Multiprocessor Programming Course

Dario Bahena Tapia - Ricardo Zavaleta Vazquez - Isai Barajas Cicourel

October 2015

Contents

Chapter 1

Mutual Exclusion

1.1 LockFreeQueueTest

1.1.1 Particular Case (problem)

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

1.1.2 Solution

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

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

1.1.3 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.

1.1.4 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.

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.

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.

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.

1.1.5 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).

1.2 Peterson Algorithm

1.2.1 Particular Case

In this experiment, the particular case we are trying to study is again mutual exclusion where two threads try to use a shared resource.

We are trying to do so in such a way that there is no starvation.

1.2.2 Solution

During the lecture we discussed that Peterson algorithm takes the best of other two algorithms (which we called the LockOne and the LockTwo) to create a better one. The LockOne class that we discussed used an approach where mutual exclusion was attempted using a flag which indicates that a given thread has the intention of using a resource. We concluded that it satisfies the mutual exclusion but it can potentially dead lock unless executions are not interleaved.

On the other hand, the LockTwo class achieved mutual exclusion by forcing the threads let the other one execute first. However, this approach also had dead lock problems when one thread ran completely before the other one. So we observe that both algorithms complement each other.

The Peterson algorithm combines these two methods to achieve a starvation-free algorithm. It uses both the *flags* approach and the *victimization* approach.

Here we show the *lock()* and *unlock()* methods:

```
1   private boolean[] flag = new boolean[2];
2   private int victim;
3
4   public void lock() {
5     int i = ThreadID.get();
6     int j = 1-i;
7     flag[i] = true;
8     victim  = i;
9     while (flag[j] && victim == i) {}; // spin
10  }
11
12  public void unlock() {
13    int i = ThreadID.get();
14    flag[i] = false;
15  }
```

1.2.3 Experiment Description

Now let's discuss how the proposed test case excersices the Peterson algorithm.

The test creates 2 threads that need to be coordinate in order to increment a common counter. Both threads have to cooperate to increase this counter from 0 to 1024. Each of the threads will increase by one the counter 512 times. The expected result is that regardless of the order in which each thread executes the increment, at the end the counter must stay at 1024. If that is not the case, then it means that mutual exclusion did not work.

1.2.4 Sample Results

This experiment was successfull in all tries. This is the output of the test:

```
.parallel

Time: 0.01

OK (1 test)
```

1.2.5 Interpretation

The results in the proposed system were all OK. In all cases, the threads were able to cooperate to increase the counter to 1024.

One thing that is worth mentioning is that this algorihtm has a limitation. The limitation is that it only works for two threads. In a later experiment we will discover other algorithms that allow the coordination of more that two threads.

1.3 Bakery Test

1.3.1 Particular Case

In this experiment, we are dealing with the problem of mutual exclusion with a number of participants > 2.

We have already seen that Peterson Algorithm works for two threads. It garantees mutual exclusion and is starvation free. Therefore, the algorithm is deadlock free. Now let us focus in an algorithm that can be used to coordinate more threads.

1.3.2 Solution

The algorithm that we will play with is called *Bakery*. Its name comes from the fact that it is similar to the protocol used in bakeries where one enters the store and picks a number that indicates the order in which each client will be attended.

In the algorithm, the fact of entering the store is done by setting a flag. After that, the thread calculates the next number in the machine.

The algorithm then chooses the next number to be attended. When that happens, the thread is allowed to enter the critical section.

One thing that we ougth to mention is that threads can have the same number assigned. If that is the case, the next in the line is decided using a lexicographical ordering of the threads ids.

Here we show the interesting methods used in this algorithm:

```
1   public void lock() {
2     int me = ThreadID.get();
3     flag[me]  = true;
4     int max = Label.max(label);
5     label[me] = new Label(max + 1);
6     while (conflict(me)) {};  // spin
7   }
8
9   public void unlock() {
10    flag[ThreadID.get()] = false;
11  }
12
13  private boolean conflict(int me) {
14    for (int i = 0; i < label.length; i++) {
```

```
15        if (i != me && flag[i] && label[me].compareTo(label[i]) < 0) {
16          return true;
17        }
18      }
19      return false;
20    }
21
22    static class Label implements Comparable<Label> {
23      int counter;
24      int id;
25      Label() {
26        counter = 0;
27        id = ThreadID.get();
28      }
29      Label(int c) {
30        counter = c;
31        id = ThreadID.get();
32      }
33      static int max(Label[] labels) {
34        int c = 0;
35        for (Label label : labels) {
36          c = Math.max(c, label.counter);
37        }
38        return c;
39      }
40
41      public int compareTo(Bakery.Label other) {
42        if (this.counter < other.counter
43            || (this.counter == other.counter && this.id < other.id)) {
44          return -1;
45        } else if (this.counter > other.counter) {
46          return 1;
47        } else {
48          return 0;
49        }
50      }
51    }
```

In the *lock()* method, we observe that the thread signals its intention of accessing the critical section by setting its flag to true. After that, the thread picks its number and then it sits and waits for its turn. The thread has to wait if another thread which announced its intention to use the resource has a smaller label.

1.3.3 Experiment Description

Now let us explain how the experiments work. In this case we have 8 threads increasing a counter from 0 to 1024. Each thread will increase the counter 128 times. At the end, the counter must remain in 1024. If that is not the case, then there is a problem with the mutual exclusion algorithm.

1.3.4 Sample Results

In this exercise we observed that in the machine that we have been using, the test for the Bakery algorithm fails consistently with the following error:

```
.F
Time: 0.019
There was 1 failure:
1) testParallel(mutex.BakeryTest)junit.framework.AssertionFailedError:
expected:<1019> but was:<1024>
        at mutex.BakeryTest.testParallel(BakeryTest.java:47)
        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
```

1.3.5 Interpretation

Let us elaborate on the meaning of the error that we got. The assertion failure says that at the end of the algorithm, our counter didn't reach the expected 1024. This problem might indicate that our counter wasn't actually being protected by the lock because it looks like multiple threads found the counter with a value of i and increased it to $i + 1$. This is wrong.

After investigating the problem in more detail, we found out that the problem in this code has to do with the way the *label* array is accessed and modified. For example, in the lock method, multiple threads can reach the point where the max is calculated and hence these mutiple threads can take the same turn. The algorithm takes care of this problem by introducing a lexicographical ordering in the comparison: if two threads have the same number, the decision is made based on the $threadID$.

Also, as explained in the textbook, we cannot be sure that we have a correct memory consistency: it is possible that the processor modified the order

in which operations in the *lock()* method are executed. It is also possible that the processor decided to execute an instruction before another.

1.3.6 Proposed solution

The following fragment of code shows one possible way of fixing the code:

```
1   public void lock() {
2     int me = ThreadID.get();
3     flag[me] = true;
4     synchronized (label){
5       int max = Label.max(label);
6       label[me] = new Label(max + 1);
7       while (conflict(me)) {}; // spin
8     }
9   }
```

Note that this solution is not quite satisfactory as we are basically using a Java mechanism to ensure synchronized access in an algorithm that promises to provide synchronized access.

In any case, let us show the output of the program after this fix:

```
.
Time: 0.015

OK (1 test)
```

After the fix the results in the proposed system were all OK. In all cases, the threads were able to cooperate to increase the counter to 1024.

One thing that is worth mentioning is that unlike the Peterson Algorithm, for example, the Bakery algorithm is able to coordinate multiple threads (at least in theory). This fact was shown in the test case were 8 threads were coordinated to increase the counter.

Chapter 2

Foundations of Shared Memory

2.1 Safe Boolean MRSW Register

2.1.1 Particular Case

In class we saw the difference between Safe, Regular and Atomic Registers. In this excersice we are experimenting with a Safe Boolean Register. In particular, we are trying a register that allows one single writer whose writen value can be examined by multiple readers. We need to remember that Safe registers are those that :

- If a read does not overlap with a write, then the read returns the last written value
- If a read overlaps with a write, then the read can return any value in the domain of the register's values

2.1.2 Solution

One way to achive this behaviour is using a SRSW Safe register. The code that is provided with the book does this by having a boolean SRSW register per thread, in other words, it is an array of boolean values. When doing a write, the writer iterates over this array and writes the new value. The readers simply access its assigned slot in the array (based on the thread id) and return the stored value. The code for this type of register is shown next:

```
1   public Boolean read() {
2     return s_table[ThreadID.get()];
3   }
4
5   public void write(Boolean x) {
6     for (int i = 0; i < s_table.length; i++)
7       s_table[i] = x;
8   }
```

Please note how this type of register does not make use of any synchronization mechanism.

2.1.3 Experiment Description

This program provides two test cases. The first one is a sequential test and the second one is a parallel test. The former simply calls a write and then

a read from the same thread. This is not rocket science. The reader must retrieve what the writer wrote.

The second test case is slightly more interesting. The writer writes first one value and then another one. After that, 8 reader threads are started and they read the last written value. According to the rules of a safe register, all threads must read the same value because the reads and the writes did not overlap.

2.1.4 Sample Results

We found out that the tests failed ocassionally. The manifestation of the failure is a

indexOutOfBoundsException as shown above.

```
[oraadm@gdlaa008 Register]$ junit register.SafeBooleanMRSWRegisterTest
.sequential read and write
.parallel read
Exception in thread "Thread-7" java.lang.ArrayIndexOutOfBoundsException: 8
        at register.SafeBooleanMRSWRegister.read(SafeBooleanMRSWRegister.java:26)
        at register.SafeBooleanMRSWRegisterTest$ReadThread.run(SafeBooleanMRSWRegisterTest.java:62)

Time: 0.008

OK (2 tests)
```

So, the way to fix this problem was to make sure that in each test case the threads ids are reset calling the static method *ThreadID.reset*(). With this hack, the tests passed.

```
1   /**
2    * Sequential calls.
3    */
4   public void testSequential() {
5     ThreadID.reset(); // We have to reset the thread ids
6     System.out.println("sequential read and write");
7     instance.write(true);
8     boolean result = instance.read();
9     assertEquals(result, true);
10  }
11
12  /**
13   * Parallel reads
14   */
15  public void testParallel() throws Exception {
16    ThreadID.reset(); // We have to reset the thread ids
```

```
17        instance.write(false);
18        instance.write(true)
19        System.out.println("parallel read"
20        for (int i = 0; i < THREADS; i++) {
21          thread[i] = new ReadThread(
22        }
23        for (int i = 0; i < THREADS; i ++) {
24          thread[i].start();
25        }
26        for (int i = 0; i < THREADS; i ++) {
27          thread[i].join();
28        }
29      }
```

And here is the resulting execution after this fix:

```
[oraadm@gdlaa008 Register]$ junit register.SafeBooleanMRSWRegisterTest
.sequential read and write
.parallel read

Time: 0.004

OK (2 tests)
```

2.1.5 Interpretation

In this experiment we saw a way of implementing Safe Multi-Reader Single-Writer registers. These registers were able of storing boolean values.

Unfortunately, none of the test cases exercises the case where we have concurrent writes and reads. Hence, this experiment only exercised one of the two rules of a safe register.

2.2 RegMRSWRegisterTest

2.2.1 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.

2.2.2 Solution

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

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

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

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

2.2.3 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.

2.2.4 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)
```

2.3 Atomic MRSW Register

2.3.1 Particular Case

The characteristics of this type of register are:

- It should never happen that $R^i \rightarrow W^i$
- It should never happen that for any j, $W^i \rightarrow W^j \rightarrow R^j$
- If $R^i \rightarrow R^j$ then $i \leq j$

The book proposes to do this by using multiple SRSW registers.

2.3.2 Solution

The algorithm then requires a 2 dimensional table. When a writer decides to update the register, it has to update the values in cells $A[i][i]$, where i is the thread id. Apart from writing the value, it has to also update the cell with a timestamp.

In the other hand, when a reader wants to read from the register, it checks the timestamp of the cell $A[i][i]$. After that, it has to check other cells in the same column (ie, cells $A[x][i]$) to see if there has been an update in between. That is done by comparing the timestamps. If there is a newer timestamp, the reader has then to update all cells in its row (ie $A[i][x]$). This is the way to indicate subsequent readers which version of the value has the previous reader retrieved.

It is easy to see that the first property is satisfied because there is no way to read a future value. In order to read a value, this value has to be written before.

The second property is also satisfied because when a reader reads the value, it reads the one with the most recent timestamp by scaning the timestamps in its corresponding column. This reader also helps subsequent readers by updating all cells in its row. Forcing property 3 to be satisfied as well.

The *read()* and *write()* methods are shown next:

```
1   public synchronized T read() {
2     int me = ThreadID.get();
3     StampedValue<T> value = a_table[me][me];
4     for (int i = 0; i < a_table.length; i++) {
5       value = StampedValue.max(value, a_table[i][me]);
```

```
6       }
7       for (int i = 0; i < a_table.length; i++) {
8         a_table[me][i] = value;
9       }
10      return value.value;
11    }
12
13    public synchronized void write(T v) {
14      int me = ThreadID.get();
15      long stamp  = lastStamp.get() + 1;
16      lastStamp.set(stamp); // remember for next time
17      StampedValue<T> value = new StampedValue<T>(stamp, v);
18      for (int i = 0; i < a_table.length; i++) {
19        a_table[me][i] = value;
20      }
21    }
```

2.3.3 Experiment Description

The two test cases presented demonstrates the same as previous examples so we will not describe the experiment further.

2.3.4 Sample Results

For this test, the results were always successful. Here is the sample output:

```
.sequential read and write
.parallel read

Time: 0.004

OK (2 tests)
```

2.3.5 Interpretation

In this experiment we observed a way of implementing MRSW registers. We saw that, at least in the processor we used, the algorithm seems to work correctly.

It would have been great, though, to also have test cases that demonstrate that this particular register satisfies property 3 which is the one that makes it different from the Safe version.

2.4 AtomicMRMWRegisterTest

2.4.1 Particular Case (problem)

The problem we are trying to solve here, is that of implementing an atomic multi-valued [1] 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!).

2.4.2 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:

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

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

2.4.3 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.

2.4.4 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)
```

2.5 SimpleSnapshotTest

2.5.1 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.

2.5.2 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:

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

2.5.3 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).

2.5.4 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)
```

2.6 Wait-Free Snapshots

2.6.1 Particular Case

In this experiment we are dealing with the problem of creating an algorithm to create a Snapshot of a set of Register in such a way that the algorithm garantees a Wait-Free operation.

2.6.2 Solution

The purpose of a Wait-Free snapshot is to overcome the problems of the Simple snapshot presented before. Let us remember that such snapshot executes sucesive collect() operations. Once it achieves a *clean double collect*, it returns the snapshot. Otherwise, it uses the following idea:

When a double collect fails, it is because a update interfered. That means that the updater could take a snapshot right before its update and other threads could use it as their snapshot too.

Since this algorithm is a bit more complex, let us explain in more detail how the implementation works.

```
1   public WFSnapshot(int capacity, T init) {
2     a_table = (StampedSnap<T>[]) new StampedSnap[capacity];
3     for (int i = 0; i < a_table.length; i++) {
4       a_table[i] = new StampedSnap<T>(init);
5     }
6   }
```

Firstly, the algorithm uses an array to store the information. The size of the array is, in this case, n, where n is the number of threads. The way in which this array is constructed is as an array of *StampedSnap*s. A StampedSnap is internally an array whose elements are of type T. A StampedSnap is a subclass of StampedValue which has 3 fields:

- *stamp*, which is a counter
- *owner*, which contains the thread id
- *value*, the actual value in the register

Now let us look at an auxiliar operation which is *collect()*. This operation returns a copy of the array. Observe that the method creates a new array of StampedSnap values and then iterates the existing array to copy each element.

```
1   private StampedSnap<T>[] collect() {
2     StampedSnap<T>[] copy = (StampedSnap<T>[]) new StampedSnap[a_table.length];
3     for (int j = 0; j < a_table.length; j++)
4       copy[j] = a_table[j];
5     return copy;
6   }
```

The next method to understand is *scan()*. The method first creates a backup or an old copy of the array. Then it takes a second collect. It then compares both copies. If both copies are equal, then it means that we have a double clean collect. And the result of the scan operation is any of the copies just as in the Sequentian Snapshot case.

On the other hand, if the copies are different, then there are two cases. For this, we have a counter that indicates how many times, the copies have been compared and found different. If it is the first time, then we take another collect. If in the second collect, the copies are equal, then we fall in the case described in the previous paragraph. If the copies are different again, then it means that we can take the old copy as a result of the scan.

The justification of this is that if a thread A sees a thread B move twice while it is performing repeated collects, then B executed a complete *update()* call within the interval of A's scan, and so it is correct for A to use B's snapshot.

```
1   public T[] scan() {
2     StampedSnap<T>[] oldCopy;
3     StampedSnap<T>[] newCopy;
4     boolean[] moved = new boolean[a_table.length];
5     oldCopy = collect();
6     collect: while (true) {
7       newCopy = collect();
8       for (int j = 0; j < a_table.length; j++) {
9         // did any thread move?
10        if (oldCopy[j].stamp != newCopy[j].stamp) {
11          if (moved[j]) {        // second move
12            return oldCopy[j].snap;
```

```
13            } else {
14              moved[j] = true;
15              oldCopy = newCopy;
16              continue collect;
17            }
18          }
19        }
20        // clean collect
21        T[] result = (T[]) new Object[a_table.length];
22        for (int j = 0; j < a_table.length; j++)
23          result[j] = newCopy[j].value;
24        return result;
25      }
26    }
```

Finally, we have the *update()* operation which simply updates the register with the new stamp.

```
1   public void update(T value) {
2     int me = ThreadID.get();
3     T[] snap = this.scan();
4     StampedSnap<T> oldValue = a_table[me];
5     StampedSnap<T> newValue =
6       new StampedSnap<T>(oldValue.stamp+1, value, snap);
7     a_table[me] = newValue;
8   }
```

2.6.3 Experiment Description

In this experiment, the test looks a bit different. Let us explain how. In this case we have two experiments:

1. testSequential. We spawn a new thread that updates the register with a value of *FIRST* and right after that, it scans the value of the register. The expected result is that this read retrieves the value of *FIRST*

2. testParallel. We spawn *THREADS* number of threads (in our case it is 2). Each of them will first update its register with a value of *FIRST* and then with a value of *SECOND*. Each thread then updates a position in a matrix of size *THREADSxTHREADS* with the result

of doing a *scan*() operation. At the end we compare consecutive rows in the matrix. The comparison is done entry by entry and we should see that for some entries the first entry is greater and for some others it is smaller than the second. What we must see is that all entries are equal and we allow the first to sometimes be either greater or smaller than the second one.

2.6.4 Sample Results

For this test, we saw that in every try, both test cases passed. Here is the output:

```
[oraadm@gdlaa008 Register]$ junit register.WFSnapshotTest
.sequential
.parallel

Time: 0.003

OK (2 tests)
```

2.6.5 Interpretation

In this experiment we were able to observe how a Wait-Free Snapshot can be constructed based on the idea of *clean double collects* and on the idea of using the updater's collect when the clean double collect method fails.

Chapter 3

Spin Locks and Contention

3.1 Problem Definition

The concept of Spin Lock comes from the following problem: *In the case where a thread tries to adquire a lock on a resource and it fails, what should be done?*

There are at least to ways to react to this problem:

1. Keep trying getting the lock. In this case the lock is called a Spin Lock. The thread is then said to be in a busy wait.

2. Suspend the execution of the current thread and let the others do some work. This action is called *blocking*

In this chapter we will study different ways in which spin locks can be implemented.

3.2 Test-And-Set Locks

3.2.1 Particular Case

The particular case we are trying to solve is that of mutual exclusion with two participants. The problem with previous approaches for solving this problem (e.g. Peterson Algorithm) is that in the real world the assumption of having a "sequential consistent memory" and a "program order guarantee among reads-writes by a given thread" are not true.

There are at least two reasons for that. The first one is that compilers, in order to improve performance of the program, can in some cases reorder certain operations . As the book clearly mentions, most programming languages preserve execution order that affect individual variables, but not across variables. This means that reads and/or writes of different variables can end up in a different order that the one appearing in the source code once the program is compiled or executed. This can clearly affect our Peterson Algorithm.

The second reason that has to do with the contradiction of sequential consistent memory is due to the hardware design which in most multiprocessor systems allow writes to be written to store buffers. These buffers are written to memory only when needed. Hence, race conditions arise, where one variable might be read before another because the latter was in a store buffer.

The generic way to solve this problem is by using instructions called *memory barriers* or *memory fences*. These instructions are able to prevent reordering operations due to write buffering. The idea is that these instructions can be called in specific parts of the code to guarantee our proposed order of execution.

The problem in this section is *how can we implement an actually working mutual exclusion algorithm with consensus number of two using memory barrier*?

3.2.2 Solution

One solution for this problem is using a synchronization instruction that includes memory barriers. For example, we can use the $testAndSet()$ operation which has a consensus number of two and which atomically stores *true* in a variable an returns the previously stored value.

To design an algorithm using this instruction the idea is the following. A value of false is used to indicate that a lock is free. Then, we store a value of true to indicate that we are using the lock. Once we are done, we can set the value back to false to allow other threads take the lock.

In the java code, the *testAndSet*() is done with the *getAndSet*() method. And setting the value to false is achieved with the method *set*(). The code for this is show next:

```
1  public class TASLock implements Lock {
2    AtomicBoolean state = new AtomicBoolean(false);
3    public void lock() {
4      while (state.getAndSet(true)) {} // spin
5    }
6    public void unlock() {
7      state.set(false);
8    }
9    ...
10 }
```

Observe how calling the *lock* consists simply of setting the value of the *state* variable to true by using a memory barrier operation. The same technique is used for the *unlock()* method.

3.2.3 Experiment Description

To demonstrate that this implementation works, the test that was provided does the following:

1. Initiate a shared counter with a value of 0
2. Start 8 threads
3. Each thread has to increment the counter by one 1024 times
4. At the end, the counter must hold a value of $8 * 1024$

3.2.4 Sample Results

For this test, we saw that in every try, it always passed. Here is the output:

```
[oraadm@gdlaa008 Spin]$ junit spin.TASLockTest
.parallel

Time: 0.026

OK (1 test)
```

3.2.5 Interpretation

It was shown that the algorithm seems to work and allows synchronization of two threads (consensus number was two) to access a shared resource. The way of achieving this is through the use of the memory barrier functions *getAndSet()* and *set()* of the *AtomicBoolean* class.

3.3 TTASLockTest

3.3.1 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.

3.3.2 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:

```
1  public class TTASLock implements Lock {
2    AtomicBoolean state = new AtomicBoolean(false);
3    public void lock() {
4      while (true) {
5        while (state.get()) {};  // spin
6        if (!state.getAndSet(true))
7          return;
8      }
9    }
10   public void unlock() {
11     state.set(false);
12   }
13   ...
14 }
```

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).

3.3.3 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.

3.3.4 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).

3.4 BackoffLockTest

3.4.1 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 subsubproblem here is how to reduce the contention out of the *TTASLock*.

3.4.2 Solution

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

```
1  public class TTASLock implements Lock {
2    AtomicBoolean state = new AtomicBoolean(false);
3    public void lock() {
4      while (true) {
5        while (state.get()) {}; // spin
6        if (!state.getAndSet(true))
7          return;
8      }
9    }
10   public void unlock() {
11     state.set(false);
12   }
13   ...
14 }
```

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 [1]. 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):

```
1  public class BackoffLock implements Lock {
2    private Backoff backoff;
3    private AtomicBoolean state = new AtomicBoolean(false);
4    private static final int MIN_DELAY = 32;
5    private static final int MAX_DELAY = 1024;
6    public void lock() {
7      Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);
8      while (true) {
9        while (state.get()) {};    // spin
10       if (!state.getAndSet(true)) { // try to acquire lock
11         return;
12       } else {                   // backoff on failure
13         try {
14           backoff.backoff();
15         } catch (InterruptedException ex) {
16         }
17       }
18     }
19   }
20   public void unlock() {
21     state.set(false);
22   }
23   ...
24 }
25
26 public class Backoff {
27   ...
28
29   /**
30    * Backoff for random duration.
31    * @throws java.lang.InterruptedException
```

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

```
32     */
33    public void backoff() throws InterruptedException {
34      int delay = random.nextInt(limit);
35      if (limit < maxDelay) { // double limit if less than max
36        limit = 2 * limit;
37      }
38      Thread.sleep(delay);
39    }
40  }
```

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).

3.4.3 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).

3.4.4 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)
```

3.5 CLH Queue Lock

3.5.1 Particular Case

Using queue locks solve two problems that appear in simpler locking protocols. Firstly, by using a queue, cache-coherence traffic is reduced because each thread spin on a different memory location; and secondly, by using a queue, threads can know whether their turn has come by inspecting the status of their predecesor in the queue.

The particular case that we want to study in this experiment is *How can we implemente a space efficient Queue Lock?*

3.5.2 Solution

One solution for this problem is given by the CLHLock protocol. The algorithm that implement this protocol need two fields local to the thread: A pointer to the current node and a pointer to the previous node. The queue is constructed by having the pointer *previous* pointing to the previous node's *current* field. This field contains a boolean value which is false when the thread releases the lock. When this is the case, the next thread is allowed to acquire the lock.

3.5.3 Experiment Description

To demonstrate that this implementation works, the test that was provided does the following:

1. Initiate a shared counter with a value of 0
2. Start 8 threads
3. Each thread has to increment the counter by one 1024 times
4. At the end, the counter must hold a value of $8 * 1024$

These are the details of the system we used to run the experiments:

- Processor: Intel Core i5 @2.5 GHz. 2 Cores.
- L2 Cache per Core: 256 KB

- L3 Cache: 3 MB
- System Memory: 16 GB

3.5.4 Sample Results

For this test, we saw that in every try, it always passed.

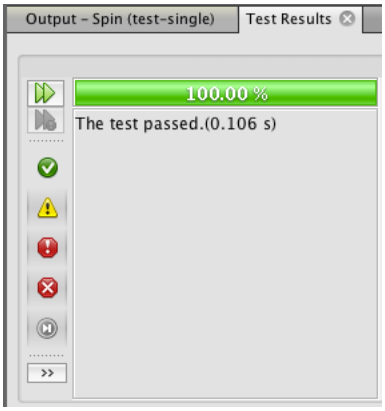

Figure 3.1: Successful execution of the tests for CLH Queue Lock test

```
compile-test-single:
Testsuite: spin.CLHLockTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.106 sec

test-single:
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 3.2: Successful execution of the tests for CLH Queue Lock test

3.5.5 Interpretation

It was shown that the algorithm seems to work and allows synchronization of Multiple threads trying to access a shared variable.

One important thing about this algorith is that threads spin checking for different memory addresses which avoids invalidation of cached copies. Another important advantage is that it requieres a limited number of memory and also provides first-in-first-out fairness.

The case were this algorithm is not good is when, in a NUMA architecture, the *previous* pointer points to a remote location. In that case, the performance of the algorithm degradates.

3.6 MCSLockTest

3.6.1 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.

3.6.2 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 [2]. 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:

```
1  public class MCSLock implements Lock {
2    AtomicReference<QNode> queue;
3    ThreadLocal<QNode> myNode;
4    public MCSLock() {
5      queue = new AtomicReference<QNode>(null);
6      // initialize thread-local variable
7      myNode = new ThreadLocal<QNode>() {
8        protected QNode initialValue() {
9          return new QNode();
10       }
11     };
12   }
13   public void lock() {
14     QNode qnode = myNode.get();
15     QNode pred = queue.getAndSet(qnode);
16     if (pred != null) {
17       qnode.locked = true;
```

[2] 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.

```
18          pred.next = qnode;
19          while (qnode.locked) {}      // spin
20        }
21      }
22      public void unlock() {
23        QNode qnode = myNode.get();
24        if (qnode.next == null) {
25          if (queue.compareAndSet(qnode, null))
26            return;
27          while (qnode.next == null) {} // spin
28        }
29        qnode.next.locked = false;
30        qnode.next = null;
31      }
32      ...
33      static class QNode {      // Queue node inner class
34        boolean locked = false;
35        QNode   next = null;
36      }
37  }
```

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.

3.6.3 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.

3.6.4 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):

```
1        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:

```
1       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*).

3.7 CompositeLockTest

3.7.1 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.

3.7.2 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.

3.7.3 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.

3.7.4 Observations and Interpretations

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

```
.
Time: 0.066

OK (1 test)
```

3.8 Hierarchical Backoff Lock

3.8.1 Particular Case

In the following exercises we will be dealing with the idea of Hierarchical locks. The motivation of this type of locks are the architecture of modern cache-coherent computers where processors are organized as clusters. Communication between processors is then categorized in intra-cluster comunication and inter-cluster comunication. The former is faster and cheaper than the latter.

Given the following background, the idea is to build locking algorithms that work efficiently in these architectures. The problem is then: *how can we create locking algorithm for this type of systems?*

3.8.2 Solution

The first algorithm that we are going to study is the Hierarchical Backoff Lock. This algorithm uses the *test-and-test-and-set* lock that we presented before. If a thread from a cluster releases a lock, then it is more likely that another thread from the same cluster acquires it. This strategy reduces the overal time needed to switch the lock ownership. Let us take a quick look at the source code in java:

```
1   public void lock() {
2     int myCluster = ThreadID.getCluster();
3     Backoff localBackoff =
4         new Backoff(LOCAL_MIN_DELAY, LOCAL_MAX_DELAY);
5     Backoff remoteBackoff =
6         new Backoff(REMOTE_MIN_DELAY, REMOTE_MAX_DELAY);
7     while (true) {
8       if (state.compareAndSet(FREE, myCluster)) {
9         return;
10      }
11      int lockState = state.get();
12      try {
13        if (lockState == myCluster) {
14          localBackoff.backoff();
15        } else {
16          remoteBackoff.backoff();
17        }
18      } catch (InterruptedException ex) {
```

```
19          }
20        }
21      }
22
23      public void unlock() {
24        state.set(FREE);
25      }
```

For locking, we create two backoff objects which are the ones that have a method to perform the backoff for a random period. Notice that we distinguish between two cases: a local backoff and a remote backoff. The delays are as follows:

```
1     private static final int LOCAL_MIN_DELAY = 8;
2     private static final int LOCAL_MAX_DELAY = 256;
3     private static final int REMOTE_MIN_DELAY = 256;
4     private static final int REMOTE_MAX_DELAY = 1024;
```

A remote delay is longer than a local delay. Then we perform a *compareAndSet()* operation. If the state of the lock is free, then we set its value to the value of our cluster. That indicates that we are holding the lock now.

If the state was not free, then we perform a backoff. If the lock is being hold by another thread in the cluster, then we do a short backoff. Otherwise, we do a long backoff.

The unlock method simply sets the state to free.

3.8.3 Experiment Description

To demonstrate that this implementation works, the test that was provided does the following:

1. Initiate a shared counter with a value of 0
2. Start 8 threads
3. Each thread has to increment the counter by one 1024 times
4. At the end, the counter must hold a value of $8 * 1024$

3.8.4 Sample Results

For this test, we saw that in every try, it always passed. Here is the output:

```
[oraadm@gdlaa008 Spin]$ junit spin.HBOLockTest
.
Time: 0.262

OK (1 test)
```

3.8.5 Interpretation

In this experimetn we discovered how to implement a locking algorithm in a cache-coherent architecture by performing backoffs of different delays depending on whether the thread holding the lock is in the same or in a different cluster of CPUs.

One problem that we can foresee though, is that this algorithm could cause starvation of remote threads because the algorithms give more priority to local threads. This is not a desirable property. In the next algorithm we shall see a way of overcoming this problem

Chapter 4

Monitors and Blocking Synchronization

4.1 QueueTest

4.1.1 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 [1] capabilities offered by Java language; when compared to merely using the *synchronized* feature.

4.1.2 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

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

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

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

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

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

4.1.3 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).

4.1.4 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)
```

4.2 Count Down Latch

4.2.1 Particular Case

The purpose of this experiment to our eyes is to demonstrate how monitor locks and conditions are used. Let us remember both concepts before going further.

The concept of monitor consists on encapsulating three components: data, methods and synchronization mechanism. This allows to have our datastructures (or classes) take care of synchronization instead of overwhelming the user of the API with this task.

Now, the concept of *conditions* comes into play because with them we can signal threads about an event. For example, instead of having threads spinning waiting for a value to become false, we can instead signal threads and let them know that they can now re-check for their locking condition.

4.2.2 Solution

The java interface for *Lock* suggests two methods to implement the protocol drafter above. First, there is a method *await()* which basically asks the thread wait till it is signaled about an event. The accompaning method *signalAll* achieves this latter task.

4.2.3 Experiment Description

This experiment is a bit different from others showed before. The idea in this experiment is to show how the methods *await()* and *signalAll()* are combined in a working program.

So, the idea of the test is the following. Initially we have 8 threads. We start them as usual but we do not allow them to procced normally. Instead, we will ask them to wait for a signal. Internally, what we do is decrement a shared variable. When this variable becomes 0, the signal is triggered. Initially this signal is 1, meaning that we only need to decrement the variable once.

After that, each thread will started and wait for the next signal. The variable that controlls this other signal is initially set to 8. Each thread will be in charge of decrementing this variable by one. Once it becomes 0, the second signal will be triggered announcing that our test must finish.

These are the details of the system we used to run the experiments:

- Processor: Intel Core i5 @2.5 GHz. 2 Cores.
- L2 Cache per Core: 256 KB
- L3 Cache: 3 MB
- System Memory: 16 GB

4.2.4 Sample Results

For this test, we saw that in every try, it always passed.

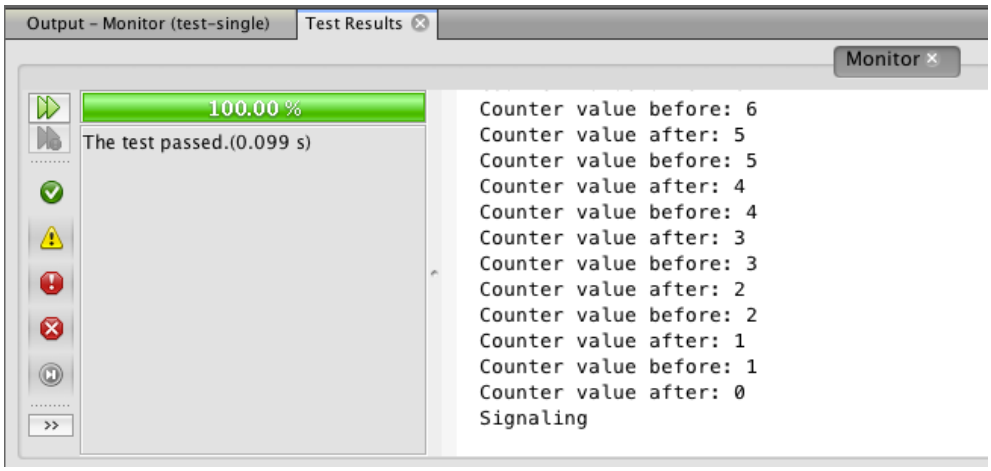

Figure 4.1: Successful execution of the tests for Count Down latch

4.2.5 Interpretation

This experiment showed us the way in which Monitors are used. We saw that we did not need to continously poll for a flag to acquire a lock. Instead, the monitor sent the threads a signal to indicate them that a particular condition has been met. In our particular test, these signals indicated: (1) that the threads can start; and (2) that the threads must stop.

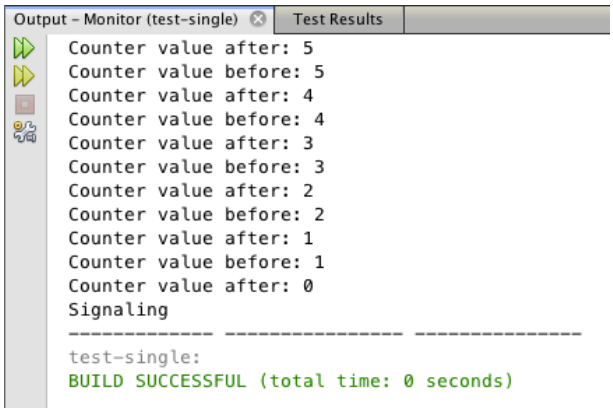

Figure 4.2: Successful execution of the tests for Count Down latch

Chapter 5

Linked Lists: The Role of Locking

5.1 LockFreeQueueTest

5.1.1 Particular Case (problem)

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

5.1.2 Solution

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

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

5.1.3 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.

5.1.4 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.

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.

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.

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.

5.1.5 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).

5.2 LazyListTest

5.2.1 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).

5.2.2 Solution

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

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

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

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

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

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

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

5.2.3 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.

5.2.4 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.

Chapter 6

Concurrent Queues and the ABA Problem

6.1 BoundedQueueTest

6.1.1 Particular Case (problem)

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

6.1.2 Solution

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

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

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

- It does not allow null values.
- There is a lock dedicated to *enq* method, which guarantees mutual exclusion to the critical section for enqueuers.
- It spins while the queue is full (the try/catch block was added just to allow compilation). This line was incorrectly asking whether queue was empty, while it needs to ask whether is still full.
- Once we know queue is room, we allocate the new node and update *tail* pointer.
- We atomically get and increment the atomic counter for the size (this was previously a decrement, which was incorrect).
- If the atomically retrieved previous value of size was zero, it means the queue was empty and there may had been waiting dequeuers; therefore we acquire the lock for *deq* method and inform all dequeuers that the queue has not at least an item.

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

6.1.3 Experiment Description

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

6.1.4 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.

6.2 SynchronousDualQueueTest

6.2.1 Particular Case (problem)

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

6.2.2 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.

6.2.3 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.

6.2.4 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.

6.3 LockFreeQueueRecycleTest

6.3.1 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.

6.3.2 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).

6.3.3 Experiment Description

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

6.3.4 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)
```

6.4 BoundedQueueTest

6.4.1 Particular Case (problem)

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

6.4.2 Solution

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

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

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

- It does not allow null values.
- There is a lock dedicated to *enq* method, which guarantees mutual exclusion to the critical section for enqueuers.
- It spins while the queue is full (the try/catch block was added just to allow compilation). This line was incorrectly asking whether queue was empty, while it needs to ask whether is still full.
- Once we know queue is room, we allocate the new node and update *tail* pointer.
- We atomically get and increment the atomic counter for the size (this was previously a decrement, which was incorrect).
- If the atomically retrieved previous value of size was zero, it means the queue was empty and there may had been waiting dequeuers; therefore we acquire the lock for *deq* method and inform all dequeuers that the queue has not at least an item.

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

6.4.3 Experiment Description

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

6.4.4 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.

Chapter 7

Concurrent Stacks and Elimination

7.1 Exchanger

7.1.1 Particular Case

The problem we are trying to solve is how to allow two threads exchange values they hold. The idea is that if a Thread A calls an *exchange*() method with a given argument and thread B calls the *exchange*() method with another arguments, then thread A will return B's argument and B will return A's argument.

Let us discuss a little bit why such method is required. The motivation of this comes from the problem of implementing a concurrent queue. Imagine that two threads are trying to access the queue and one of them wants to enqueue an element and the other one wants to dequeue an element. In this situation, we can avoid contention in the top of the stack by letting the threads simply exchange their values. That is, the enqueuer can pass its value to the dequeuer thread. This approach is called the *Elimination backoff stack*.

7.1.2 Solution

The solution in this excercise is based on the *A Scalable Elimination-based Exchange Channel* paper. Since, this algorithm is a bit more involved and it is not discussed in the book, let us describe it in more detail.

The algorithm uses an array of *AtomicReference*s. Note that the size of this array depends on the number of processors in the system. Actually, it has to be of half the number of processors. The reason for this is that if we have n threads, then we can only have up to $n/2$ concurrent exchanges and each slot in the array is used for an exchange.

```
1  public class Exchanger<V> {
2    private static final int SIZE =
3        (Runtime.getRuntime().availableProcessors() + 1) / 2;
4    private static final long BACKOFF_BASE = 128L;
5    static final Object FAIL = new Object();
6    private final AtomicReference[] arena;
7    private final Random random;
8    public Exchanger() {
9      arena = new AtomicReference[SIZE + 1];
10     random = new Random();
11     for (int i = 0; i < arena.length; ++i)
```

```
12          arena[i] = new AtomicReference();
13      }
```

Now, from our program we can call the *exchange()* methods in two ways. One in which we can specify a timeout for the exchange operation and another where there is no timeout.

```
1     public V exchange(V x) throws InterruptedException {
2       try {
3         return (V)doExchange(x, Integer.MAX_VALUE);
4       } catch (TimeoutException cannotHappen) {
5         throw new Error(cannotHappen);
6       }
7     }
8     public V exchange(V x, long timeout, TimeUnit unit)
9     throws InterruptedException, TimeoutException {
10      return (V)doExchange(x, unit.toNanos(timeout));
11    }
```

The interesting method is *doExchange()*. Imagine that a thread first tries to do an exchange. So it calls this method and gets into the whie loop. It takes the first slot in the arena and finds it is not occupied and hence it will try to occupy the slot setting its id in the slot using a *compareAndSet* operation. Then it will sit there and wait for someone else to come and exchange its value in the *waitForHole()* call.

Now imagine that another thread comes and tries to exchange its value with the previous thread. It then enters the loop, looks into the first slot and sees that someone has put a request in that slot. In other words, some other thread is willing to exchange a value. To do this, it fills the hole with the requested value and the other thread is signaled to continue. Our current thread then returns with the value of the other thread.

The first thread, as we mentioned, gets signaled and sets the slot to null to indicate that the exchange has finished and returns with the value of the other thread.

```
1     private Object doExchange(Object item, long nanos)
2         throws InterruptedException, TimeoutException {
3       Node me = new Node(item);
4       long lastTime = System.nanoTime();
```

```
5       int idx = 0;
6       int backoff = 0;
7       while (true) {
8         AtomicReference<Node> slot = (AtomicReference<Node>)arena[idx];
9         // If this slot is occupied, an item is waiting...
10        Node you = slot.get();
11        if (you != null) {
12          Object v = you.fillHole(item);
13          slot.compareAndSet(you, null);
14          if (v != FAIL) // ... unless it's cancelled
15            return v;
16        }
17        // Try to occupy this slot
18        if (slot.compareAndSet(null, me)) {
19          Object v = ((idx == 0)?
20            me.waitForHole(nanos) :
21            me.waitForHole(randomDelay(backoff)));
22          slot.compareAndSet(me, null);
23          if (v != FAIL)
24            return v;
25          if (Thread.interrupted())
26            throw new InterruptedException();
27          long now = System.nanoTime();
28          nanos -= now - lastTime;
29          lastTime = now;
30          if (nanos <= 0)
31            throw new TimeoutException();
32          me = new Node(item);
33          if (backoff < SIZE - 1)
34            ++backoff;
35          idx = 0; // Restart at top
36        } else // Retry with a random non-top slot <= backoff
37          idx = 1 + random.nextInt(backoff + 1);
38      }
39    }
40    ...
```

The algorithm adds some other improvements to the base case already described. The main improvement is the backoff logic which comes to play when two threads are already exchanging their values in a slot. In that case, the thread will try using another slot.

7.1.3 Experiment Description

The experiment here creates an array of integers of size $THREADS$. It then spawns $THREADS$ threads. Each thread will exchange its thread id with another one and will record this value in the array. For example, suppose that thread 1 and thread 2 are the participants. In this case thread 2 will store $array[2] = 1$ and thread 1 will store $array[1] = 2$. The condition that the test must satisfy is $i \neq array[array[i]]$. In this case $1 \neq array[array[1]]$.

7.1.4 Sample Results

For this test, we observed that the test always passes:

```
.exchange
OK

Time: 0.007

OK (1 test)
```

7.1.5 Interpretation

We see that the proposal of Scherer et. al works correctly. As we mentioned previously, this algorithm is very important if we want to implement a concurrent queue because this exchanger mechanism allows for what is called *Elimination*.

Chapter 8

Counting, Sorting and Distributed Coordination

8.1 TreeTest

8.1.1 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.

8.1.2 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 ($\mathrm{O}(1)$ vs $\mathrm{O}(log(p))$, where p is the number of threads or processors; but it offers a better throughput.

8.1.3 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.

8.1.4 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.