)
```

13 SynchronousDualQueueTest

13.1 Particular Case (problem)

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

13.2 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.

13.3 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.

13.4 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.

Now, if we dig deeper into the explanation, we can remember that the solution dos pairing between enqueuers and dequeuers; but if the dequeuers come first and there is no data, the *deq* method will return null, and the test is just not prepared to handle such data (it will need an adaptation like the one explained above). Since a dequeuer dies, there will be enqueuers that remain waiting forever for their pairing mates. This should be the reason behind the hanging.

14 TreeTest

14.1 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.

14.2 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 representant 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 (1) vs O ($log(p)$)), where p is the number of threads or processors; but it offers a better throughput.

14.3 Experiment Description

The program consists of a single unit test *testGetAndIncrement*, which creates 8 threads to perform each 2^20 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.

14.4 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.

15 CoarseCuckooHashSetTest

15.1 Particular Case (problem)

The problem we try to solve is that of concurrent hashing, when we use the hash to implement a set data structure.

15.2 Solution

The solution uses three techniques combined:

- cuckoo-hashing: which consists in shifting to the right elements, by performing consecutive swaps on elements having a collision due hash function.

- double hashing where we use a couple of hash tables; the first one is based on a simple hash function (modulus of the element's hash code), and the second table is indexed with random numbers generated out of a sequence feed from the hash code of elements.
- coarse access: as the whole structure is being protected by mutual exclusion with a lock.

Below the most representative function, *add*, along with the two hash functions used:

```
private final int hash0(Object x) {
  return x.hashCode() % size;
}

private final int hash1(Object x) {
  random.setSeed(x.hashCode());
  return random.nextInt(size);
}

public boolean add(T x) {
  lock.lock();
  try {
      if (contains(x)) {
        return false;
      }
      for (int i = 0; i < LIMIT; i++) {
        if ((x = swap(0, hash0(x), x)) == null) {
          return true;
        } else if ((x = swap(1, hash1(x), x)) == null) {
          return true;
        }
      }
      System.out.println("uh-oh");
      throw new CuckooException();
    } finally {
      lock.unlock();
    }
  }
```

Something additional to mention about this solution, is that when adding a new element we constraint the maximum number of swaps to perform (while shifting the elements to the right). In this implementation is *LIMIT* is 32; if after those attempts we could not arrange all elements on its slot we raise an exception.

15.3 Experiment Description

The test program consists of three test cases:

- *testSequential*: add 512 elements to the hash set (sequentially), perform a *constains* test for each one (sequentially) and finally remove the 512 elements (sequentially).

- *testParallelAdd*: add 512 elements to the hash set (in parallel with 8 threads), perform a *constains* test for each one (sequentially) and finally remove the 512 elements (sequentially).

- *testParallelRemove*: add 512 elements to the hash set (sequentially), perform a *constains* test for each one (sequentially) and finally remove the 512 elements (in parallel with 8 threads).

- *testParallelBoth*: adds and removes 512 elements to the hash set in parallel with 8 adder threads, and 8 remover threads; the removers complain if someone else removed its designated range of elements.

15.4 Observations and Interpretations

The tests works fine on both machines (2 and 24 cores respectively), with the exception that the parallel test remover threads complain when they try to delete an element which does not exist yet (which is an expected consequence of the test). A sample output is shown below:

```
[oraadm@gdlaa008 orig]$ junit hash.CoarseCuckooHashSetTest

.sequential add, contains, and remove
.parallel both
Exception in thread "Thread-5" Exception in thread "Thread-9"
                Exception in thread "Thread-11" Exception in thread

"Thread-1" junit.framework.AssertionFailedError: DeqThread: duplicate

remove
at junit.framework.Assert.fail(Assert.java:57)
at junit.framework.TestCase.fail(TestCase.java:227)
at

...

.parallel add
.parallel remove

Time: 0.053

OK (4 tests)
```

16 LockFreeHashSetTest

16.1 Particular Case (problem)

The problem we try to solve is that of implementing a lock-free hash set.

16.2 Solution

The solution uses a technique called “Recursive Split-Ordering”, where we keep all items on a single lock-free linked list, and we keep separately an increasing array of buckets (which are just references to different positions of the list). While any element could be found in the set by transversing the list, idea is to use hash functions to take advantage of the shortcuts that the buckets represent. The algorithm takes its name from the fact that the split-ordered-keys on the list, are the bitwise reverse representation of their items keys (hash codes).

16.3 Experiment Description

The test is exactly the same as that described for *CoarseCuckooHashSetTest*.

16.4 Observations and Interpretations

The test runs fine in 3 and 24 core machines, and similarly to *CoarseCuckooHashSetTest*, suffers from a design flag on the test program itself (remover threads may attempt to delete an element which has not been added yet). Below a sample output of its execution:

```
[oraadm@gdlaa008 orig]$ junit hash.LockFreeHashSetTest
.sequential add, contains, and remove
.parallel both
Exception in thread "Thread-1" junit.framework.AssertionFailedError:
                DeqThread: duplicate pop
at junit.framework.Assert.fail(Assert.java:57)
at junit.framework.TestCase.fail(TestCase.java:227)
at

hash.LockFreeHashSetTest$RemoveThread.run(LockFreeHashSetTest.java:143)
.parallel add
.parallel remove

Time: 0.037

OK (4 tests)
```

17 StripedCuckooHashSetTest

17.1 Particular Case (problem)

The problem we try to solve is that of concurrent hashing, when we use the hash to implement a set data structure.

17.2 Solution

This solution is a refinement of the cuckoo technique we mentioned for other solutions like *CoarseCuckooHashSetTest*; providing a matrix of reentrant locks, where each cell protects a cell on the hash table. The following constructor shows how the locks are initialized:

```
  public StripedCuckooHashSet(int capacity) {
    super(capacity);
    lock  = new ReentrantLock[2][capacity];
    for (int i = 0; i < 2; i++) {
      for (int j = 0; j < capacity; j++) {
        lock[i][j] = new ReentrantLock();
      }
    }
  }
```

This solution extends the solution named *PhasedCuckooHashSet*. Part of its main differences are the *acquire* and *release* methods; which follow certain order on the locks associated to items, in order to avoid deadlocks:

```
  public final void acquire(T x) {
    Lock lock0 = lock[0][hash0(x) % lock[0].length];
    Lock lock1 = lock[1][hash1(x) % lock[1].length];
    lock0.lock();
    lock1.lock();
  }

  public final void release(T x) {
    Lock lock0 = lock[0][hash0(x) % lock[0].length];
    Lock lock1 = lock[1][hash1(x) % lock[1].length];
    lock0.unlock();
    lock1.unlock();
    }
```

17.3 Experiment Description

The test is in essence the same as that described for *CoarseCuckooHash-SetTest*, with some small differences: among them, the most relevant is the remover thread class used in *testParallelBoth*, which does not fail anymore if the element it tried to remove was not present; it only increments a counter of missed deletions. This minor change prevents the test from failing like the others.

17.4 Observations and Interpretations

The test runs fine in 2 and 24 cores, without any assertion error; below a sample execution:

```
[oraadm@gdlaa008 orig]$ junit hash.StripedCuckooHashSetTest
.sequential add, contains, and remove
.parallel both
.parallel add
.parallel remove

Time: 0.017

OK (4 tests)
```

18 ArraySumTest

18.1 Particular Case (problem)

The problem we want to solve is that of summing up all the elements of an array; but of course, we want to do it in parallel.

18.2 Solution

The solution uses the divide and conquer approach, creating a tree of tasks (*Future* from java concurrency library); which represent the processing we do while splitting the arrays in two halves and summing up each half recursively. Base case are arrays of a single element. Below the heart of the solution, the recursive function which create new tasks for subarrays:

```
public Integer call() throws InterruptedException, ExecutionException {
  if (size == 1) {
    return a[start];
  } else {
    int lhsSize = size / 2;
    int rhsStart = lhsSize + 1;
    int rhsSize = size - lhsSize;
    Future<Integer> lhs = exec.submit(new SumTask(a, start, lhsSize));
    Future<Integer> rhs = exec.submit(new SumTask(a, rhsStart, rhsSize));
    return rhs.get() + lhs.get();
  }
}
```

18.3 Experiment Description

The test merely consists in trying the algorithm with a couple of arrays of consecutive positive integers, of respective sizes 3 and 4. The test validates that the expected sum is 6 and 10 respectively.

18.4 Observations and Interpretations

The test presents failures like the one below, where the array of 3 elements gave as sum 7 instead of 6:

```
[oraadm@gdlaa008 orig]$ junit steal.ArraySumTest
.F
Time: 0.009
There was 1 failure:
1) testRun(steal.ArraySumTest)junit.framework.AssertionFailedError:
                expected:<6> but was:<7>
at steal.ArraySumTest.testRun(ArraySumTest.java:31)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at

sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
at

sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

FAILURES!!!
Tests run: 1,  Failures: 1,  Errors: 0
```

The problem lied on the program *ArraySum.java*, on the assignment of variable *rhsStart*, which represents the starting point of the right subarray. While the original program had $lhsSize + 1$ as assignment, the proper expression was $lhsSize + start$ (so we take into account both the relative start of the initial subarray we got, as well as the length of the left sub-subarray). After this fix, the test worked fine on both 2 and 24 cores machines.

19 FibTaskTest

19.1 Particular Case (problem)

The problem is calculating the sum of the first n Fibonacci numbers, in parallel.

19.2 Solution

The solution merely follows the recursive definition of the Fibonacci sequence, whose base cases are 1 (for first and second elements of the sequence). Each recursive call of the function creates a new task (*Future* from java concurrency library). Below the code of the main function:

```java
public Integer call() {
  try {
    if (arg > 2) {
      Future<Integer> left = exec.submit(new FibTask(arg-1));
      Future<Integer> right = exec.submit(new FibTask(arg-2));
      return left.get() + right.get();
    } else {
      return 1;
    }
  } catch (Exception ex) {
    ex.printStackTrace();
    return 1;
  }
}
```

Note that *Future* class represents asynchronous calculations, and that the *get* method is blocking (waits for calculation to be complete); hence we would effectively create a tree of tasks until we reach the base cases and start evaluating bottom up.

19.3 Experiment Description

The experiment merely request the calculation of the sum up to a known number (16), and asserts the expected result (987).

19.4 Observations and Interpretations

The test works as expected in both 2 and 24 cores machines. Below a sample output:

```
[oraadm@gdlaa008 orig]$ junit steal.FibTaskTest
.run

Time: 0.128

OK (1 test)
```

20 MMThreadTest

20.1 Particular Case (problem)

The problem is that of matrix multiplication, but done in parallel.

20.2 Solution

The solution creates one thread per cell on the result matrix, and for each one it creates a new worker thread; which in turn calculates the cell (i, j) value by computing the dot product of row i of left matrix by column j of right matrix (according to the definition of the matrix product). Below the most relevant sections of the code:

```java
public MMThread(double[][] a, double[][] b) {
  n = a.length;
  this.a = a;
  this.b = b;
  this.c = new double[n][n];
}

void multiply() {
  Worker[][] worker = new Worker[n][n];
  // create one thread per matrix entry
  for (int row = 0; row < n; row++) {
    for (int col = 0; col < n; col++) {
      worker[row][col] = new Worker(row,col);
    }
  }
  ...
}

class Worker extends Thread {
  int row, col;
  Worker(int row, int col) {
    this.row = row; this.col = col;
  }
  public void run() {
    double dotProduct = 0.0;
    for (int i = 0; i < n; i++) {
      dotProduct += a[row][i] * b[i][col];
    }
    c[row][col] = dotProduct;
```

```
        }
    }
```

20.3 Experiment Description

The test consists of multiplying two identify matrices of dimensions 3×3, and compare the result with one of the operands (it should be identity matrix again).

20.4 Observations and Interpretations

The test runs fine on 2 and 24 core machines, below a sample output:

```
[oraadm@gdlaa008 orig]$ junit steal.MMThreadTest
.run

Time: 0.004

OK (1 test)
```

21 BitReversedCounterTest

21.1 Particular Case (problem)

The problem is actually a subproblem of the *FineGrainedHeap* program, which has a *next* variable indicating next slot to use on insert. The problem is to ensure that two consecutive paths from root to leaves, where insertion takes place, share no common nodes other than the root (this is useful to ensure that each insertion thread locks a disjoint set of nodes, reducing contention on multi-threaded scenarios).

21.2 Solution

The solution is a bit cryptic, but small enough to fit here. It can be proved that the *reverseIncrement* method below, produces sequence of numbers with the above property. Given time constraints, we did not prove such claim (specially cause this test itself, did not exhibit concurrency):

```
public int reverseIncrement() {
  if (counter++ == 0) {
    reverse = highBit = 1;
    return reverse;
  }
  int bit = highBit >> 1;
  while (bit != 0) {
    reverse ^= bit;
    if ((reverse & bit) != 0) break;
    bit >>= 1;
  }
  if (bit == 0)
    reverse = highBit <<= 1;
  return reverse;
}
```

21.3 Experiment Description

The test does not really make any assertions, and simply call the *reverseIncrement* and *reverseDecrement* methods printing their results.

21.4 Observations and Interpretations

The test did not show any failures or hanging, given that is totally sequential and does not really validate anything. Below a sample execution output:

```
[oraadm@gdlaa008 orig]$ junit priority.BitReversedCounterTest
.increment
inc:       0       1
inc:       1       2
inc:       2       3
inc:       3       4
inc:       4       6
inc:       5       5
inc:       6       7
inc:       7       8
inc:       8       12
inc:       9       10
inc:       10       14
inc:       11       9
inc:       12       13
inc:       13       11
inc:       14       15
inc:       15       16
inc:       16       24
inc:       17       20
inc:       18       28
inc:       19       18
inc:       20       26
inc:       21       22
inc:       22       30
inc:       23       17
inc:       24       25
inc:       25       21
inc:       26       29
inc:       27       19
inc:       28       27
inc:       29       23
inc:       30       31
inc:       31       32
inc:       32       48
inc:       33       40
inc:       34       56
inc:       35       36
inc:       36       52
inc:       37       44
inc:       38       60
inc:       39       34
inc:       40       50
inc:       41       42
inc:       42       58
inc:       43       38
inc:       44       54
inc:       45       46
inc:       46       62
inc:       47       33
```

```
inc:        48        49
inc:        49        41
inc:        50        57
inc:        51        37
inc:        52        53
inc:        53        45
inc:        54        61
inc:        55        35
inc:        56        51
inc:        57        43
inc:        58        59
inc:        59        39
inc:        60        55
inc:        61        47
inc:        62        63
inc:        63        64
decrement
dec:        63
dec:        47
dec:        55
dec:        39
dec:        59
dec:        43
dec:        51
dec:        35
dec:        61
dec:        45
dec:        53
dec:        37
dec:        57
dec:        41
dec:        49
dec:        33
dec:        62
dec:        46
dec:        54
dec:        38
dec:        58
dec:        42
dec:        50
dec:        34
dec:        60
dec:        44
dec:        52
dec:        36
dec:        56
dec:        40
dec:        48
dec:        32

Time: 0.022

OK (1 test)
```

22 SimpleLinearTest

22.1 Particular Case (problem)

The problem is to implement a bounded priority queue, which are a multi-set data structure; each set has an associated priority. The contract or interface for a priority queue is essentially two methods: *add(x,k)* which adds element x to the set of priority k, and *removeMin* which returns and removes the element with smaller (higher) priority from all the sets. This type of priority queues are called "bounded", because the priority of elements is taken from a finite predefined set.

22.2 Solution

Given that the problem is to implement a bounded priority queue, an array shall suffice. So we create an array of sets as big as the range of values for the priority. The *add* and *removeMin* methods are straighforward, but they are built on the assumption that the auxiliary class *Bin* is thread safe (which is the case, as it is implemented with locks). Below the relevant snippets of code:

```
public class SimpleLinear<T> implements PQueue<T> {
  int range;
  Bin<T>[] pqueue;

  ...

  public void add(T item, int key) {
    pqueue[key].put(item);
  }

  ...

  public T removeMin() {
    for (int i = 0; i < range; i++) {
       T item = pqueue[i].get();
       if (item != null) {
         return item;
       }
    }
    return null;
    }
}
```

22.3 Experiment Description

The test program consists of the following test cases:

- *testAdd*: Serial test which adds 512 random numbers in a fixed range (these numbers become the priority), and later validates that they are removed in ascending order with *removeMin*.

- *testParallelAdd*: Same as *testAdd*, with the exception that the addition of the elements is done in parallel with 8 threads.

- *testParallelBoth*: Same as *testParallelAdd*, except that the removal of items is also performed in parallel with 8 threads.

22.4 Observations and Interpretations

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

```
[oraadm@gdlaa008 orig]$ junit priority.SimpleLinearTest
.add
OK.
...
OK.
OK.
.testParallelBoth
OK.
.testParallelAdd
OK.

Time: 0.116

OK (3 tests)
```

23 ArraySumTest

23.1 Particular Case (problem)

The problem we want to solve is that of summing up all the elements of an array; but of course, we want to do it in parallel (this is the same as the test program of same name from chapter 14).

23.2 Solution

The solution uses the divide and conquer approach, creating a tree of tasks (*Future* from java concurrency library); which represent the processing we do while splitting the arrays in two halves and summing up each half recursively. Base case are arrays of a single element. Below the heart of the solution, the recursive function which create new tasks for subarrays:

```java
  public Integer call() throws InterruptedException, ExecutionException {
    if (size == 1) {
      return a[start];
    } else {
      int lhsSize = size / 2;
      int rhsStart = lhsSize + 1;
      int rhsSize = size - lhsSize;
      Future<Integer> lhs = exec.submit(new SumTask(a, start, lhsSize));
      Future<Integer> rhs = exec.submit(new SumTask(a, rhsStart, rhsSize));
      return rhs.get() + lhs.get();
    }
  }
```

Above code is exactly the same as the one from chapter 14; we are not sure whether this repetition was intentional, but putting anyway for the sake of completeness.

23.3 Experiment Description

The test merely consists in trying the algorithm with a couple of arrays of consecutive positive integers, of respective sizes 3 and 4. The test validates that the expected sum is 6 and 10 respectively (again, same as chapter 14).

23.4 Observations and Interpretations

The test presents failures like the one below, where the array of 3 elements gave as sum 7 instead of 6:

```
[oraadm@gdlaa008 orig]$ junit steal.ArraySumTest
.F
Time: 0.01
There was 1 failure:
1) testRun(steal.ArraySumTest)junit.framework.AssertionFailedError:
             expected:<6> but was:<7>
at steal.ArraySumTest.testRun(ArraySumTest.java:31)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at

sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
at

sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

FAILURES!!!
Tests run: 1,  Failures: 1,  Errors: 0
```

The problem lied on the program *ArraySum.java*, on the assignment of variable *rhsStart*, which represents the starting point of the right subarray. While the original program had $lhsSize + 1$ as assignment, the proper expression was $lhsSize + start$ (so we take into account both the relative start of the initial subarray we got, as well as the length of the left sub-subarray). After this fix, the test worked fine on both 2 and 24 cores machines.

Again, above fix and behavior were the same as homonym program from chapter 14.

24 FibTaskTest

24.1 Particular Case (problem)

The problem is calculating the sum of the first n Fibonacci numbers, in parallel.

Note: this exercise is exactly the same as the homonym from chapter 14; we believe the authors of the book made a mistake and ended up with this repetition. Still we are repeating here for the sake of completeness.

24.2 Solution

The solution merely follows the recursive definition of the Fibonacci sequence, whose base cases are 1 (for first and second elements of the sequence). Each recursive call of the function creates a new task (*Future* from java concurrency library). Below the code of the main function:

```java
public Integer call() {
  try {
    if (arg > 2) {
      Future<Integer> left = exec.submit(new FibTask(arg-1));
      Future<Integer> right = exec.submit(new FibTask(arg-2));
      return left.get() + right.get();
    } else {
      return 1;
    }
  } catch (Exception ex) {
    ex.printStackTrace();
    return 1;
  }
}
```

Note that *Future* class represents asynchronous calculations, and that the *get* method is blocking (waits for calculation to be complete); hence we would effectively create a tree of tasks until we reach the base cases and start evaluating bottom up.

24.3 Experiment Description

The experiment merely request the calculation of the sum up to a known number (16), and asserts the expected result (987).

24.4 Observations and Interpretations

The test works as expected in both 2 and 24 cores machines. Below a sample output:

```
[oraadm@gdlaa008 orig]$ junit steal.FibTaskTest
.run

Time: 0.176

OK (1 test)
```

25 MMThreadTest

25.1 Particular Case (problem)

The problem is that of matrix multiplication, but done in parallel.

Note: this exercise is exactly the same as the homonym from chapter 14; we believe the authors of the book made a mistake and ended up with this repetition. Still we are repeating here for the sake of completeness.

25.2 Solution

The solution creates one thread per cell on the result matrix, and for each one it creates a new worker thread; which in turn calculates the cell (i, j) value by computing the dot product of row i of left matrix by column j of right matrix (according to the definition of the matrix product). Below the most relevant sections of the code:

```java
public MMThread(double[][] a, double[][] b) {
  n = a.length;
  this.a = a;
  this.b = b;
  this.c = new double[n][n];
}

void multiply() {
  Worker[][] worker = new Worker[n][n];
  // create one thread per matrix entry
  for (int row = 0; row < n; row++) {
    for (int col = 0; col < n; col++) {
      worker[row][col] = new Worker(row,col);
    }
  }
  ...
}

class Worker extends Thread {
  int row, col;
  Worker(int row, int col) {
    this.row = row; this.col = col;
  }
  public void run() {
    double dotProduct = 0.0;
    for (int i = 0; i < n; i++) {
      dotProduct += a[row][i] * b[i][col];
    }
    c[row][col] = dotProduct;
  }
}
```

25.3 Experiment Description

The test consists of multiplying two identify matrices of dimensions 3×3, and compare the result with one of the operands (it should be identity matrix again).

25.4 Observations and Interpretations

The test runs fine on 2 and 24 core machines, below a sample output:

```
[oraadm@gdlaa008 orig]$ junit steal.MMThreadTest
.run

Time: 0.004

OK (1 test)
```

26 DisBarrierTest

26.1 Particular Case (problem)

The problem we are trying to solve is that of synchronization on a multi-threaded environment; and the particular mechanism we want to use for that is a barrier.

26.2 Solution

The particular type of barrier proposed for this exercise is called a Dissemination Barrier, and it has a nice tree structure. Imagine that in order to implement the barrier, you need to perform "r" rounds; where each rounds means that all the threads involved communicated with other two threads (everyone sent a message to one predefined thread and waited to receive a message from another predefined thread). We use the communication word loosely here, as in a shared memory computer it will simply mean writing or reading from a shared variable.

Now, each round is essentially some sort of permutation; is like you align the n threads in one line, duplicate that line and do a shuffle on it, then you pair its elements with the original line. It turns out (can be proved), that if you do $ceil(log_2(n))$ rounds and on each round k use the formula $(i + 2^k) \bmod n$ for thread communication (where i is the slot associated to each thread on an array), then all the threads can safely assumed that everybody has finished its call to *await* and they can move on. The implementation is compact enough to fit here, so here it goes:

```
public class DisBarrier implements Barrier {
  int size;
  int logSize;
  Node[][] node;
  ThreadLocal<Boolean> mySense;
  ThreadLocal<Integer> myParity;

  public DisBarrier(int capacity) {
    size = capacity;
    logSize = 0;
    while (capacity != 1) {
      this.logSize++;
```

```java
      capacity >>= 1;
    }
    node = new Node[logSize][size];
    for (int r = 0; r < logSize; r++) {
      for (int i = 0; i < size; i++) {
        node[r][i] = new Node();
      }
    }
    int distance = 1;
    for (int r = 0; r < logSize; r++) {
      for (int i = 0; i < size; i++) {
        node[r][i].partner = node[r][(i + distance) % size];
      }
      distance *= 2;
    }
    this.mySense = new ThreadLocal<Boolean>() {
      protected Boolean initialValue() { return true; };
    };
    this.myParity = new ThreadLocal<Integer>() {
      protected Integer initialValue() { return 0; };
    };
  }

  public void await() {
    int parity = myParity.get();
      boolean sense = mySense.get();
      int i = ThreadID.get();
    for (int r = 0; r < logSize; r++) {
      node[r][i].partner.flag[parity] = sense;
      while (node[r][i].flag[parity] != sense) {}
    }
    if (parity == 1) {
      mySense.set(!sense);
    }
    myParity.set(1 - parity);
  }

  private static class Node {
    boolean[] flag = {false, false};  // signal when done
    Node partner;                     // partner node
  }
}
```

26.3 Experiment Description

The test consists in having a shared array of 8 slots, one per thread to be created. Then the test has two phases, both using the barrier; the first phase is to wait all threads to write on their assigned slot on the array and the second stage is to allow all threads to validate that the rest actually finished their writing. These two phases are repeated 8 times.

26.4 Observations and Interpretations

The test hangs time to time, on the 24 cores machine (with 2 cores machine, issue was more difficult to reproduce); the hanging threads have an stack like this:

```
"Thread-5" prio=10 tid=0x00007f1bc0174800 nid=0x8414 runnable [0x00007f1b6bb59000]
java.lang.Thread.State: RUNNABLE
at barrier.DisBarrier.await(DisBarrier.java:59)
at barrier.DisBarrierTest$TestThread.run(DisBarrierTest.java:79)
```

The stack above refers to the following wait cycle from the *await* method:

```
while (node[r][i].flag[parity] != sense) {}
```

The problem then implies that the thread in charge of writing to the flag of the hanging thread, never made the write ... well, that is the appearance but in reality what happened is that the flag variable was not declared as volatile, therefore there is no guarantee about when that new value will be propagated from caches to main memory, and from there to the hanging thread core's cache. Thus, even when the write was actually done the other thread could not see it:

As with other tests from chapter 2, we can solve this issue simply by declaring the flag as volatile:

```
volatile boolean[] flag = {false, false}
```

With above fix the test no longer hangs on both 2 and 24 core machines.

27 SenseBarrierTest

27.1 Particular Case (problem)

The problem we are trying to solve is that of synchronization on a multi-threaded environment; and the particular mechanism we want to use for that is a barrier (same as *DisBarrierTest*).

27.2 Solution

The solution called a "Sense Barrier" is well suited for reusing barrier structures (given that barriers construction, in general, is kind of expensive). The approach introduces the concept of a *sense*; which is basically a boolean flag used for both the whole barrier structure and for each thread. An implementation detail is that the barrier's sense is a shared variable among threads (hence declared *volatile*) and the thread's sense is implemented with a *ThreadLocal* variable.

The main idea behind this solution is that the barrier's sense is initialized to the negation of the threads' senses; and that a call to the *await* method will block all threads to wait for the barrier's sense to be negated. Exception is the last thread to come, which is allowed to actually change the barrier's sense hence unblocking the rest of the threads. Prior leaving the method each thread also negates its own local sense flag (so the contract between barrier and thread senses is kept for next usage). The code is small enough to fit here, and is pasted below:

```
public class SenseBarrier implements Barrier {
  AtomicInteger count;      // how many threads have arrived
  int size;                 // number of threads
  volatile boolean sense;   // object's sense
  ThreadLocal<Boolean> threadSense;

  public SenseBarrier(int n) {
    count = new AtomicInteger(n);
    size = n;
    sense = false;
    threadSense = new ThreadLocal<Boolean>() {
      protected Boolean initialValue() { return !sense; };
    };
  }

  public void await() {
    boolean mySense = threadSense.get();
    int position = count.getAndDecrement();
    if (position == 1) { // I'm last
      count.set(this.size);       // reset counter
      sense = mySense;            // reverse sense
    } else {
      while (sense != mySense) {} // busy-wait
    }
    threadSense.set(!mySense);
  }
}
```

One specially attractive feature of this approach, that we also saw while studying Spin locks in chapter 7, is that the threads are spinning on a local variable (which is expected to be cached on their respective core). One may think that the while expression is using both a local and a global variable, but the global one is likely to be cached for most of the time; and its entry in the cache gets invalidated only when is time to leave the barrier (after last thread changed it), so traffic on the cores' bus only occurs at the end.

27.3 Experiment Description

The test consists in having a shared array of 8 slots, one per thread to be created. Then the test has two phases, both using the barrier; the first phase is to wait all threads to write on their assigned slot on the array and the second stage is to allow all threads to validate that the rest actually finished their writing. These two phases are repeated 8 times.

The test above is basically the same as that described for *DisBarrierTest*.

27.4 Observations and Interpretations

Contrary to *DisBarrierTest*, this test does not hang on 2, nor on 24 core machines. This is because the shared variable *sense* is properly declared as *volatile*. Below a sample successful execution:

```
[oraadm@gdlaa008 orig]$ junit barrier.SenseBarrierTest
.Testing 8 threads, 8 rounds

Time: 0.012

OK (1 test)
```

28 TreeBarrierTest

28.1 Particular Case (problem)

The problem we are trying to solve is that of synchronization on a multi-threaded environment; and the particular mechanism we want to use for that is a barrier (same as other cases from same chapter 17). The subproblem we care about here, is to reduce the contention generated on the last phase of solutions like *SenseBarrierTest*.

28.2 Solution

This solution is a natural extension of the "Sense Barrier" seen for exercise *SenseBarrierTest*; where the barrier as a whole is split into a several smaller sense barriers, and they are placed as the leaves of a tree (each leaf node is setup with same amount of elements, and this size is also called the *radix*). Then inside each leaf node, we will suffer smaller contention than with a single sense barrier (simply cause the number of threads to be unblocked is smaller). A key difference with the single sense barrier approach though, is that the children nodes need to call *await* method on the parent; this effectively means that all calls will eventually reach the root node (and that is the way to know all the threads have reached a consensus, and they can move on). The relevant parts of code are shown below:

```java
public class TreeBarrier implements Barrier {
  int radix;     // tree fan-in
  Node[] leaf;  // array of leaf nodes
  int leaves;    // used to build tree
  ThreadLocal<Boolean> threadSense; // thread-local sense
  public TreeBarrier(int n, int r) {
    radix = r;
    leaves = 0;
    this.leaf = new Node[n / r];
    int depth = 0;
    threadSense = new ThreadLocal<Boolean>() {
      protected Boolean initialValue() { return true; };
    };
    // compute tree depth
    while (n > 1) {
      depth++;
      n = n / r;
    }
    Node root = new Node();
    build(root, depth - 1);
  }

  void build(Node parent, int depth) {
    // are we at a leaf node?
    if (depth == 0) {
      leaf[leaves++] = parent;
    } else {
      for (int i = 0; i < radix; i++) {
        Node child = new Node(parent);
        build(child, depth - 1);
      }
    }
  }

  public void await() {
    int me = ThreadID.get();
    Node myLeaf = leaf[me / radix];
    myLeaf.await();
  }

  private class Node {
    AtomicInteger count;
    Node parent;
    volatile boolean sense;
    // construct root node
    public Node() {
      sense = false;
      parent = null;
      count = new AtomicInteger(radix);
    }
    public Node(Node parent) {
      this();
      this.parent = parent;
    }
    public void await() {
      boolean mySense = threadSense.get();
      int position = count.getAndDecrement();
      if (position == 1) {     // I'm last
        if (parent != null) { // root?
          parent.await();
        }
        count.set(radix);      // reset counter
```

```
          sense = mySense;
        } else {
          while (sense != mySense) {};
        }
        threadSense.set(!mySense);
      }
    }
  }
```

28.3 Experiment Description

The test consists in having a shared array of slots, as big as the number of threads to use. Then the test has two phases, both using the barrier; the first phase is to wait all threads to write on their assigned slot on the array and the second stage is to allow all threads to validate that the rest actually finished their writing. These two phases are repeated 8 times.

The test above is basically the same as that described for *DisBarrierTest* and *SenseBarrierTest*; except that we have a couple of tests instead of one (each one tries a different radix [6] and number of threads; the combinations of these two parameters are $threads = 8$,$radix = 2$ and $threads = 27$,$radix = 3$ respectively.

28.4 Observations and Interpretations

The test runs fine in 2 and 24 core machines, a sample execution is shown below:

```
[oraadm@gdlaa008 orig]$ junit barrier.TreeBarrierTest
.Testing 27 threads, 8 rounds
finished radix 3
.Testing 8 threads, 8 rounds
finished radix 2

Time: 0.023

OK (2 tests)
```

[6]The *radix* parameter controls the size of the leaf nodes, which represent smaller sense barriers

29 ListTest (ofree)

29.1 Particular Case (problem)

The problem we try to solve here is how to implement a concurrent list by using software transactional (*STM*) memory.

29.2 Solution

The solution uses the "obstruction-free" flavour of the *STM*, where the building blocks of the list use the so called *FreeObject*. Each one of these free objects has three logical fields: an *owner* field to keep track of what thread is using it, *oldVersion* which is the object state prior beginning of transaction, and *newVersion* version that reflects current transaction updates. The possible values for *oldVersion* and *newVersion* fields are *COMMITTED*, *ABORTED* and *ABORTED*.

Each time a transaction *A* accesses one of these objects for the first time, it "opens" that object (possibly reseting the logical fields mentioned above). If we call *B* the previous owner transaction, the following rules govern this solution (quoted from the book):

- If *B* was *COMMITTED*, then the new version is current. A installs itself as the object's current owner, sets the old version to the prior new version, and the new version to a copy of the prior new version (if the call is a setter), or to the new version itself (if the call is a getter).

- Symmetrically, if *B* was *ABORTED*, then the old version is current. A installs itself as the objects current owner, sets the old version to the prior old version, and the new version to a copy of the prior old version (if the call is a setter), or to the old version itself (if the call is a getter).

- If *B* is still *ACTIVE*, then *A* and *B* conflict, so *A* consults the contention manager for advice whether to abort *B*, or to pause, giving *B* a chance to finish. One transaction aborts another by successfully calling *compareAndSet()* to change the victim's status to *ABORTED*.

Contrary to other chapters, the transactional memory one implies much more code to consider; so we did not consider relevant to paste it all here. The explanation given above hopefully suffices for our purposes.

29.3 Experiment Description

The test program populates serially the list protected by STM (initial size of 1024), and then creates certain amount of threads to perform indefinitely, alternating random insertions and deletions. The main test thread interrupts them and performs sanity checks. This test is performed with 1 (sequential) and 32 threads (parallel).

29.4 Observations and Interpretations

The test using a single thread (sequential), gives the following null pointer exception on the 24 cores machine:

```
.TestSequential
TinyTM.exceptions.PanicException: java.lang.NullPointerException
at TinyTM.TThread.doIt(TThread.java:51)
at TinyTM.list.ofree.ListTest$TestThread.run(ListTest.java:117)
Caused by: java.lang.NullPointerException
at TinyTM.list.ofree.TNode.getNext(TNode.java:54)
at TinyTM.list.ofree.List.add(List.java:40)
at TinyTM.list.ofree.ListTest$TestThread$1.call(ListTest.java:119)
at TinyTM.list.ofree.ListTest$TestThread$1.call(ListTest.java:117)
at TinyTM.TThread.doIt(TThread.java:41)
... 1 more
inserts: 0, removes: 0, missed: 0, delta: 0
commits: 0, aborts: 0
```

While debugging the sequential exception above, we found that in method *FreeObject:openRead*, a new location was never getting a *newVersion* value after its construction:

```
Locator newLocator = new Locator();
... more code ...
return newLocator.newVersion <<== NULL !!!
```

As possible solutions for this problem, we realized that either we could use constructor that takes *newVersion* as argument or we set it like with method *openWrite*. We opted for the first option, using existing *locator* object:

```
Locator newLocator = new Locator(locator.newVersion);
```

With above fix the sequential test no longer presented problems on 2 nor on 24 cores. But the parallel one was still failing with assertion errors like the ones below:

```
Testcase: testParallel(TinyTM.list.ofree.ListTest): FAILED
Bad size, expected:<1518> but was:<1502>
junit.framework.AssertionFailedError: Bad size, expected:<1518> but
                was:<1502>
at TinyTM.list.ofree.ListTest.sanityCheck(ListTest.java:196)
at TinyTM.list.ofree.ListTest.testParallel(ListTest.java:84)
```

Due time constraints, we could not dig further into the reasons for the above error; but professor acknowledged it was ok (after all, we covered the entire set of programs from the book).