Multiprocessor Programming Course

Dario Bahena Tapia - Ricardo Zavaleta Vazquez - Isai Barajas Cicourel

December 2015

Contents

Chapter 1

Introduction

1.1 About this document

This document is the result of going through the sample programs provided with *The Art of Multiprocessor Programming* by Maurice Herlihy and Nir Shavit. These programs can be found in the book's companion web page at: `<http://people.csail.mit.edu/shanir/>`

The last course that we took for the CINVESTAV's MSc program was focused on multiprocessor programming. The textbook we used was Herlihy's. We were asked to try each of the programs provided in the web page and report our observations.

For that reason, the style of this document has the look and feel of a experiment report. The order in which we present the programs is the same as in the book. For each of the programs we first give an explanation of the problem that it is trying to solve or show. Then we describe the chosen algorithm. In most of the cases we also give a brief description of interesting points of the code.

After that, we describe any problems found while trying to run the programs. Sometimes the program itself had syntax problems, for example, and we reported how to fix it.

Each program comes with a set of tests. We ran those tests and also reported the results. Sometimes the test failed (either consistently or ocassionally). In those cases we describe what the problem is along with the fix.

We think that this exercise allowed us to understand the topics explained during the lectures. It also was of help to understand topics not covered during the course but explained in the book.

We ought to mention that while this report was written by the three of us, we got help from other students in our group. We used to gather once a week to run the programs and discuss them. We must say that it was a great experience since we all collaborated and learnt how other people think, code and solve problems.

1.2 About the machine used to run the experiments

For running the programs, we needed a machine with a lot of cores in order to be able to find any possible problem in the algorithms. At the beginning we observed that on our personal computers, most of the programs worked correctly. However, when we ran the programs on bigger machines, a lot of problems began to show up.

We got access to a Linux Server with the following characteristics:

- Number of Physical CPUs: 2
- Number of Cores: 24
- Number of Threads: 48
- Each core is a Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz
- Operating System is Oracle Enterprise Linux 6.4

So, unless otherwise specified, we will assume that the programs we run on this machine.

Chapter 2

Mutual Exclusion

2.1 LockFreeQueueTest

2.1.1 Particular Case (problem)

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

2.1.2 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.

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

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

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

2.1.3 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.

2.1.4 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.

2.1.5 Proposed changes to fix the problems

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2 Peterson Algorithm

2.2.1 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.

2.2.2 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:

```java
private boolean[] flag = new boolean[2];
private int victim;

public void lock() {
  int i = ThreadID.get();
  int j = 1-i;
  flag[i] = true;
  victim  = i;
  while (flag[j] && victim == i) {}; // spin
}

public void unlock() {
  int i = ThreadID.get();
  flag[i] = false;
}
```

2.2.3 Experiment Description

Now let's discuss how the proposed test case exercises 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.

2.2.4 Sample Results

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

```
.parallel

Time: 0.01

OK (1 test)
```

2.2.5 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 algorithm 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.

2.3 Bakery Test

2.3.1 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 guarantees 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.

2.3.2 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 ought 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:

```
public void lock() {
  int me = ThreadID.get();
  flag[me]  = true;
  int max = Label.max(label);
  label[me] = new Label(max + 1);
  while (conflict(me)) {};  // spin
}

public void unlock() {
  flag[ThreadID.get()] = false;
}

private boolean conflict(int me) {
  for (int i = 0; i < label.length; i++) {
```

```
      if (i != me && flag[i] && label[me].compareTo(label[i]) < 0) {
        return true;
      }
    }
    return false;
  }

  static class Label implements Comparable<Label> {
    int counter;
    int id;
    Label() {
      counter = 0;
      id = ThreadID.get();
    }
    Label(int c) {
      counter = c;
      id = ThreadID.get();
    }
    static int max(Label[] labels) {
      int c = 0;
      for (Label label : labels) {
        c = Math.max(c, label.counter);
      }
      return c;
    }

    public int compareTo(Bakery.Label other) {
      if (this.counter < other.counter
          || (this.counter == other.counter && this.id < other.id)) {
        return -1;
      } else if (this.counter > other.counter) {
        return 1;
      } else {
        return 0;
      }
    }
  }
```

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.

2.3.3 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.

2.3.4 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
```

2.3.5 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.

2.3.6 Proposed solution

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

```java
public void lock() {
  int me = ThreadID.get();
  flag[me]  = true;
  synchronized (label){
    int max = Label.max(label);
    label[me] = new Label(max + 1);
    while (conflict(me)) {};  // spin
  }
}
```

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.

2.4 Filter Lock

2.4.1 Particular Case

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

We are trying to do so using waiting rooms called levels that a thread must traverse before acquiring the lock.

2.4.2 Solution

We now consider two mutual exclusion protocols that work for n threads, where n is greater than 2. The Filter lock, is a direct generalization of the Peterson lock to multiple threads.

The Filter lock, creates $n-1$ "waiting rooms", called levels, that a thread must traverse before acquiring the lock. The Peterson lock uses a two-element boolean flag array to indicate whether a thread is trying to enter the critical section. The Filter lock generalizes this notion with an n-element integer $level[]$ array, where the value of $level[A]$ indicates the highest level that thread A is trying to enter.

2.4.3 Experiment Description

The test creates 8 threads that need to be coordinate in order to increment a common counter. All threads have to cooperate to increase this counter from 0 to 1024. Each of the threads will increase by one the counter 128 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.

2.4.4 Observations and Interpretations

When executing the test on a dual core machine there was a a low chance of failure, but when executed on a Quad core and Six core machines the problems where more consistent.

```
Testcase: testParallel(mutex.FilterTest):          FAILED
expected:<1023> but was:<1024>
```

2.4.5 Proposed changes to fix the problem

Even do the victim and level arrays are *volatile*, the atomicity is not guarantee and so to fix this I added the *synchronized* reserved word to the *sameOrHigher()* method in order to guarantee atomicity while checking the levels.

```
// Is there another thread at the same or higher level?
private synchronized boolean
  sameOrHigher(int me, int myLevel) {
  for (int id = 0; id < size; id++)
    if (id != me && level[id] >= myLevel) {
      return true;
    }
    return false;
}
```

Once the change is made the execution works fine on every equipment used.

```
Script Directory: .
Using OS: Linux
Linux Java 6 Runtime: ./jre/linux/bin
Will execute test from input: mutex.FilterTest
Classpath: .:./lib/*:./output/:./build/
Running JUnit using Linux
JUnit version 4.12
.
Time: 0.021

OK (1 test)
```

2.4.6 Proposed solution

Once the comparing of the level was made atomic, the spin made for conflicting threads was properly working and able to cooperate to increase the counter to 1024.

One thing that is worth mentioning is that the *volatile* keyword seems to either have an implementation problem or not documented restriction, since it is not helping guarantee the use of the waiting rooms causing additional increases in the counter, thus ending with a different number rather than the one expected.

2.5 Queue Lock

2.5.1 Particular Case

In this experiment, the particular case we are trying to study is mutual exclusion where several threads try to use a queue.

We are trying to do so using bounded partial queue.

2.5.2 Solution

According to the theory the *enq()* and *deq()* methods operate on opposite ends of the queue, so as long as the queue is neither full nor empty, an *enq()* call and a *deq()* call should, in principle, be able to proceed without interference, guarantee mutual exclusion for the methods.

Here, we implement a bounded queue as a linked list of entries as shown by the Entry Class.

```java
/**
 * Individual queue item.
 */
protected class Entry {
  /**
   * Actual value of queue item.
   */
  public T value;
  /**
   * next item in queue
   */
  public Entry next;
  /**
   * Constructor
   * @param x Value of item.
   */
  public Entry(T x) {
    value = x;
    next = null;
  }
}
```

2.5.3 Experiment Description

The test creates 8 threads that need to be coordinate in order to *enq()* and *deq()* a range of numbers. All threads have to cooperate to add and remove elements from the queue. Each of the threads will enqueue and dequeue values into the queue, if everything works according to the test there will be mutual exclusion and the mapping of elements will corresponds to the queue elements. If that is not the case, a duplicate fail will be raised.

2.5.4 Observations and Interpretations

The tests executed as expected and no errors where found. Since the ReentrantLock is used to acquire an explicit lock, the mutual exclusion is guarantee by the Java java.util.concurrent.locks package.

```
/**
 * Lock out other enqueuers (dequeuers)
 */
ReentrantLock enqLock, deqLock;
```

Chapter 3

Foundations of Shared Memory

3.1 Safe Boolean MRSW Register

3.1.1 Particular Case

In class we saw the difference between Safe, Regular and Atomic Registers. In this exercise we are experimenting with a Safe Boolean Register. In particular, we are trying a register that allows one single writer whose written 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

3.1.2 Solution

One way to achieve this behavior 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:

```
public Boolean read() {
  return s_table[ThreadID.get()];
}

public void write(Boolean x) {
  for (int i = 0; i < s_table.length; i++)
    s_table[i] = x;
}
```

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

3.1.3 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.

3.1.4 Sample Results

We found out that the tests failed occasionally. 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.

```
  /**
   * Sequential calls.
   */
  public void testSequential() {
    ThreadID.reset(); // We have to reset the thread ids
    System.out.println("sequential read and write");
    instance.write(true);
    boolean result = instance.read();
    assertEquals(result, true);
  }

  /**
   * Parallel reads
   */
  public void testParallel() throws Exception {
    ThreadID.reset(); // We have to reset the thread ids
```

```
    instance.write(false);
    instance.write(true)
    System.out.println("parallel read"
    for (int i = 0; i < THREADS; i++) {
      thread[i] = new ReadThread(
    }
    for (int i = 0; i < THREADS; i ++) {
      thread[i].start();
    }
    for (int i = 0; i < THREADS; i ++) {
      thread[i].join();
    }
  }
```

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)
```

3.1.5 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.

3.2 RegMRSWRegisterTest

3.2.1 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.

3.2.2 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):

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

```
    for (int i = 0; i < RANGE; i++)
      if (r_bit[i]) {
        return (byte)i;
      }
    return -1; // never reached
  }
}
```

3.2.3 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.

3.2.4 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)
```

3.3 Regular Boolean MRSW Register

3.3.1 Particular Case

A register, is an object that encapsulates a value that can be observed by a *read()* method and modified by a *write()* method

For Boolean registers, the only difference between safe and regular arises when the newly written value x is the same as the old. A regular register can only return x, while a safe register may return either Boolean value.

3.3.2 Solution

A register that implements the *Register¡Boolean¿* interface is called a Boolean register (we sometimes use 1 and 0 as synonyms for true and false). This register uses the *ThreadLocal* Java java.lang package, which provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one (via its get or set method) has its own, independently initialized copy of the variable. *ThreadLocal* instances are typically private static fields in classes that wish to associate state with a thread

```
public class RegBooleanMRSWRegister
implements Register<Boolean> {
  ThreadLocal<Boolean> last;
  private boolean s_value;
  RegBooleanMRSWRegister(int capacity) {
    this.last = new ThreadLocal<Boolean>() {
      protected Boolean initialValue() { return false; };
    };
  }
  public void write(Boolean x) {
    if (x != last.get()) {  // if new value different ...
      last.set(x);          // remember new value
      s_value =x;           // update register
    }
  }
  public Boolean read() {
    return s_value;
  }
}
```

3.3.3 Experiment Description

The test creates 8 threads that reads the value of a register. All threads have to be able to read the current value of the register despite the previous value is contrary to the current one. The expected result is must be a 1 or true value.

3.3.4 Observations and Interpretations

The tests executed as expected and no errors where found.

3.4 Sequential Snapshot

3.4.1 Particular Case

In order to read multiple register values atomically we use an atomic snapshot. An atomic snapshot constructs an instantaneous view of an array of atomic registers. We construct a wait-free snapshot, meaning that a thread can take an instantaneous snapshot of memory without delaying any other thread.

3.4.2 Solution

Each thread repeatedly calls *collect()*, and returns as soon as it detects a clean double collect (one in which both sets of timestamps were identical). This construction always returns correct values. The *update()* calls are wait-free, but *scan()* is not because any call can be repeatedly interrupted by *update()*, and may run forever without completing. It is however obstruction-free, since a *scan()* completes if it runs by itself for long enough.

```
public class SeqSnapshot<T> implements Snapshot<T> {
  private T[] a_value;
  public SeqSnapshot(int capacity, T init) {
    a_value = (T[])new Object[capacity];
    for (int i = 0; i < a_value.length; i++) {
      a_value[i] = init;
    }
  }
  public synchronized void update(T v) {
    a_value[ThreadID.get()] = v;
  }
  public synchronized T[] scan() {
    T[] result = (T[]) new Object[a_value.length];
    for (int i = 0; i < a_value.length; i++)
      result[i] = a_value[i];
    return result;
  }
}
```

3.4.3 Experiment Description

The test creates 2 threads that write the register twice with the the values *FIRST* and *SECOND* and later parallel reads the values.

3.4.4 Observations and Interpretations

When executing the test the test fails on the parallel part due to a index out of bound exception.

```
Testcase: testSequential(register.SeqSnapshotTest):       Caused
    an ERROR
2
java.lang.ArrayIndexOutOfBoundsException: 2
        at register.SeqSnapshot.update(SeqSnapshot.java:26)
        at register.SeqSnapshotTest.testSequential(
            SeqSnapshotTest.java:40)
```

3.4.5 Proposed changes to fix the problem

By resetting the thread id at the beginning of the test the id is guarantee to be the experted while doing the parallel test.

```
  class MyThread extends Thread {
    public void run() {
      ThreadID.reset();
      instance.update(FIRST);
      instance.update(SECOND);
      Object[] a = instance.scan();
      for (Object x : a) {
        Integer i = (Integer) x;
        int me = ThreadID.get();
        for (int j = 0; j < THREADS; j++) {
          results[me][j] = (Integer)a[j];
        }
      }
    }
  }
```

Once the change is made the execution works fine on every equipment used.

```
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
    0.097 sec

```

```
------------- Standard Output ---------------
sequential
parallel
------------- ---------------- ---------------
test-single:
BUILD SUCCESSFUL (total time: 0 seconds)
```

3.4.6 Proposed solution

Once the ThreadID is reset before changing from the sequential to the parallel test the execution never fails, since the out of bound happens due to the id being incremental.

3.5 Atomic MRSW Register

3.5.1 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.

3.5.2 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 (i.e., 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 (i.e. $A[i][x]$). 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 scanning 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:

```
public synchronized T read() {
  int me = ThreadID.get();
  StampedValue<T> value = a_table[me][me];
  for (int i = 0; i < a_table.length; i++) {
```

```
      value = StampedValue.max(value, a_table[i][me]);
    }
    for (int i = 0; i < a_table.length; i++) {
      a_table[me][i] = value;
    }
    return value.value;
  }

  public synchronized void write(T v) {
    int me = ThreadID.get();
    long stamp  = lastStamp.get() + 1;
    lastStamp.set(stamp); // remember for next time
    StampedValue<T> value = new StampedValue<T>(stamp, v);
    for (int i = 0; i < a_table.length; i++) {
      a_table[me][i] = value;
    }
  }
```

3.5.3 Experiment Description

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

3.5.4 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)
```

3.5.5 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.

3.6 AtomicMRMWRegisterTest

3.6.1 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 compared 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!).

3.6.2 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.

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

3.6.3 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.

3.6.4 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 capability, nor the interlacing of writes and reads. We believe the authors just copied paste a previous test (probably one that just cared about single reader and multiple writers), and forgot to update. Due time constraints we did not modify the test to exercise those missing features; a sample execution is shown below:

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

Time: 0.004

OK (2 tests)
```

3.7 SimpleSnapshotTest

3.7.1 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:

```java
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.

3.7.2 Solution

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

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

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

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

3.7.3 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).

3.7.4 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)
```

3.8 Wait-Free Snapshots

3.8.1 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 guarantees a Wait-Free operation.

3.8.2 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 successive 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.

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

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 *StampedSnaps*. 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.

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

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 Sequential 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.

```
public T[] scan() {
  StampedSnap<T>[] oldCopy;
  StampedSnap<T>[] newCopy;
  boolean[] moved = new boolean[a_table.length];
  oldCopy = collect();
  collect: while (true) {
    newCopy = collect();
    for (int j = 0; j < a_table.length; j++) {
      // did any thread move?
      if (oldCopy[j].stamp != newCopy[j].stamp) {
        if (moved[j]) {        // second move
          return oldCopy[j].snap;
```

```
          } else {
            moved[j] = true;
            oldCopy = newCopy;
            continue collect;
          }
        }
      }
      // clean collect
      T[] result = (T[]) new Object[a_table.length];
      for (int j = 0; j < a_table.length; j++)
        result[j] = newCopy[j].value;
      return result;
    }
  }
```

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

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

3.8.3 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.

3.8.4 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)
```

3.8.5 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 4

Spin Locks and Contention

4.1 Problem Definition

The concept of Spin Lock comes from the following problem: *In the case where a thread tries to acquire 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.

4.2 Test-And-Set Locks

4.2.1 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*?

4.2.2 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:

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

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.

4.2.3 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 $8 * 1024$

4.2.4 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)
```

4.2.5 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.

4.3 TTASLockTest

4.3.1 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.

4.3.2 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:

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

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

4.3.3 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.

4.3.4 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).

4.4 BackoffLockTest

4.4.1 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 mutual exclusion problem. The particular subproblem is doing that with spinning locks (those which repeatedly try to acquire the lock); and the very concrete sub-subproblem here is how to reduce the contention out of the *TTASLock*.

4.4.2 Solution

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

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

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

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]. Every time we do this waiting, the range is enlarged twice (meaning that on average the random number is twice large); 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):

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

public class Backoff {
  ...

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

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

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

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

4.4.3 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).

4.4.4 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)
```

4.5 CLH Queue Lock

4.5.1 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?*

4.5.2 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.

4.5.3 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 $8 * 1024$

4.5.4 Sample Results

In this case, we found that in our 24-cores machine, the test hung. To make it work, we had to make the *locked* variable in the

```
[oraadm@gdlaa008 Spin]$ junit spin.CLHLockTest
[8] Leaving critical zone
[9] Entering critical zone
[9] Leaving critical zone
[8] Entering critical zone
[8] Leaving critical zone
[9] Entering critical zone
[9] Leaving critical zone
[8] Entering critical zone
[9] Entering critical zone
...
[9] Leaving critical zone
[8] Entering critical zone
[9] Entering critical zone
[8] Leaving critical zone
[8] Entering critical zone
[8] Leaving critical zone

Time: 0.755

OK (1 test)
```

4.5.5 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 algorithm 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 degrades.

4.6 MCSLockTest

4.6.1 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.

4.6.2 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:

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

[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.

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

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

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

4.6.3 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.

4.6.4 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):

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

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

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

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

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

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

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

  ...

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

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

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

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

We immediately notice same problem as with *locked* flag; the *next* pointer also needs immediate propagation to main memory, otherwise the threads will not detect then is being set to null. Let us also note that the hanging due *next* variable, triggers the hanging on the spinning loop for *lock* method; the *locked* flag is volatile, so a write to it would propagate right away and invalidate other's caches; but since the thread doing the *unlock* never exits the loop, it can not perform such write. Putting as volatile both attributes 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*).

4.7 Array Lock

4.7.1 Particular Case

The ALock, is a simple array-based queue lock. In a queue, each thread can learn if its turn has arrived by checking whether its predecessor has finished.

4.7.2 Solution

The threads share an *AtomicInteger* tail field, initially zero. To acquire the lock, each thread atomically increments tail. Call the resulting value the threads slot. The slot is used as an index into a Boolean flag array. If *flag[j]* is true, then the thread with slot j has permission to acquire the lock.

```java
public class ALock implements Lock {
  // thread-local variable
  ThreadLocal<Integer> mySlotIndex = new ThreadLocal<Integer> () {
    protected Integer initialValue() {
      return 0;
    }
  };
  volatile AtomicInteger tail;
  volatile boolean[] flag;
  int size;
  /**
   * Constructor
   * @param capacity max number of array slots
   */
  public ALock(int capacity) {
    size = capacity;
    tail = new AtomicInteger(0);
    flag = new boolean[capacity];
    flag[0] = true;
  }
  public void lock() {
    int slot = tail.getAndIncrement() % size;
    mySlotIndex.set(slot);
    while (! flag[mySlotIndex.get()]) {}; // spin
  }
  public void unlock() {
    flag[mySlotIndex.get()] = false;
    flag[(mySlotIndex.get() + 1) % size] = true;
  }
```

```java
  // any class implementing Lock must provide these methods
  public Condition newCondition() {
    throw new java.lang.UnsupportedOperationException();
  }
  public boolean tryLock(long time,
      TimeUnit unit)
      throws InterruptedException {
    throw new java.lang.UnsupportedOperationException();
  }
  public boolean tryLock() {
    throw new java.lang.UnsupportedOperationException();
  }
  public void lockInterruptibly() throws InterruptedException {
    throw new java.lang.UnsupportedOperationException();
  }
}
```

4.7.3 Experiment Description

The test creates 8 threads that need to be coordinate in order to increment a common counter. All threads have to cooperate to increase this counter from 0 to 1024. Each of the threads will increase by one the counter 128 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.

4.7.4 Observations and Interpretations

When executing the test values where lost from the queue.

```
1009
1010
1011
1013
1014
1015
```

4.7.5 Proposed changes to fix the problem

By adding the volatile keyword in the ALock class the execution was fixed.

```java
volatile AtomicInteger tail;
volatile boolean[] flag;
int size;
```

Once the change is made the execution works fine on every equipment used.

4.7.6 Proposed solution

Apparently the AtomicInteger keyword is not guarantee that the values are propagated through the memory hierarchy and by making the value atomic the execution works.

4.8 Composite Fast Path Lock

4.8.1 Particular Case

In this experiment, we extend the *CompositeLock* algorithm to encompass a fast path in which a solitary thread acquires an idle lock without acquiring a node and splicing it into the queue.

4.8.2 Solution

We use a *FASTPATH* flag to indicate that a thread has acquired the lock through the fast path. Because we need to manipulate this flag together with the tail fields reference, we take a high-order bit from the tail fields integer stamp. The private *fastPathLock()* method checks whether the tail fields stamp has a clear *FASTPATH* flag and a null reference. If so, it tries to acquire the lock simply by applying *compareAndSet()* to set the FASTPATH flag to true, ensuring that the reference remains null. An uncontended lock acquisition thus requires a single atomic operation. The *fastPathLock()* method returns true if it succeeds, and false otherwise.

4.8.3 Experiment Description

The test creates 2 threads that need to be coordinate in order to increase a counter. All threads have to cooperate in order to increase the counter into the desire value, this lock implements a explicit lock that the test calls. If that is not the case, an assertion fail will be raised.

4.8.4 Observations and Interpretations

The tests executed as expected and no errors where found. Since the *CompositeFastPathLock* is used to acquire an explicit lock, the mutual exclusion is guarantee.

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.895 sec

test-single:
BUILD SUCCESSFUL (total time: 3 seconds)
```

4.9 Time Out Lock

4.9.1 Particular Case

In this experiment we consider the Java Lock interface which includes a *tryLock()* method that allows the caller to specify a *timeout* or maximum duration the caller is willing to wait to acquire the lock. The TOLock implements this mechanism in order to wait for a resource.

4.9.2 Solution

The *tryLock()* method creates a new QNode with a null pred field and appends it to the list as in the CLHLock class. If the lock was free, the thread enters the critical section. Otherwise, it spins waiting for its predecessors QNodes pred field to change. If the predecessor thread times out, it sets the pred field to its own predecessor, and the thread spins instead on the new predecessor.

```
    while (System.nanoTime() - startTime < patience) {
      QNode predPred = pred.pred;
      if (predPred == AVAILABLE) {
        return true;
      } else if (predPred != null) {  // skip predecessors
        pred = predPred;
      }
    }
    // timed out; reclaim or abandon own node
    if (!tail.compareAndSet(qnode, pred))
      qnode.pred = pred;
```

4.9.3 Experiment Description

The test creates 8 threads that need to be coordinate in order to increase a counter. All threads have to cooperate in order to increase the counter into the desire value, this lock implements a explicit lock that the test calls, but with the addition of a time out. If that is not the case, an assertion fail will be raised.

4.9.4 Proposed changes to improve execution

By making volatile the QNode's in the TOLock class the execution finishes faster.

```
    volatile static QNode AVAILABLE = new QNode();
    volatile AtomicReference<QNode> tail;
    ThreadLocal<QNode> myNode;
```

4.9.5 Observations and Interpretations

The tests executed as expected and no errors where found. But the time out generates some waiting depending on the number of cores the machine has.

```
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 22.551 sec

------------- Standard Output ---------------
locking
Created Threads
Started Threads
Finished Threads
timeout
timeouts: 8191
------------- ---------------- ---------------
test-single:
BUILD SUCCESSFUL (total time: 24 seconds)
```

4.10 CompositeLockTest

4.10.1 Particular Case (problem)

The particular problem we are trying to solve, is that of looking for a good balance between the trade-offs 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.

4.10.2 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.

4.10.3 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.

4.10.4 Observations and Interpretations

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

```
.
Time: 0.066

OK (1 test)
```

4.11 Hierarchical Backoff Lock

4.11.1 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 communication and inter-cluster communication. 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?*

4.11.2 Solution

The first algorithm that we are going to study is the Hierarchical Back-off 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 overall time needed to switch the lock ownership. Let us take a quick look at the source code in java:

```java
public void lock() {
  int myCluster = ThreadID.getCluster();
  Backoff localBackoff =
      new Backoff(LOCAL_MIN_DELAY, LOCAL_MAX_DELAY);
  Backoff remoteBackoff =
      new Backoff(REMOTE_MIN_DELAY, REMOTE_MAX_DELAY);
  while (true) {
    if (state.compareAndSet(FREE, myCluster)) {
      return;
    }
    int lockState = state.get();
    try {
      if (lockState == myCluster) {
        localBackoff.backoff();
      } else {
        remoteBackoff.backoff();
      }
    } catch (InterruptedException ex) {
```

```
      }
    }
  }

  public void unlock() {
    state.set(FREE);
  }
```

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

```
  private static final int LOCAL_MIN_DELAY = 8;
  private static final int LOCAL_MAX_DELAY = 256;
  private static final int REMOTE_MIN_DELAY = 256;
  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 back-off. If the lock is being hold by another thread in the cluster, then we do a short back-off. Otherwise, we do a long back-off.

The unlock method simply sets the state to free.

4.11.3 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 $8 * 1024$

4.11.4 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)
```

4.11.5 Interpretation

In this experiment we discovered how to implement a locking algorithm in a cache-coherent architecture by performing back-offs 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 5

Monitors and Blocking Synchronization

5.1 Queue Lock

5.1.1 Particular Case

In this experiment, the particular case we are trying to study is mutual exclusion where several threads try to use a queue.

We are trying to do so using bounded partial queue.

5.1.2 Solution

According to the theory the *enq()* and *deq()* methods operate on opposite ends of the queue, so as long as the queue is neither full nor empty, an *enq()* call and a *deq()* call should, in principle, be able to proceed without interference, guarantee mutual exclusion for the methods.

Here, we implement a bounded queue as a linked list of entries as shown by the Entry Class.

```
/**
 * Individual queue item.
 */
protected class Entry {
  /**
   * Actual value of queue item.
   */
  public T value;
  /**
   * next item in queue
   */
  public Entry next;
  /**
   * Constructor
   * @param x Value of item.
   */
  public Entry(T x) {
    value = x;
    next = null;
  }
}
```

5.1.3 Experiment Description

The test creates 8 threads that need to be coordinate in order to *enq()* and *deq()* a range of numbers. All threads have to cooperate to add and remove elements from the queue. Each of the threads will enqueue and dequeue values into the queue, if everything works according to the test there will be mutual exclusion and the mapping of elements will corresponds to the queue elements. If that is not the case, a duplicate fail will be raised.

5.1.4 Observations and Interpretations

The tests executed as expected and no errors where found. Since the ReentrantLock is used to acquire an explicit lock, the mutual exclusion is guarantee by the Java java.util.concurrent.locks package.

```java
/**
 * Lock out other enqueuers (dequeuers)
 */
ReentrantLock enqLock, deqLock;
```

5.2 Non-Reentrant Mutual Exclusion

5.2.1 Particular Case

In this experiment we use a re-entrant mutual exclusion Lock with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities.

5.2.2 Solution

The Mutex lock extends a *AbstractQueuedSynchronizer*, we have a *tryAcquire()* that only acquire the lock when the state is FREE, an *tryRelease()* that releases the lock by setting the state to FREE and a method to verify the current state of the lock. Due to the AbstractQueued we use a framework for implementing blocking locks and related synchronizers (semaphores, events, etc) that rely on first-in-first-out (FIFO) wait queues. This extended class is designed to be a useful basis for most kinds of synchronizers that rely on a single atomic int value to represent state, thus the object does all the work and we forward the conditions.

```
private final LockSynch sync = new LockSynch();

public void lock() {
  sync.acquire(0);
}
public boolean tryLock() {
  return sync.tryAcquire(0);
}
public void unlock() {
  sync.release(0);
}
public Condition newCondition() {
  return sync.newCondition();
}
public boolean isLocked() {
  return sync.isHeldExclusively();
}
public boolean hasQueuedThreads() {
  return sync.hasQueuedThreads();
}
public void lockInterruptibly() throws InterruptedException {
  sync.acquireInterruptibly(0);
}
```

```java
  public boolean tryLock(long timeout, TimeUnit unit) throws
      InterruptedException {
    return sync.tryAcquireNanos(1, unit.toNanos(timeout));
  }
```

5.2.3 Experiment Description

The test creates 8 threads that need to be coordinate in order to increase a counter. All threads have to cooperate in order to increase the counter into the desire value, this lock implements a explicit lock that the test calls, but with the addition of a re-entrant lock sync, which in addition to implementing the Lock interface, this class defines methods *isLocked()* and *hasQueueThreads()*, as well as some associated protected access methods that may be used for instrumentation and monitoring. If that is not the case, an assertion fail will be raised.

5.2.4 Observations and Interpretations

The tests executed as expected and no errors where found. Since we extend the AbstractQueuedSynchronizer, we are mainly using the Java java.util.concurrent.locks package.

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
    0.382 sec

————————— Standard Output ——————————
parallel
————————— ——————————— ——————————
test-single:
BUILD SUCCESSFUL (total time: 1 second)
```

5.3 Count Down Latch

5.3.1 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 data structures (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.

5.3.2 Solution

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

Let us explain how the implementation of the methods *countDown()* and *await()* work:

```
  public void await() throws InterruptedException {
    lock.lock();
    try {
      while (counter > 0)
        condition.await();
    } finally {
      lock.unlock();
    }
  }

  public void countDown() {
    lock.lock();
    try {
      counter--;
      if (counter == 0) {
```

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

Basically, when we create an object of type *CountDownLatch*, we initialize a counter with a value. The goal of the object is to count down to zero. To achieve this, we need to execute the method *countDown()* a number of times.

For example, one thread can create a *CountDownLatch* object with a counter of 10 and then it sits to wait, i.e. calls the *await()* function. Other threads then call the method *countDown* to decrement the counter. When the counter reaches zero, then the first thread gets signaled.

5.3.3 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 proceed 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 controls 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.

Let us quickly review the test case:

At the beginning we see the two *CountDownLatch* objects. One is used to start and one is used to finish. The one for start is initialized with 1 because only one thread will trigger it. The one for start is initialized with 8 because we will create 8 threads and each of them will call *countDown()* once.

```
...
static final int THREADS = 8; // number threads to test
```

```java
static final int START = 1492;
static final int STOP  = 1776;
CountDownLatch startSignal = new CountDownLatch(1);
CountDownLatch doneSignal = new CountDownLatch(THREADS);
...
public void testCountDownLatch() {
  System.out.format("Testing %d threads\n", THREADS);
  CountDownLatch startSignal = new CountDownLatch(1);
  CountDownLatch doneSignal  = new CountDownLatch(THREADS);
  Thread[] thread = new Thread[THREADS];
  int[] log = new int[THREADS];
  // create threads
  for (int j = 0; j < THREADS; j++)
    thread[j] = new TestThread(j, THREADS, startSignal, doneSignal, log);
  // start threads
  for (int j = 0; j < THREADS; j++)
    thread[j].start();
  // do something
  for (int i = 0; i < THREADS; i++)
    log[i] = START;
  // give threads permission to start
  startSignal.countDown();
  // wait for threads to complete
  try {
    doneSignal.await();
  } catch (InterruptedException e) {
    fail("interrupted exception");
  }
  for (int i = 0; i < THREADS; i++)
    if (log[i] != STOP)
      fail("found bad value: " + log[i]);
}
```

So, at start the main thread creates 8 threads and make them run. When these threads start, the are put to wait immediately. Then the main thread signals them to actually start and it puts himself to wait.

```java
public class TestThread extends Thread {
  int threads;
  CountDownLatch startSignal;
  CountDownLatch doneSignal;
  int[] log;
  int index;
  public TestThread(int index, int threads,
```

```
      CountDownLatch startSignal, CountDownLatch doneSignal, int[] log) {
    this.threads = threads;
    this.log = log;
    this.index = index;
    this.startSignal = startSignal;
    this.doneSignal = doneSignal;
  }
  public void run() {
    ThreadID.set(index);
    try {
      startSignal.await(); // wait for permission to start
    } catch (InterruptedException ex) {
      ex.printStackTrace();
    }
    if (log[index] != START) {
      System.out.format("%d\tError expected %d found %d at %d\n", START,
log[index], index);
    }
    log[index] = STOP;
    doneSignal.countDown();  // notify driver we are done
  }
}
```

Then each of the 8 threads will run and do their stuff. At the end, each of the threads calls *countDown()* for the doneSignal. When the counter for this latch reaches 0, the program ends.

5.3.4 Sample Results

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

```
[oraadm@gdlaa008 Monitor]$ junit monitor.CountDownLatchTest
.Testing 8 threads

Time: 0.01

OK (1 test)
```

5.3.5 Interpretation

This experiment showed us the way in which Monitors are used. We saw that we did not need to continuously 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.

Chapter 6

Linked Lists: The Role of Locking

6.1 Coarse List

6.1.1 Particular Case

In this exercise we are trying to solve the problem of accessing a List data structure by multiple threads simultaneously. While allowing multiple threads accessing the data structure, we still want to maintain its consistency.

As mentioned in the book, the invariants that we want to satisfy are:

1. Sentinels are neither added or removed (i.e. the head and the tail)
2. Nodes are sorted by key
3. Keys are unique

6.1.2 Solution

Our first approach to solving this problem is then through a coarse grained list. This means that we use a lock in such a way that any access to the list is sequential because this lock must be hold and there is only one lock.

Let's take a quick look at the add and remove items to grasp the idea behind this algorithm. Notice how the whole logic of the *add()* and *remove()* operations are guarded by the calls to *lock()* and *unlock()*.

```
/**
 * Add an element.
 * @param item element to add
 * @return true iff element was not there already
 */

public boolean add(T item) {
  Node pred, curr;
  int key = item.hashCode();
  lock.lock();
  try {
    pred = head;
    curr = pred.next;
    while (curr.key < key) {
      pred = curr;
      curr = curr.next;
    }
    if (key == curr.key) {
```

```
        return false;
      } else {
        Node node = new Node(item);
        node.next = curr;
        pred.next = node;
        return true;
      }
    } finally {
      lock.unlock();
    }
  }

  /**
   * Remove an element.
   * @param item element to remove
   * @return true iff element was present
   */
  public boolean remove(T item) {
    Node pred, curr;
    int key = item.hashCode();
    lock.lock();
    try {
      pred = this.head;
      curr = pred.next;
      while (curr.key < key) {
        pred = curr;
        curr = curr.next;
      }
      if (key == curr.key) {  // present
        pred.next = curr.next;
        return true;
      } else {
        return false;         // not present
      }
    } finally {               // always unlock
      lock.unlock();
    }
  }
```

6.1.3 Experiment Description

We have four test cases:

1. *testSequential()*. The main thread inserts multiple elements to the list, then checks if all elements are there and finally removes all elements one by one.

2. *testParallelAdd()*. The main thread spawns multiple threads that add elements to the list concurrently. When they are done, the main thread checks if all elements are in there and finally removes all elements.

3. *testParallelRemove()*. The main thread inserts multiple elements to the list, then checks if all elements are in there and finally it spawns multiple threads that remove the elements from the list concurrently.

4. *testParallelBoth()*. The main thread creates multiple threads. Some of them insert elements to the list and some of them remove elements.

6.1.4 Sample Results

For this test, we saw that it failed consistently with the following error:

```
[oraadm@gdlaa008 Lists]$ junit lists.CoarseListTest
.sequential add, contains, and remove
.parallel both
Exception in thread "Thread-3" Exception in thread "Thread-5" Exception in
thread "Thread-9" Exception in thread "Thread-15"
junit.framework.AssertionFailedError: RemoveThread: duplicate remove: 32
        at junit.framework.Assert.fail(Assert.java:57)
        at junit.framework.TestCase.fail(TestCase.java:227)
        at lists.CoarseListTest$RemoveThread.run(CoarseListTest.java:145)
junit.framework.AssertionFailedError: RemoveThread: duplicate remove: 121
        at junit.framework.Assert.fail(Assert.java:57)
        at junit.framework.TestCase.fail(TestCase.java:227)
        at lists.CoarseListTest$RemoveThread.run(CoarseListTest.java:145)
junit.framework.AssertionFailedError: RemoveThread: duplicate remove: 17
        at junit.framework.Assert.fail(Assert.java:57)
        at junit.framework.TestCase.fail(TestCase.java:227)
        at lists.CoarseListTest$RemoveThread.run(CoarseListTest.java:145)
junit.framework.AssertionFailedError: RemoveThread: duplicate remove: 66
        at junit.framework.Assert.fail(Assert.java:57)
        at junit.framework.TestCase.fail(TestCase.java:227)
        at lists.CoarseListTest$RemoveThread.run(CoarseListTest.java:145)
.parallel add
.parallel remove

Time: 0.017

OK (4 tests)
```

So, the test that fails consistently is *parallel both*. It complains saying that an specific element was removed multiple times by one or more threads.

6.1.5 Proposed solution

This failure turns out to be a test issue. Each remover thread is in charge of removing a specific set of elements, say from 25 to 50. In the same way, an inserter thread is in charge of adding a specific set of elements. Since the threads for a given interval are spawned concurrently, it might be the case that a remover thread is trying to remove an element that hasn't been inserted yet.

One way of solving this problem is by asking the remover thread to retry till it is successful. (Obviously this fix should be improved further since it might lead to a hang in user's application – Imagine the case where an specific element is never inserted. However, for this specific case, the fix is good enough.)

This is the code of the remover thread before the fix:

```
class removethread extends thread {
  int value;
  removethread(int i) {
    value = i;
  }
  public void run() {
    for (int i = 0; i < per_thread; i++) {
      if (!instance.remove(value + i)) {
        fail("removethread: duplicate remove: " + (value + i));
      }
    }
  }
}
```

And here is the code with the fix:

```
class removethread extends thread {
  int value;
  removethread(int i) {
    value = i;
  }
  public void run() {
    for (int i = 0; i < per_thread; i++) {
      while (!instance.remove(value + i)) {}
    }
  }
```

```
    }
```

And here is the test output after the fix:

```
[oraadm@gdlaa008 Lists]$ junit lists.CoarseListTest
.sequential add, contains, and remove
.parallel both
.parallel add
.parallel remove

Time: 0.016

OK (4 tests)
```

6.1.6 Interpretation

In this experiment we observed how to implement a simple list that allows concurrent access by making all the access sequential. We saw that even though the algorithm is correct, a correct usage of the API is also required.

6.2 Fine-Grained Synchronization List

6.2.1 Particular Case

The Fine-Grained synchronization improves concurrency by locking individual nodes, rather than locking the list as a whole. Instead of placing a lock on the entire list, we add a Lock to each node, along with *lock()* and *unlock()* methods.

6.2.2 Solution

Just as in the coarse-grained list, *remove()* makes currA unreachable by setting predA next field to currA successor. To be safe, *remove()* must lock both predA and currA. To guarantee progress, it is important that all methods acquire locks in the same order, starting at the head and following next references toward the tail. The *FineList* class *add()* method uses hand-over-hand locking to traverse the list. The finally blocks release locks before returning.

```
public boolean add(T item) {
  int key = item.hashCode();
  head.lock();
  Node pred = head;
  try {
    Node curr = pred.next;
    curr.lock();
    try {
      while (curr.key < key) {
        pred.unlock();
        pred = curr;
        curr = curr.next;
        curr.lock();
      }
      if (curr.key == key) {
        return false;
      }
      Node newNode = new Node(item);
      newNode.next = curr;
      pred.next = newNode;
      return true;
    } finally {
      curr.unlock();
    }
```

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

```
  public boolean remove(T item) {
    Node pred = null, curr = null;
    int key = item.hashCode();
    head.lock();
    try {
      pred = head;
      curr = pred.next;
      curr.lock();
      try {
        while (curr.key < key) {
          pred.unlock();
          pred = curr;
          curr = curr.next;
          curr.lock();
        }
        if (curr.key == key) {
          pred.next = curr.next;
          return true;
        }
        return false;
      } finally {
        curr.unlock();
      }
    } finally {
      pred.unlock();
    }
  }
```

6.2.3 Experiment Description

The test creates 8 threads that need to be coordinate in order to add and remove values from a list. All threads have to cooperate in order to parallel add and remove values from the list while avoiding duplicate remove of the values. If that is not the case, a fail will be raised.

6.2.4 Observations and Interpretations

The tests executed as expected and no errors where found.

```
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
    0,183 sec

------------- Standard Output ---------------
sequential add, contains, and remove
parallel add
parallel remove
parallel both
------------- ---------------- ---------------
test-single:
BUILD SUCCESSFUL (total time: 2 seconds)
```

6.3 Optimistic Synchronization

6.3.1 Particular Case

In this experiment, the particular case we are trying to study an Optimistic Synchronization which to reduce synchronization costs we take a chance by, search without acquiring locks, lock the nodes found, and then confirm that the locked nodes are correct.

6.3.2 Solution

The *OptimisticList add()* method traverses the list ignoring locks, acquires locks, and validates before adding the new node, we must validate and guarantee freedom from interference by checking that predA points to currA and is reachable from head.

```
public boolean add(T item) {
  int key = item.hashCode();
  while (true) {
    Entry pred = this.head;
    Entry curr = pred.next;
    while (curr.key <= key) {
      pred = curr; curr = curr.next;
    }
    pred.lock(); curr.lock();
    try {
      if (validate(pred, curr)) {
        if (curr.key == key) { // present
          return false;
        } else {               // not present
          Entry entry = new Entry(item);
          entry.next = curr;
          pred.next = entry;
          return true;
        }
      }
    } finally {                // always unlock
      pred.unlock(); curr.unlock();
    }
  }
}
```

The *remove()* method traverses ignoring locks, acquires locks, and validates before removing the node.

```
public boolean remove(T item) {
  int key = item.hashCode();
  while (true) {
    Entry pred = this.head;
    Entry curr = pred.next;
    while (curr.key < key) {
      pred = curr; curr = curr.next;
    }
    pred.lock(); curr.lock();
    try {
        if (validate(pred, curr)) {
          if (curr.key == key) { // present in list
            pred.next = curr.next;
            return true;
          } else {               // not present in list
            return false;
          }
        }
    } finally {                // always unlock
      pred.unlock(); curr.unlock();
    }
  }
}
```

6.3.3 Experiment Description

The test creates 8 threads that need to be coordinate in order to add and remove values from a list. All threads have to cooperate in order to parallel add and remove values from the list while avoiding duplicate remove of the values. If that is not the case, a fail will be raised.

6.3.4 Observations and Interpretations

When executing the test duplicate removes where found.

```
junit.framework.AssertionFailedError: RemoveThread: duplicate
    remove:
```

6.3.5 Proposed changes to fix the problem

The test approach is naive, since it assumes that data would exist always prior deletion, thus not handling the case when there is no data to remove. By doing a retry until there is data to remove the problem seems to be fix.

```
  public boolean remove(T item) {
    int key = item.hashCode();
    boolean bRetry = true;
    while (true) {
      Entry pred = this.head;
      Entry curr = pred.next;
      while (curr.key < key) {
        pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
        while (bRetry) {
          if (validate(pred, curr)) {
            bRetry = false;
            if (curr.key == key) { // present in list
              pred.next = curr.next;
              return true;
            } else {              // not present in list
              return false;
            }
          }
        }
      } finally {              // always unlock
        pred.unlock(); curr.unlock();
      }
    }
  }
}
```

Once the change is made the execution works fine on every equipment used.

```
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
    0,071 sec

------------- Standard Output ---------------
sequential add, contains, and remove
parallel add
parallel remove
parallel both
------------- ---------------- ---------------
```

```
test-single:
BUILD SUCCESSFUL (total time: 3 seconds)
```

6.3.6 Proposed solution

This error may imply a bad scheduling of the list. Apparently old JVM depending on the OS, executed using a logic that avoided this from happening by working in a round robin fashion.

6.4 LazyListTest

6.4.1 Particular Case (problem)

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

6.4.2 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 behavior 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 (hence 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:

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

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

6.4.3 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 exception that it uses 8 threads instead of two for each operation (*add / remove*).
- The threads that remove elements do not care only in successfully removing certain number of times (like with the queue); here they expect to remove a particular subset of the values.

6.4.4 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.

6.5 Lock-Free List

6.5.1 Particular Case

We have seen some algorithms for implementing a concurrent list. Now it is time to see how a Lock-Free implementation can be done. That is our current problem.

6.5.2 Solution

There are two main ideas behind the Lock-Free implementation. The first one is to use a lazy approach. That is, do not remove an element to the list immediately. Instead, mark it as *deleted*. The second idea is to treat the node's *next* and a new field *marked* as a single atomic unit.

The second idea is important because that will allow us to have concurrent *add()* and *remove()* with out missing updates.

Three methods are important in the implementation of the Lock-Free List we are discussing. The first one is the *find()* method which encapsulates the mechanism to search for a *current* and *previous* nodes that will be affected by an *add()* or *remove()* operation. These two fields are grouped in a *Window* object.

This method traverses the list. If it finds a node that is marked as deleted, then it tries to actually remove it from the list. It does so by applying a *compareAndSet()* to both the *pred.next* and its *mark*. This latter field is implicit in the *AtomicMarkableReference¡T¿* object.

```
  public Window find(Node head, int key) {
    Node pred = null, curr = null, succ = null;
    boolean[] marked = {false}; // is curr marked?
    boolean snip;
    retry: while (true) {
      pred = head;
      curr = pred.next.getReference();
      while (true) {
        succ = curr.next.get(marked);
        while (marked[0]) {           // replace curr if marked
          snip = pred.next.compareAndSet(curr, succ, false, false);
          if (!snip) continue retry;
          curr = pred.next.getReference();
          succ = curr.next.get(marked);
```

```
      }
      if (curr.key >= key)
        return new Window(pred, curr);
      pred = curr;
      curr = succ;
    }
  }
}
```

The next interesting method is *add()* which after finding its window, if it sees that the value we are trying to insert is already there, returns false. Otherwise it adds the new node using the *compareAndSet()* operation.

```
/**
 * Add an element.
 * @param item element to add
 * @return true iff element was not there already
 */
public boolean add(T item) {
  int key = item.hashCode();
  boolean splice;
  while (true) {
    // find predecessor and curren entries
    Window window = find(head, key);
    Node pred = window.pred, curr = window.curr;
    // is the key present?
    if (curr.key == key) {
      return false;
    } else {
      // splice in new node
      Node node = new Node(item);
      node.next = new AtomicMarkableReference(curr, false);
      if (pred.next.compareAndSet(curr, node, false, false)) {
        return true;
      }
    }
  }
}
```

Finally, we have the *remove()* method. After finding its window, if it does not see the value it is trying to delete, it returns false. Otherwise, it marks the node as *logically deleted*. After that, it tries to *physically delete* the node once. For the method, it is OK if it fails since any other thread trying to add or remove will attempt to do it again.

```
/**
 * Remove an element.
 * @param item element to remove
 * @return true iff element was present
 */
public boolean remove(T item) {
  int key = item.hashCode();
  boolean snip;
  while (true) {
    // find predecessor and curren entries
    Window window = find(head, key);
    Node pred = window.pred, curr = window.curr;
    // is the key present?
    if (curr.key != key) {
      return false;
    } else {
      // snip out matching node
      Node succ = curr.next.getReference();
      snip = curr.next.attemptMark(succ, true);
      if (!snip)
        continue;
      pred.next.compareAndSet(curr, succ, false, false);
      return true;
    }
  }
}
```

6.5.3 Experiment Description

Again, we have the same four test cases that we have been dealing with:

1. *testSequential()*. The main thread inserts multiple elements to the list, then checks if all elements are there and finally removes all elements one by one.

2. *testParallelAdd()*. The main thread spawns multiple threads that add elements to the list concurrently. When they are done, the main thread checks if all elements are in there and finally removes all elements.

3. *testParallelRemove()*. The main thread inserts multiple elements to the list, then checks if all elements are in there and finally it spawns multiple threads that remove the elements from the list concurrently.

4. *testParallelBoth()*. The main thread creates multiple threads. Some of them insert elements to the list and some of them remove elements.

6.5.4 Sample Results

Similar to the other implementations, the test cases turns out to have a problem. Here is the output:

```
[oraadm@gdlaa008 Lists]$ junit lists.LockFreeListTest
.sequential add, contains, and remove
.parallel both
Exception in thread "Thread-13" junit.framework.AssertionFailedError:
RemoveThread: duplicate remove: 109
        at junit.framework.Assert.fail(Assert.java:57)
        at junit.framework.TestCase.fail(TestCase.java:227)
        at lists.LockFreeListTest$RemoveThread.run(LockFreeListTest.java:145)
.parallel add
.parallel remove

Time: 0.022

OK (4 tests)
```

The fix for this is the same that has been discussed before. Here is the test output after the fix:

```
[oraadm@gdlaa008 Lists]$ junit lists.LockFreeListTest
.sequential add, contains, and remove
.parallel both
.parallel add
.parallel remove

Time: 0.016

OK (4 tests)
```

6.5.5 Interpretation

We observed a way in which a list could be implemented allowing it to be lock free. When using the same set of unit tests as before, we see that the implementation works correctly.

Chapter 7

Concurrent Queues and the ABA Problem

7.1 LockFreeQueueTest Test

7.1.1 Particular Case

In this exercise we deal with the problem of implementing a Queue data structure with the particular characteristic of being lock free.

7.1.2 Solution

Here we present the implementation of the Lock Free Queue that is based on the paper by Michael and Scott.

Here is the *enqueue()* method:

```
/**
 * Append item to end of queue.
 * @param item
 */
public void enq(T item) {
  if (item == null) throw new NullPointerException();
  Node node = new Node(item); // allocate & initialize new node
  while (true) {                // keep trying
    Node last = tail.get();     // read tail
    Node next = last.next.get(); // read next
    if (last == tail.get()) { // are they consistent?
      AtomicReference[] target = {last.next, tail};
      Object[] expect = {next, last};
      Object[] update = {node, node};
      if (multiCompareAndSet(
          (AtomicReference<T>[]) target,
          (T[]) expect,
          (T[]) update)){
        return;
      }
    }
  }
}
```

Here is the dequeue method.

```
/**
 * Remove and return head of queue.
```

```
   * @return remove first item in queue
   * @throws queue.EmptyException
   */
  public T deq() throws EmptyException {
    while (true) {
      Node first = head.get();
      Node last = tail.get();
      Node next = first.next.get();
      if (first == head.get()) {// are they consistent?
        if (first == last) {    // is queue empty or tail falling behind?
          if (next == null) {   // is queue empty?
            throw new EmptyException();
          }
          // tail is behind, try to advance
          tail.compareAndSet(last, next);
        } else {
          T value = next.value; // read value before dequeuing
          if (head.compareAndSet(first, next))
            return value;
        }
      }
    }
```

7.1.3 Experiment Description

The test consists of 4 test cases. Let us briefly describe each one of them:

- testSequential. In this test case 512 elements are enqueued in the data structure. After that, each of the elements are dequeued. At each dequeue, we verify that the dequeued elements are in sequential order (just as they were inserted).

- testParallelEnqueue. This test cases spawns 8 enqueuer threads. Each of them will add 512 elements to the queue. When they are done, the main thread dequeues each of the elements performing the same check described above.

- testParallelDequeue. In this test case, the master thread enqueues 512 elements. When done, 8 dequeuer threads are spawned. The job of each of these dequeuer threads is to remove the elements. Additionally, an array that keeps track of which elements have been already dequeued is maintained. The test checks that no element is removed more than once.

- testParallelBoth. This is the most complete test in the suite. It spawns 8 enqueuer threads and 8 dequeuer threads. Each enqueuer adds 512 elements to the queue. The dequeuers must remove each of these elements. A check similar to the one described in the *testParallelDequeue* test is performed.

7.1.4 Sample Results and Interpretation

As it is, the test fails as follows:

```
[oraadm@gdlaa008 Queue]$ junit queue.LockFreeQueueTest
.sequential push and pop
F.parallel both
.parallel deq
.parallel enq
E
Time: 0.012
There was 1 error:
1) testParallelEnq(queue.LockFreeQueueTest)queue.EmptyException
  at queue.LockFreeQueue.deq(LockFreeQueue.java:64)
  at queue.LockFreeQueueTest.testParallelEnq(LockFreeQueueTest.java:69)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
  at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
There was 1 failure:
1) testSequential(queue.LockFreeQueueTest)junit.framework.AssertionFailedError:
unexpected empty queue
  at queue.LockFreeQueueTest.testSequential(LockFreeQueueTest.java:51)
  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: 4,  Failures: 1,  Errors: 1
```

The problem with it, is that it is using a stub function for multi-compareAndSet:

```java
  private static <T> boolean multiCompareAndSet(
      AtomicReference<T>[] target,
      T[] expect,
```

```
      T[] update) {
    return true;
  }
```

Actually, what we did to get around it, is to implement the enqueue code as the book says, that is:

```
  /**
   * Append item to end of queue.
   * @param item
   */
  public void enq(T item) {
    if (item == null) throw new NullPointerException();
    Node node = new Node(item); // allocate & initialize new node
    while (true) {       // keep trying
      Node last = tail.get();      // read tail
      Node next = last.next.get(); // read next
      if (last == tail.get()) { // are they consistent?
        if (next == null) {
          if (last.next.compareAndSet(next,node)) {
            tail.compareAndSet(last,node);
            return;
          }
        }
        else {
          tail.compareAndSet(last,next);
        }
      }
    }
  }
```

After that, we were able to successfully execute the test cases:

```
[oraadm@gdlaa008 Queue]$ junit queue.LockFreeQueueTest
.sequential push and pop
.parallel both
.parallel deq
.parallel enq

Time: 0.016

OK (4 tests)
```

7.2 HW Queue

7.2.1 Particular Case

The HW Queue uses the Atomic values defined by the Java language in order to implement a locking free queue.

7.2.2 Solution

The unbounded queue implementation used by the *HWQueue* class is blocking, meaning that the *deq()* method does not return until it has found an item to dequeue.

```
public T deq() {
  while (true) {
    int range = tail.get();
    for (int i = 0; i < range; i++) {
      T value = items[i].getAndSet(null);
      if (value != null) {
        return value;
      }
    }
  }
}
```

The *enq()* method uses the *AtomicInteger getAndIncrement()* method that atomically incremented the counter.

```
public void enq(T x) {
  int i = tail.getAndIncrement();
  items[i].set(x);
}
```

7.2.3 Experiment Description

The test creates 8 threads that need to be coordinate in order to add and remove values from a list. All threads have to cooperate in order to parallel add and remove values from the list while avoiding duplicate remove of the values. If that is not the case, a fail will be raised.

7.2.4 Observations and Interpretations

The tests executed as expected and no errors where found.

```
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
    0,121 sec

------------- Standard Output ---------------
sequential push and pop
parallel enq
parallel deq
parallel both
------------- ---------------- ---------------
test-single:
BUILD SUCCESSFUL (total time: 1 second)
```

7.3 Bounded Partial Queue

7.3.1 Particular Case

In this experiment we implement a bounded queue as a linked list and using the Java *ReentrantLock* implementation for the enqueue-er and dequeue-er explicit call of the lock.

7.3.2 Solution

According to the theory there is a limited concurrency we can expect a bounded queue implementation with multiple concurrent enqueue-ers and dequeue-ers to provide, but can extend the enq() and deq() methods to operate on opposite ends of the queue, so as long as the queue is neither full nor empty, an enq() call and a deq() call should, in principle, be able to proceed without interference.

The *deq()* method reads the size field to check whether the queue is empty. If so, the dequeue-er must wait until an item is enqueued. The dequeue-er waits on *notEmptyCondition*, which temporarily releases *deqLock*, and blocks until the condition is signaled.

```
public T deq() {
  T result;
  boolean mustWakeEnqueuers = true;
  deqLock.lock();
  try {
    while (size.get() == capacity) {
      try {
        notEmptyCondition.await();
      } catch (InterruptedException ex) {}
    }
    result = head.next.value;
    head = head.next;
    if (size.getAndIncrement() == 0) {
      mustWakeEnqueuers = true;
    }
  } finally {
    deqLock.unlock();
  }
  if (mustWakeEnqueuers) {
    enqLock.lock();
    try {
```

```
        notFullCondition.signalAll();
      } finally {
        enqLock.unlock();
      }
    }
    return result;
  }
```

The *enq()* method thread acquires the *enqLock*, and reads the size field. While that field is equal to the capacity, the queue is full, and the enqueue-er must wait until a dequeue-er makes room. The enqueue-er waits on the *notFullCondition* field, releasing the enqueue lock temporarily, and blocking until that condition is signalled.

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

7.3.3 Experiment Description

The test creates 8 threads that need to be coordinate in order to *enq()* and *deq()* a range of numbers. All threads have to cooperate to add and remove elements from the queue. Each of the threads will enqueue and dequeue values into the queue, if everything works according to the test there will be mutual exclusion and the mapping of elements will corresponds to the queue elements. If that is not the case, a duplicate fail will be raised.

7.3.4 Observations and Interpretations

The tests executed as expected and no errors where found. Since the ReentrantLock is used to acquire an explicit lock, the mutual exclusion is guarantee by the Java java.util.concurrent.locks package.

```
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
    0,076 sec

------------- Standard Output ---------------
sequential push and pop
parallel enq
parallel deq
parallel both
------------- ---------------- ---------------
test-single:
BUILD SUCCESSFUL (total time: 1 second)
```

7.4 Unbounded Queue

7.4.1 Particular Case

In this experiment we implement a unbounded queue, the representation is the same as the bounded queue, except there is no need to count the number of items in the queue, or to provide conditions on which to wait.

7.4.2 Solution

According to the theory an item is actually enqueued when the *enq()* call sets the last nodes next field to the new node, even before *enq()* resets tail to refer to the new node. After that instant, the new item is reachable along a chain of the next references. The actual head is the successor of the node referenced by head, and the actual tail is the last item reachable from the head. Both the *enq()* and *deq()* methods are total as they do not wait for the queue to become empty or full.

```
public T deq() throws EmptyException {
  T result;
  deqLock.lock();
  try {
    if (head.next == null) {
      throw new EmptyException();
    }
    result = head.next.value;
    head = head.next;
  } finally {
    deqLock.unlock();
  }
  return result;
}
```

```
public void enq(T x) {
  if (x == null) throw new NullPointerException();
  enqLock.lock();
  try {
    Node e = new Node(x);
    tail.next = e;
    tail = e;
  } finally {
    enqLock.unlock();
  }
```

```
}
```

This queue cannot deadlock, because each method acquires only one lock, either *enqLock* or *deqLock* and is much simpler than the bounded queue.

7.4.3 Experiment Description

The test creates 8 threads that need to be coordinate in order to *enq()* and *deq()* a range of numbers. All threads have to cooperate to add and remove elements from the queue. Each of the threads will enqueue and dequeue values into the queue, if everything works according to the test there will be mutual exclusion and the mapping of elements will corresponds to the queue elements. If that is not the case, a duplicate fail will be raised.

7.4.4 Observations and Interpretations

The tests executed as expected and no errors where found. Since the *ReentrantLock* is used to acquire an explicit lock, the mutual exclusion is guarantee by the Java java.util.concurrent.locks package. As a interesting note, once each 100 executions the running time increases exponentially as shown below.

```
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
    10,102 sec

------------- Standard Output ---------------
sequential push and pop
parallel enq
parallel deq
parallel both
------------- ---------------- ---------------
test-single:
BUILD SUCCESSFUL (total time: 11 seconds)
```

7.5 SynchronousDualQueueTest

7.5.1 Particular Case (problem)

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

7.5.2 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.

7.5.3 Experiment Description

The test creates 16 threads, grouping them on enqueuers and dequeuers; for each group it divides the workload (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.

7.5.4 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:

```java
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.

7.6 SynchronousQueueTest

7.6.1 Particular Case

In this exercise we are dealing again with a first-in-first-out (FIFO) data structure, i.e. a Queue. The difference with respect of the previous examples is that in this case we have multiple producers and multiple consumers *rendezvous*ing with one another. In other words, a producer that puts an item in the queue blocks until that item is removed by a consumer.

7.6.2 Solution

The following is a naive implementation of a Synchronous queue. Notice that it contains the following fields:

```java
public class SynchronousQueue<T> {
  T item = null;
  boolean enqueuing;
  Lock lock;
  Condition condition;
      ...
      }
```

Let us talk about them:

- item. The first item in the queue (the one to be dequeued)
- enqueuing. Used to sync the enqueuers
- lock. Used for mutual exclusion
- condition. Used to block partial methods

Now, let us talk about how the enqueue and the dequeue actually work:

```java
  public T deq() throws InterruptedException {
    lock.lock();
    try {
      while (item == null) {
        condition.await();
      }
```

```
      T t = item;
      item = null;
      condition.signalAll();
      return t;
    } finally {
      lock.unlock();
    }
  }

  public void enq(T value) throws InterruptedException {
    if (value == null) throw new NullPointerException();
    try {
      while (enqueuing) { // another enqueuer's turn
        condition.await();
      }
      enqueuing = true; // my turn starts
      item = value;
      condition.signalAll();
      while (item != null) {
        condition.await();
      }
      enqueuing = false;  // my turn ends
      condition.signalAll();
    } finally {
      lock.unlock();
    }
  }
}
```

The enqueue method first has to check whether another enqueuer is enqueueng an element. If that is the case, then the enqueuer has to wait till the enqueuer is paired with a dequeuer. After that, it will set the flag *enqueuing* to true, sets the item and signals any other thread that is waiting for this one to finish its enqueue.

Once the enqueuer has put its value in the queue, it waits till someone consumes its value (*item* becomes *null*). Then the enqueuer sets the *enqueuing* flag back to false and finishes.

On the other hand, the dequeuer threads after acquiring the lock, they have to wait till an element is put in the queue. When it finds this is true, then they take the item and signal other threads to indicate they have finished dequeuing.

7.6.3 Experiment Description

This exercise consists of only one test and it works as follow:

It creates $2 * 8$ threads. Each of the threads will enqueue 512/8 elements to the queue. At the same time, 8 threads will try to dequeue the elements. When dequeueing, the test will put a value of *true* in the slot *arr*[*value*]. The test expects that each slot should not be set to *true* twice.

7.6.4 Sample Results

First, we need to mention that we had a problem compiling the test cases because we needed to add a try-catch block around the *enq() deq()* call like this:

```java
try{
  int value = (Integer)instance.deq();
  if (map[value]) {
    fail("DeqThread: duplicate pop");
  }
  map[value] = true;
} catch(InterruptedException ie) {
  System.out.println("Exception" + ie);
}
```

After that, we noticed that they consistently fails with the following error:

```
[oraadm@gdlaa008 Queue]$ junit queue.SynchronousQueueTest
.parallel both
Exception in thread "Thread-10" Exception in thread "Thread-6" Exception in thread "Thread-2" Exception in th
read "Thread-0" Exception in thread "Thread-8" Exception in thread "Thread-4" Exception in thread "Thread-14"
 Exception in thread "Thread-12" java.lang.IllegalMonitorStateException
        at java.util.concurrent.locks.ReentrantLock$Sync.tryRelease(ReentrantLock.java:155)
        at java.util.concurrent.locks.AbstractQueuedSynchronizer.release(AbstractQueuedSynchronizer.java:1260)
        at java.util.concurrent.locks.ReentrantLock.unlock(ReentrantLock.java:460)
        at queue.SynchronousQueue.enq(SynchronousQueue.java:63)
        at queue.SynchronousQueueTest$EnqThread.run(SynchronousQueueTest.java:61)
java.lang.IllegalMonitorStateException
        at java.util.concurrent.locks.ReentrantLock$Sync.tryRelease(ReentrantLock.java:155)
        at java.util.concurrent.locks.AbstractQueuedSynchronizer.release(AbstractQueuedSynchronizer.java:1260)
        at java.util.concurrent.locks.ReentrantLock.unlock(ReentrantLock.java:460)
        at queue.SynchronousQueue.enq(SynchronousQueue.java:63)
        at queue.SynchronousQueueTest$EnqThread.run(SynchronousQueueTest.java:61)
java.lang.IllegalMonitorStateException
        at java.util.concurrent.locks.ReentrantLock$Sync.tryRelease(ReentrantLock.java:155)
        at java.util.concurrent.locks.AbstractQueuedSynchronizer.release(AbstractQueuedSynchronizer.java:1260)
```

```
        at java.util.concurrent.locks.ReentrantLock.unlock(ReentrantLock.java:460)
        at queue.SynchronousQueue.enq(SynchronousQueue.java:63)
        at queue.SynchronousQueueTest$EnqThread.run(SynchronousQueueTest.java:61)
java.lang.IllegalMonitorStateException
        at java.util.concurrent.locks.ReentrantLock$Sync.tryRelease(ReentrantLock.java:155)
        at java.util.concurrent.locks.AbstractQueuedSynchronizer.release(AbstractQueuedSynchronizer.java:1260)
        at java.util.concurrent.locks.ReentrantLock.unlock(ReentrantLock.java:460)
        at queue.SynchronousQueue.enq(SynchronousQueue.java:63)
        at queue.SynchronousQueueTest$EnqThread.run(SynchronousQueueTest.java:61)
java.lang.IllegalMonitorStateException
        at java.util.concurrent.locks.ReentrantLock$Sync.tryRelease(ReentrantLock.java:155)
        at java.util.concurrent.locks.AbstractQueuedSynchronizer.release(AbstractQueuedSynchronizer.java:1260)
        at java.util.concurrent.locks.ReentrantLock.unlock(ReentrantLock.java:460)
        at queue.SynchronousQueue.enq(SynchronousQueue.java:63)
        at queue.SynchronousQueueTest$EnqThread.run(SynchronousQueueTest.java:61)
java.lang.IllegalMonitorStateException
        at java.util.concurrent.locks.ReentrantLock$Sync.tryRelease(ReentrantLock.java:155)
        at java.util.concurrent.locks.AbstractQueuedSynchronizer.release(AbstractQueuedSynchronizer.java:1260)
        at java.util.concurrent.locks.ReentrantLock.unlock(ReentrantLock.java:460)
        at queue.SynchronousQueue.enq(SynchronousQueue.java:63)
        at queue.SynchronousQueueTest$EnqThread.run(SynchronousQueueTest.java:61)
java.lang.IllegalMonitorStateException
        at java.util.concurrent.locks.ReentrantLock$Sync.tryRelease(ReentrantLock.java:155)
        at java.util.concurrent.locks.AbstractQueuedSynchronizer.release(AbstractQueuedSynchronizer.java:1260)
        at java.util.concurrent.locks.ReentrantLock.unlock(ReentrantLock.java:460)
        at queue.SynchronousQueue.enq(SynchronousQueue.java:63)
        at queue.SynchronousQueueTest$EnqThread.run(SynchronousQueueTest.java:61)
java.lang.IllegalMonitorStateException
        at java.util.concurrent.locks.ReentrantLock$Sync.tryRelease(ReentrantLock.java:155)
        at java.util.concurrent.locks.AbstractQueuedSynchronizer.release(AbstractQueuedSynchronizer.java:1260)
        at java.util.concurrent.locks.ReentrantLock.unlock(ReentrantLock.java:460)
        at queue.SynchronousQueue.enq(SynchronousQueue.java:63)
        at queue.SynchronousQueueTest$EnqThread.run(SynchronousQueueTest.java:61)
```

The problem is a bad use of the monitor API. Here is the corrected code:

```java
public void enq(T value) throws InterruptedException {
  if (value == null) throw new NullPointerException();
  lock.lock();
  try {
    while (enqueuing) { // another enqueuer's turn
      condition.await();
    }
    enqueuing = true; // my turn starts
    item = value;
    condition.signalAll();
    while (item != null) {
      condition.await();
    }
    enqueuing = false;  // my turn ends
    condition.signalAll();
  } catch(Exception e) {
```

```
        System.out.println("Exception: " + e);
    } finally {
        lock.unlock();
    }
}
```

After this fix, here is the result:

```
[oraadm@gdlaa008 Queue]$ junit queue.SynchronousQueueTest
.parallel both

Time: 0.167

OK (1 test)
```

7.7 LockFreeQueueRecycleTest

7.7.1 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.

7.7.2 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).

7.7.3 Experiment Description

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

7.7.4 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)
```

7.8 LockFreeQueueTest Test

7.8.1 Particular Case

In this exercise we deal with the problem of implementing a Queue data structure with the particular characteristic of being lock free.

7.8.2 Solution

Here we present the implementation of the Lock Free Queue that is based on the paper by Michael and Scott.

Here is the *enqueue()* method:

```java
/**
 * Append item to end of queue.
 * @param item
 */
public void enq(T item) {
  if (item == null) throw new NullPointerException();
  Node node = new Node(item); // allocate & initialize new node
  while (true) {                    // keep trying
    Node last = tail.get();     // read tail
    Node next = last.next.get(); // read next
    if (last == tail.get()) { // are they consistent?
      AtomicReference[] target = {last.next, tail};
      Object[] expect = {next, last};
      Object[] update = {node, node};
      if (multiCompareAndSet(
          (AtomicReference<T>[]) target,
          (T[]) expect,
          (T[]) update)){
        return;
      }
    }
  }
}
```

Here is the dequeue method.

```java
/**
 * Remove and return head of queue.
```

```
   * @return remove first item in queue
   * @throws queue.EmptyException
   */
  public T deq() throws EmptyException {
    while (true) {
      Node first = head.get();
      Node last = tail.get();
      Node next = first.next.get();
      if (first == head.get()) {// are they consistent?
        if (first == last) {    // is queue empty or tail falling behind?
          if (next == null) {   // is queue empty?
            throw new EmptyException();
          }
          // tail is behind, try to advance
          tail.compareAndSet(last, next);
        } else {
          T value = next.value; // read value before dequeuing
          if (head.compareAndSet(first, next))
            return value;
        }
      }
    }
```

7.8.3 Experiment Description

The test consists of 4 test cases. Let us briefly describe each one of them:

- testSequential. In this test case 512 elements are enqueued in the data structure. After that, each of the elements are dequeued. At each dequeue, we verify that the dequeued elements are in sequential order (just as they were inserted).

- testParallelEnqueue. This test cases spawns 8 enqueuer threads. Each of them will add 512 elements to the queue. When they are done, the main thread dequeues each of the elements performing the same check described above.

- testParallelDequeue. In this test case, the master thread enqueues 512 elements. When done, 8 dequeuer threads are spawned. The job of each of these dequeuer threads is to remove the elements. Additionally, an array that keeps track of which elements have been already dequeued is maintained. The test checks that no element is removed more than once.

- testParallelBoth. This is the most complete test in the suite. It spawns 8 enqueuer threads and 8 dequeuer threads. Each enqueuer adds 512 elements to the queue. The dequeuers must remove each of these elements. A check similar to the one described in the *testParallelDequeue* test is performed.

7.8.4 Sample Results and Interpretation

As it is, the test fails as follows:

```
[oraadm@gdlaa008 Queue]$ junit queue.LockFreeQueueTest
.sequential push and pop
F.parallel both
.parallel deq
.parallel enq
E
Time: 0.012
There was 1 error:
1) testParallelEnq(queue.LockFreeQueueTest)queue.EmptyException
  at queue.LockFreeQueue.deq(LockFreeQueue.java:64)
  at queue.LockFreeQueueTest.testParallelEnq(LockFreeQueueTest.java:69)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
  at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
There was 1 failure:
1) testSequential(queue.LockFreeQueueTest)junit.framework.AssertionFailedError:
unexpected empty queue
  at queue.LockFreeQueueTest.testSequential(LockFreeQueueTest.java:51)
  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: 4,  Failures: 1,  Errors: 1
```

The problem with it, is that it is using a stub function for multi-compareAndSet:

```java
private static <T> boolean multiCompareAndSet(
      AtomicReference<T>[] target,
      T[] expect,
```

```
        T[] update) {
    return true;
  }
```

Actually, what we did to get around it, is to implement the enqueue code as the book says, that is:

```
  /**
   * Append item to end of queue.
   * @param item
   */
  public void enq(T item) {
    if (item == null) throw new NullPointerException();
    Node node = new Node(item); // allocate & initialize new node
    while (true) {      // keep trying
      Node last = tail.get();     // read tail
      Node next = last.next.get(); // read next
      if (last == tail.get()) { // are they consistent?
        if (next == null) {
          if (last.next.compareAndSet(next,node)) {
            tail.compareAndSet(last,node);
            return;
          }
        }
        else {
          tail.compareAndSet(last,next);
        }
      }
    }
  }
```

After that, we were able to successfully execute the test cases:

```
[oraadm@gdlaa008 Queue]$ junit queue.LockFreeQueueTest
.sequential push and pop
.parallel both
.parallel deq
.parallel enq

Time: 0.016

OK (4 tests)
```

7.9 Bounded Partial Queue

7.9.1 Particular Case

In this experiment we implement a bounded queue as a linked list and using the Java *ReentrantLock* implementation for the enqueue-er and dequeue-er explicit call of the lock.

7.9.2 Solution

According to the theory there is a limited concurrency we can expect a bounded queue implementation with multiple concurrent enqueue-ers and dequeue-ers to provide, but can extend the enq() and deq() methods to operate on opposite ends of the queue, so as long as the queue is neither full nor empty, an enq() call and a deq() call should, in principle, be able to proceed without interference.

The *deq()* method reads the size field to check whether the queue is empty. If so, the dequeue-er must wait until an item is enqueued. The dequeue-er waits on *notEmptyCondition*, which temporarily releases *deqLock*, and blocks until the condition is signaled.

```
public T deq() {
  T result;
  boolean mustWakeEnqueuers = true;
  deqLock.lock();
  try {
    while (size.get() == capacity) {
      try {
        notEmptyCondition.await();
      } catch (InterruptedException ex) {}
    }
    result = head.next.value;
    head = head.next;
    if (size.getAndIncrement() == 0) {
      mustWakeEnqueuers = true;
    }
  } finally {
    deqLock.unlock();
  }
  if (mustWakeEnqueuers) {
    enqLock.lock();
    try {
```

```
        notFullCondition.signalAll();
      } finally {
        enqLock.unlock();
      }
    }
    return result;
  }
```

The *enq()* method thread acquires the *enqLock*, and reads the size field. While that field is equal to the capacity, the queue is full, and the enqueue-er must wait until a dequeue-er makes room. The enqueue-er waits on the *notFullCondition* field, releasing the enqueue lock temporarily, and blocking until that condition is signalled.

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

7.9.3 Experiment Description

The test creates 8 threads that need to be coordinate in order to *enq()* and *deq()* a range of numbers. All threads have to cooperate to add and remove elements from the queue. Each of the threads will enqueue and dequeue values into the queue, if everything works according to the test there will be mutual exclusion and the mapping of elements will corresponds to the queue elements. If that is not the case, a duplicate fail will be raised.

7.9.4 Observations and Interpretations

The tests executed as expected and no errors where found. Since the ReentrantLock is used to acquire an explicit lock, the mutual exclusion is guarantee by the Java java.util.concurrent.locks package.

```
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
    0,076 sec

------------- Standard Output ---------------
sequential push and pop
parallel enq
parallel deq
parallel both
------------- ---------------- ---------------
test-single:
BUILD SUCCESSFUL (total time: 1 second)
```

Chapter 8

Concurrent Stacks and Elimination

8.1 Exchanger

8.1.1 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 back-off stack*.

8.1.2 Solution

The solution in this exercise 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.

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

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

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.

```java
public V exchange(V x) throws InterruptedException {
  try {
    return (V)doExchange(x, Integer.MAX_VALUE);
  } catch (TimeoutException cannotHappen) {
    throw new Error(cannotHappen);
  }
}
public V exchange(V x, long timeout, TimeUnit unit)
throws InterruptedException, TimeoutException {
  return (V)doExchange(x, unit.toNanos(timeout));
}
```

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.

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

```
    int idx = 0;
    int backoff = 0;
    while (true) {
      AtomicReference<Node> slot = (AtomicReference<Node>)arena[idx];
      // If this slot is occupied, an item is waiting...
      Node you = slot.get();
      if (you != null) {
        Object v = you.fillHole(item);
        slot.compareAndSet(you, null);
        if (v != FAIL) // ... unless it's cancelled
          return v;
      }
      // Try to occupy this slot
      if (slot.compareAndSet(null, me)) {
        Object v = ((idx == 0)?
          me.waitForHole(nanos) :
          me.waitForHole(randomDelay(backoff)));
        slot.compareAndSet(me, null);
        if (v != FAIL)
          return v;
        if (Thread.interrupted())
          throw new InterruptedException();
        long now = System.nanoTime();
        nanos -= now - lastTime;
        lastTime = now;
        if (nanos <= 0)
          throw new TimeoutException();
        me = new Node(item);
        if (backoff < SIZE - 1)
          ++backoff;
        idx = 0; // Restart at top
      } else // Retry with a random non-top slot <= backoff
        idx = 1 + random.nextInt(backoff + 1);
    }
  }
  ...
```

The algorithm adds some other improvements to the base case already described. The main improvement is the back-off 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.

8.1.3 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 $1 \neq array[array[1]]$.

8.1.4 Sample Results

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

```
.exchange
OK

Time: 0.007

OK (1 test)
```

8.1.5 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 9

Counting, Sorting and Distributed Coordination

9.1 TreeTest

9.1.1 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.

9.1.2 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 intermediate result points. When a couple of threads collide in their attempt to increment, only one of them will serve as the representative and go up in the tree with the mission of propagating the combined increment (2) up to the shared counter at the root of the tree. In its way up, it may encounter further threads that collide again with it, making it wait or continue its journey to the root.

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

9.1.3 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 attempts results in a duplicated value, as well as to ensure that all threads completed their task.

9.1.4 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.

9.2 Balancer Test

9.2.1 Particular Case

The problem we are dealing with in this experiment is that of creating a balancer class.

As stated in the text book, a *Balancer* is a simple switch with two input wires and two output wires (top and bottom wires). The token arrives to the input lines and the balancer must output the input one at a time.

A balancer has two states: *up* and *down*. If the state is *up*, the next token exits on the top of the wire. Otherwise, it exits on the bottom wire.

9.2.2 Solution

The following is a simple implementation of a Balancer class. As described above, the balancer consists of a toggle.

```
public synchronized int traverse(int input) {
  try {
    if (toggle) {
      return 0;
    } else {
      return 1;
    }
  } finally {
    toggle = !toggle;
  }
}
```

As we can observe, the traverse method simply switches the state of the toggle and outputs either 0 or 1.

9.2.3 Experiment Description

The balancer class described above is used in one test case. The test case consists of a 2-slot array. Each slot keeps track of how many times the balancer has output a signal on the *up* wire and how many on the *down* wire.

So, the test activates the balancer exactly 256 times. The final result must be that each slot has the number 128. If that is the case, we declare the test as passed, otherwise it failed.

9.2.4 Sample Results and Interpretation

Here is the result of the execution. It passed every time:

```
[oraadm@gdlaa008 Counting]$ junit counting.BalancerTest
.
Time: 0.002

OK (1 test)
```

9.3 Bitonic Counting Network

9.3.1 Particular Case

In this experiment we implement a *Bitonic* counting network, a *Bitonic* network is a comparison-based sorting network that can be run in parallel. It focuses on converting a random sequence of numbers into a bitonic sequence, one that monotonically increases, then decreases. We represent this network as switches in a counting network, but as we use shared-memory multiprocessor however, we represent this network as objects in memory mimicking a real network.

9.3.2 Solution

According to the theory each balancer is an object, whose wires are references from one balancer to another. Each thread repeatedly traverses the object, starting on some input wire and emerging at some output wire, effectively shepherding a token through the network. The *Balancer* class has a single Boolean field: toggle. The synchronized *traverse()* method complements the toggle field and returns as output wire, either 0 or 1.

```
public synchronized int traverse(int input) {
  try {
    if (toggle) {
      return 0;
    } else {
      return 1;
    }
  } finally {
    toggle = !toggle;
  }
}
```

The *Merger* class has three fields; the *width* field must be a power of 2, *half[]* is a two-element array of half-width *Merger* objects, and *layer[]* is an array of width/2 balancers implementing the final network layer. The class provides a *traverse(i)* method, where i is the wire on which the token enters. If the token entered on the lower width/2 wires, then it passes through *half[0]*, otherwise *half[1]*.

```
public Merger(int _size) {
  size = _size;
```

```
    layer = new Balancer[size / 2];
    for (int i = 0; i < size / 2; i++) {
      layer[i] = new Balancer();
    }
    if (size > 2) {
      half = new Merger[]{new Merger(size/2), new Merger(size/2)
          };
    }
  }

  public int traverse(int input) {
    int output = 0;
    if (size > 2) {
      output = half[input % 2].traverse(input / 2);
    }
    return output + layer[output].traverse(0);
  }
```

The *Bitonic* class provides a *traverse(i)* method. If the token entered on the lower width/2 wires, then it passes through *half[0]*, otherwise *half[1]*. A token that emerges from the half-merger subnetwork on wire i then traverses the final merger network from input wire i.

9.3.3 Experiment Description

The test creates a 256 array that get values mapped to it and later checks that the steps where properly executes, by validating the result in the *position-1* of each elements the bitonic sorted should have accommodated the array into a proper sequence. If the result doesn't match the mapping a new exception is thrown.

9.3.4 Observations and Interpretations

The tests executed as expected and no errors where found. The bitonic counting network as represented through software is replicating hardware that is well understood and easy to follow.

```
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
    0,052 sec

test:
Deleting: /var/folders/hn/vlmcc1gj1kl7bb9rpqb8tdqr0000gn/T/TEST-
    counting.BitonicTest.xml
```

```
BUILD SUCCESSFUL (total time: 0 seconds)
```

Chapter 10

Concurrent Hashing and Natural Parallelism

10.1 CoarseHashSetTest

10.1.1 Particular Case

The problem we are trying to solve now is that of building a concurrent Hash data structure. In the particular case described in the book, the hash data structure is used to construct a representation of a set ADT. Hence, we ought to provide three operations:

1. add. Aggregate a new element to the set
2. remove. Remove a given element from the set
3. contains. Determine whether a given element is already in the set

10.1.2 Solution

Here we present a first approach to solving the problem stated above. We do so by providing a coarse version. As in other coarse data structures, the problem with this implementation is its poor performance in parallel environments.

Let us briefly describe how this implementation works. First we should mention that we have a base abstract class that implements the three operations mentioned previously. We extend this base class by providing four methods:

- resize. Resizes the hash based on a policy. In this implementation, it simply doubles the size of t he hash.
- acquire. Acquires a lock on the hash.
- release. Releases the lock.
- policy. Returns true if each bucket contains more than 4 elements.

```
/**
 * double the set size
 */
public void resize() {
  int oldCapacity = table.length;
```

```
      lock.lock();
      try {
        if (oldCapacity != table.length) {
          return; // someone beat us to it
        }
        int newCapacity  = 2 * oldCapacity;
        List<T>[] oldTable = table;
        table = (List<T>[]) new List[newCapacity];
        for (int i = 0; i < newCapacity; i++)
          table[i] = new ArrayList<T>();
        for (List<T> bucket : oldTable) {
          for (T x : bucket) {
            int myBucket = Math.abs(x.hashCode() % table.length);
            table[myBucket].add(x);
          }
        }
      } finally {
        lock.unlock();
      }
    }
    /**
     * Synchronize before adding, removing, or testing for item
     * @param x item involved
     */
    public final void acquire(T x) {
      lock.lock();
    }
    /**
     * synchronize after adding, removing, or testing for item
     * @param x item involved
     */
    public void release(T x) {
      lock.unlock();
    }
    public boolean policy() {
      return size / table.length > 4;
    }
```

This implementation is said to be coarse because any operation on the hash locks the entire data structure.

10.1.3 Experiment Description

As in other experiments, here we have four test cases:

- testSequential. The master thread adds 512 elements to the set, then it verifies that these elements were actually added and finally removes them.
- testParallelEnq. It creates 8 *adders* threads. Each of them will add 512 elements to the set. When all of them are done, the master thread checks that all the elements were actually added and finally removes them.
- testParallelRemove. In this test case, the master thread adds 512 elements to the set and verifies that the elements were actually added. After that, 8 *remover* threads are spawned. Each of them will remove 512/8 elements.
- testParallelBoth. This test is a mix of the two above. 8 *removers* and 8 *adders* are spawned. Removers should remove elements added by the adders.

10.1.4 Sample Results and Interpretation

Here is the result of the execution:

```
[oraadm@gdlaa008 Hash]$ junit hash.CoarseHashSetTest
.sequential add, contains, and remove
.parallel both
Exception in thread "Thread-7" Exception in thread "Thread-9"
junit.framework.AssertionFailedError: DeqThread: duplicate pop
        at junit.framework.Assert.fail(Assert.java:57)
        at junit.framework.TestCase.fail(TestCase.java:227)
        at hash.CoarseHashSetTest$RemoveThread.run(CoarseHashSetTest.java:143)
Exception in thread "Thread-13" junit.framework.AssertionFailedError: DeqThread:
duplicate pop
        at junit.framework.Assert.fail(Assert.java:57)
        at junit.framework.TestCase.fail(TestCase.java:227)
        at hash.CoarseHashSetTest$RemoveThread.run(CoarseHashSetTest.java:143)
junit.framework.AssertionFailedError: DeqThread: duplicate pop
        at junit.framework.Assert.fail(Assert.java:57)
        at junit.framework.TestCase.fail(TestCase.java:227)
        at hash.CoarseHashSetTest$RemoveThread.run(CoarseHashSetTest.java:143)
Exception in thread "Thread-15" junit.framework.AssertionFailedError: DeqThread:
duplicate pop
        at junit.framework.Assert.fail(Assert.java:57)
        at junit.framework.TestCase.fail(TestCase.java:227)
        at hash.CoarseHashSetTest$RemoveThread.run(CoarseHashSetTest.java:143)
.parallel add
.parallel remove
```

```
Time: 0.043

OK (4 tests)
```

The test complains saying that when we add and remove elements from the hash table concurrently the remover threads are removing an element twice. However, we found that this is not necesarily correct because the test simply assumes that if a remove fails, it is because we have repeated elements.

If we look into the code carefully, we can see that a remove operation can also fail if it is the case that it is trying to remove an element that has not been inserted yet.

So, in order to make this test work, we have to garantee that removers try to remove elements that have been actually added to the set. We leave this enhancement as a future work.

10.2 StripedHashSetTest

10.2.1 Particular Case

In this exercise we deal with the problem of building a hash data structure that can be accessed by multiple threads without losing performance.

In the coarse version of our hash, we saw that any access to the data structure ended up being serial. So, in other words, in this experiment we want to understand how to create a hash that allows more concurrent accesses.

10.2.2 Solution

A solution for this problem is the *Striped Hash*. The main characteristic of this implementation is that the data structure contains an array of locks. Each lock protect specific sections of the hash table. Here is the implementation of the *acquire()* and *release()* methods:

```java
/**
 * Synchronize before adding, removing, or testing for item
 * @param x item involved
 */
public final void acquire(T x) {
  int myBucket = Math.abs(x.hashCode() % locks.length);
  locks[myBucket].lock();
}

/**
 * synchronize after adding, removing, or testing for item
 * @param x item involved
 */
public void release(T x) {
  int myBucket = Math.abs(x.hashCode() % locks.length);
  locks[myBucket].unlock();
}
```

Note that when we acquire or release a lock, first we need to choose which lock we should work with.

10.2.3 Experiment Description

The proposed experiment is exactly the same as the one described in the previous experiment. As mentioned before, there are four variants: *testSequential*, *testParallelEnq*, *testParallelRemove* and *testParallelBoth*.

10.2.4 Sample Results and Interpretation

Here is the result that we got:

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

Time: 0.027

OK (4 tests)
```

However, some people in the group mentioned that they had the same test issue as mentioned in the Coarse Hash Set exercise. Again, we think that it is a test issue since there is no guarantee that removers will remove elements that have been really inserted.

10.3 CoarseCuckooHashSetTest

10.3.1 Particular Case (problem)

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

10.3.2 Solution

The solution uses three techniques combined:

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

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

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

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

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

public boolean add(T x) {
  lock.lock();
  try {
```

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

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

10.3.3 Experiment Description

The test program consists of three test cases:

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

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

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

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

10.3.4 Observations and Interpretations

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

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

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

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

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

...

.parallel add
.parallel remove

Time: 0.053

OK (4 tests)
```

10.4 LockFreeHashSetTest

10.4.1 Particular Case (problem)

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

10.4.2 Solution

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

10.4.3 Experiment Description

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

10.4.4 Observations and Interpretations

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

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

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

Time: 0.037

OK (4 tests)
```

10.5 Cuckoo Hashing

10.5.1 Particular Case

Cuckoo hashing is a (sequential) hashing algorithm in which a newly added item displaces any earlier item occupying the same slot. In this test we modify the sequential Cuckoo hashing in order to change the sequential hashing algorithm to concurrent hashing.

10.5.2 Solution

We break up each method call into a sequence of phases, where each phase adds, removes, or displaces a single item x. We use a two-entry array *table[]* of tables, and two independent hash functions, (denoted as *hash0()* and *hash1()* in the code) mapping the set of possible keys to entries in the array.

```
public TCuckooHashSet(int capacity) {
  locks = new Lock[2][LOCKS];
  table = (T[][]) new Object[2][capacity];
  size = capacity;
  for (int i = 0; i < 2; i++) {
    for (int j = 0; j < LOCKS; j++) {
      locks[i][j] = new ReentrantLock();
    }
  }
}
private final int hash0(Object x) {
  return Math.abs(x.hashCode() % size);
}
private final int hash1(Object x) {
  random.setSeed(x.hashCode());
  return random.nextInt(size);
}
```

To test whether a value x is in the set, *contains(x)* tests whether either table[0][h0(x)] or $table[1][h1(x)]$ is equal to x. Similarly, *remove(x)* checks whether x is in either $table[0][h0(x)]$ or $table[1][h1(x)]$, and removes it if found.

```
public boolean remove(T x) {
  if (x == null) {
    throw new IllegalArgumentException();
  }
```

```
    int h0 = hash0(x);
    Lock lock0 = locks[0][h0 % LOCKS];
    try {
      lock0.lock();
      if (x.equals(table[0][h0])) {
        table[0][h0] = null;
        return true;
      } else {
        int h1 = hash1(x);
        Lock lock1 = locks[1][h1 % LOCKS];
        try {
          lock1.lock();
          if (x.equals(table[1][h1])) {
            table[1][h1] = null;
            return true;
          }
          return false;
        } finally {
          lock1.unlock();
        }
      }
    } finally {
      lock0.unlock();
    }
  }
```

The *add(x)* method is the most interesting. It successively removes conflicting items until every key has a slot. To add x, the method swaps x with y, the current occupant of $table[0][h0(x)]$. If the prior value y was null, it is done, otherwise, it swaps the newly nestles value y for the current occupant of $table[1][h1(y)]$ in the same way and continues swapping entries (alternating tables) until it finds an empty slot.

10.5.3 Experiment Description

The test creates 8 threads that perform *enq()* and *deq()* operations in sequential and parallel. The dequeue values are revised and if a value is missing a fail assertion is thrown.

10.5.4 Observations and Interpretations

When executing the test the test fails on the parallel part due to a missing value when executing the parallel both test.

```
Exception in thread "Thread-5" Exception in thread "Thread-7"
    junit.framework.AssertionFailedError: Th 0          DeqThread:
    missing value: 3
```

10.5.5 Proposed changes to fix the problem

By adding terminal output through System.out in the *remove()* method or Debugging the file the execution works, thus becoming a problem of observation, since the fail assertion only happens when its not observed.

```
compile-test:
.sequential add, contains, and remove
.parallel add
.parallel remove
.parallel both

Time: 0.071

OK (4 tests)
```

10.5.6 Proposed solution

It is not clear why the observation would affect the execution, thus it is possible that the issue is a propagation of the values in the memory which are forced to be executed constantly through the debug implementation.

10.6 Refinable Hash Set

10.6.1 Particular Case

In this test we want to refine the granularity of locking as the table size grows, so that the number of locations in a stripe does not continuously grow by using a globally shared owner field that combines a *Boolean* value with a reference to a thread.

10.6.2 Solution

While a resizing is in progress, the *owner* Boolean value is true, and the associated reference indicates the thread that is in charge of resizing. We use the owner as a mutual exclusion flag between the *resize()* method and any of the *add()* methods, so that while resizing, there will be no successful updates, and while updating, there will be no successful resizes.

```
public void resize() {
  int oldCapacity = table.length;
  int newCapacity = 2 * oldCapacity;
  Thread me = Thread.currentThread();
  if (owner.compareAndSet(null, me, false, true)) {
    try {
      if (table.length != oldCapacity) {  // someone else resized first
        System.out.println("Someone else resizd first");
        return;
      }
      quiesce();
      List<T>[] oldTable = table;
      table = (List<T>[]) new List[newCapacity];
      for (int i = 0; i < newCapacity; i++)
        table[i] = new ArrayList<T>();
      locks = new ReentrantLock[newCapacity];
      for (int j = 0; j < locks.length; j++) {
        locks[j] = new ReentrantLock();
      }
      initializeFrom(oldTable);
    } finally {
      owner.set(null, false);        // restore prior state
    }
  }
}
```

The *acquire()* and the *resize()* methods guarantee mutually exclusive access via the flag principle using the mark field of the *owner* flag and the tables locks array, *acquire()* first acquires its locks and then reads the mark field, while *resize()* first sets mark and then reads the locks during the *quiesce()* call. This ordering ensures that any thread that acquires the locks after *quiesce()* has completed will see that the set is in the processes of being resized, and will back off until the resizing is complete.

```java
protected void quiesce() {
  for (ReentrantLock lock : locks) {
    while (lock.isLocked()) {}  // spin
  }
}
```

```java
public void acquire(T x) {
  boolean[] mark = {true};
  Thread me = Thread.currentThread();
  Thread who;
  while (true) {
    do { // wait until not resizing
      who = owner.get(mark);
    } while (mark[0] && who != me);
    ReentrantLock[] oldLocks = this.locks;
    int myBucket = Math.abs(x.hashCode() % oldLocks.length);
    ReentrantLock oldLock = oldLocks[myBucket];
    oldLock.lock();  // acquire lock
    who = owner.get(mark);
    if ((!mark[0] || who == me) && this.locks == oldLocks) {
        // recheck
      return;
    } else {  //  unlock & try again
      oldLock.unlock();
    }
  }
}
```

10.6.3 Experiment Description

The test creates 8 threads that perform *enq()* and *deq()* operations in sequential and parallel. The dequeue values are revised and if a value is missing a fail assertion is thrown.

10.6.4 Observations and Interpretations

When executing the test the test fails on the parallel part due to a duplicate value when executing the parallel both test.

```
Assertion error: duplicate pop Exception in thread "Thread-13"
    junit.framework.AssertionFailedError: DeqThread: duplicate
    pop
```

10.6.5 Proposed changes to fix the problem

By adding terminal output through System.out in the *remove()* method or Debugging the file the execution works, thus becoming a problem of observation, since the fail assertion only happens when its not observed, this happens with the TCuckooHash and CuckooHas tests.

```
compile-test:
.sequential add, contains, and remove
.parallel add
.parallel remove
.parallel both

Time: 0.081

OK (4 tests)
```

10.6.6 Proposed solution

It is not clear why the observation would affect the execution, thus it is possible that the issue is a propagation of the values in the memory which are forced to be executed constantly through the debug implementation. Also worth mentioning that several hashing test suffer from this same situation and adding volatile didn't help as in other tests.

10.7 RefinableCuckooHashSetTest

10.7.1 Particular Case

The particular problem we are trying to solve here is how to implement a concurrent open-addressed hash (specifically a hash set).

We ought to remember that an open-addressed hash table is one in which each table entry holds one and only one item. In previous examples we had seen that an entry in the table can handle multiple elements which are stored in a linked list within the bucket.

The other thing we have to consider is that in this exercise we are willing to have a concurrent hash. Previously we saw a open-addressed hash that allowed coarse granularity. Here we want to go beyond that.

10.7.2 Solution

The solution to this problem is given by what is called the *Refinable Concurrent Cuckoo Hash*. The difference between this implementation and previously presented *Cuckoo Hashes* is that this one is a fine-grained approach where a lock is always responsible for protecting a fixed number of elements in the hash.

The interesting bits of this implementation are the *acquire()*, *release()* and *resize()* methods. Let us take a quick look at those:

```
/**
 * Synchronize before adding, removing, or testing for item
 * @param x item involved
 */
public void acquire(T x) {
  boolean[] mark = {true};
  Thread me = Thread.currentThread();
  Thread who;
  while (true) {
    do { // wait until not resizing
      who = owner.get(mark);
    } while (mark[0] && who != me);
    ReentrantLock[][] oldLocks = this.locks;
    ReentrantLock oldLock0 = oldLocks[0][hash0(x) % oldLocks[0].length];
    ReentrantLock oldLock1 = oldLocks[1][hash1(x) % oldLocks[1].length];
    oldLock0.lock();  // acquire locks
    oldLock1.lock();
```

```
      who = owner.get(mark);
      if ((!mark[0] || who == me) && this.locks == oldLocks) { // recheck
        return;
      } else { // unlock & try again
        oldLock0.unlock();
        oldLock1.unlock();
      }
    }
  }
```

The *acquire()* method first spins till if finds that no one is doing a resizing. This means that resizing and adding are exclusive operations. After this step, the thread acquires the two locks that protect the intended slot in the table. If after this point, the lock array is still the same we saw before taking the lock, then we are free to continue. Otherwise we attempt to take the locks again.

The *release()* method simply unlocks the slot:

```
  /**
   * synchronize after adding, removing, or testing for item
   * @param x item involved
   */
  public final void release(T x) {
    Lock lock0 = locks[0][hash0(x) % locks[0].length];
    Lock lock1 = locks[1][hash1(x) % locks[1].length];
    lock0.unlock();
    lock1.unlock();
  }
```

The *resize()* method is essentially the same as the one showed in the previous exercise.

10.7.3 Experiment Description

As for the tests provided for this experiment. They are the same discussed in previous exercises in this chapter. Nothing new has been added, so let us go directly to the results.

10.7.4 Sample Results

Here is the result of the execution. The tests passed as expected:

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

Time: 0.018

OK (4 tests)
```

10.8 StripedCuckooHashSetTest

10.8.1 Particular Case (problem)

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

10.8.2 Solution

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

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

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

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

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

10.8.3 Experiment Description

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

10.8.4 Observations and Interpretations

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

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

Time: 0.017

OK (4 tests)
```

Chapter 11

Skiplists and Balanced Search

11.1 Double Ended Queue

11.1.1 Particular Case

In this experiment each thread keeps a pool of tasks waiting to be executed in the form of a double-ended queue (DEQueue), providing *pushBottom()*, *popBottom()*, and *popTop()* methods.

11.1.2 Solution

According to the theory when a thread creates a new task, it calls *pushBottom()* to push that task onto its *DEQueue*. When a thread needs a task to work on, it calls *popBottom()* to remove a task from its own *DEQueue*. If the thread discovers its queue is empty, then it becomes a thief, it chooses a victim thread at random, and calls that threads DEQueues *popTop()* method to steal a task for itself.

```
public Runnable popTop() {
  int[] stamp = new int[1];
  int oldTop = top.get(stamp), newTop = oldTop + 1;
  int oldStamp = stamp[0], newStamp = oldStamp + 1;
  if (bottom <= oldTop) // empty
    return null;
  Runnable r= tasks[oldTop];
  if (top.compareAndSet(oldTop, newTop, oldStamp, newStamp))
    return r;
  return null;
}
```

11.1.3 Experiment Description

The test creates 16 threads, eight that need to be coordinate in order to *pushBottom()* and *popBottom()* an array of values. All threads have to cooperate to add and remove elements from the queue and eight *Dummy()* threads will be stealing execution from the running ones. After each thread execution, if everything works according to the test the map will have all values set to true. If that is not the case, a duplicate or missing fail will be raised.

11.1.4 Observations and Interpretations

The tests executed as expected and no errors where found.

```
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
    1.162 sec

------------- Standard Output ---------------
sequential pushBottom and popBottom
concurrent pushBottom and popBottom
------------- ---------------- ---------------
test-single:
BUILD SUCCESSFUL (total time: 2 seconds)
```

11.2 Parallel Matrix Multiplication

11.2.1 Particular Case

In this experiment, the particular case we are trying to study is matrix parallel multiplication by devising methods to add and multiply matrices.

11.2.2 Solution

To implement a parallel multiplication we create a Matrix class that provides *set()* and *get()* methods to access matrix elements, along with a constant-time *split()* method that splits an n-by-n matrix into four $(n/2)$-by-$(n/2)$ submatrices.

```
double get(int row, int col) {
  return data[row+rowDisplace][col+colDisplace];
}

void set(int row, int col, double value) {
  data[row+rowDisplace][col+colDisplace] = value;
}

Matrix[][] split() {
  Matrix[][] result = new Matrix[2][2];
  int newDim = dim / 2;
  result[0][0] = new Matrix(data, rowDisplace, colDisplace,
      newDim);
  result[0][1] = new Matrix(data, rowDisplace, colDisplace +
      newDim, newDim);
  result[1][0] = new Matrix(data, rowDisplace + newDim,
      colDisplace, newDim);
  result[1][1] = new Matrix(data, rowDisplace + newDim,
      colDisplace + newDim, newDim);
  return result;
}
```

For simplicity, we consider matrices whose dimension n is a power of 2. Any such matrix can be decomposed into four submatrices, thus their sums can be done in parallel.

11.2.3 Experiment Description

The test creates two matrices with a dimension of four which will be multiplied in parallel, the resulting matrix will be verified and a fail assertion will be thrown is the result is incorrect.

11.2.4 Observations and Interpretations

When compiling the code the Matrix class had to be changed from private to public and the offset and dimension of the Matrix must be added to the test class as shown below.

```
    Matrix aa = new Matrix(a, 0, 0, 4);
    Matrix bb = new Matrix(b, 0, 0, 4);
```

After making this changes the codes compiles and everything works fine.

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
    0.369 sec

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

11.2.5 Proposed changes to fix the problem

The MatrixTask class needs to have the visibility of the inner Matrix class change to public in order for the test to view it.

```
  public static class Matrix {
    int dim;
    double[][] data;
    int rowDisplace;
    int colDisplace;
    ---
  }
```

11.3 Unbounded Double Ended Queue

11.3.1 Particular Case

In this experiment we implement an unbounded double queue based on the previous double queue implemented.

11.3.2 Solution

According to the theory the *UnboundedDEQueue* class dynamically resizes itself as needed. We implement the *UnboundedDEQueue* in a cyclic array, with top and bottom fields as in the *BoundedDEQueue* (except indexed modulo the array's capacity). As before, if bottom is less than or equal to top, the *UnboundedDEQueue* is empty. The difference resides in the *CircularArray()* class which provides *get()* and *put()* methods that add and remove tasks, and a *resize()* method that allocates a new circular array and copies the old array's contents into the new array.

```
class CircularArray {
  private int logCapacity;
  private Runnable[] currentTasks;
  CircularArray(int logCapacity) {
    this.logCapacity = logCapacity;
    currentTasks = new Runnable[1 << logCapacity];
  }
  int capacity() {
    return 1 << this.logCapacity;
  }
  Runnable get(int i) {
    return this.currentTasks[i % capacity()];
  }
  void put(int i, Runnable task) {
    this.currentTasks[i % capacity()] = task;
  }
  CircularArray resize(int bottom, int top) {
    CircularArray newTasks =
        new CircularArray(this.logCapacity+1);
    for (int i = top; i < bottom; i++) {
      newTasks.put(i, this.get(i));
    }
    return newTasks;
  }
}
```

11.3.3 Experiment Description

The test creates 16 threads, eight that need to be coordinate in order to *pushBottom()* and *popBottom()* an array of values. All threads have to cooperate to add and remove elements from the queue and eight *Dummy()* threads will be stealing execution from the running ones. After each thread execution, if everything works according to the test the map will have all values set to true. If that is not the case, a duplicate or missing fail will be raised.

11.3.4 Observations and Interpretations

The test needs some changes in order to compile, once this changes are implemented the test executes as expected.

```
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
    1.4 sec

------------- Standard Output ---------------
sequential pushBottom and popBottom
concurrent pushBottom and popBottom
------------- ---------------- ---------------
test-single:
BUILD SUCCESSFUL (total time: 1 second)
```

11.3.5 Proposed changes to fix the problem

The UDEQueueTest class needs to have replaced the calls to non defined reset by new instance in order to compile.

```
  //instance.reset();
  instance = new UDEQueue(THREADS);
```

11.4 ArraySumTest

11.4.1 Particular Case (problem)

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

11.4.2 Solution

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

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

11.4.3 Experiment Description

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

11.4.4 Observations and Interpretations

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

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

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

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

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

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

11.5 BDEQueueTest

11.5.1 Particular Case

Note: We noticed that this problem is repeated. Actually, it should not go here. It corresponds to the chapter of Scheduling. Including it here just to follow the order of the programs in the web site.

The problem we want to solve in this exercise is how do we perform Work Distribution among threads in an effective way.

11.5.2 Solution

The solution proposed in this exercise is by using Work-Stealing Dequeues. Specifically, we will use a bounded version.

The idea is that we will have a pool of tasks and each task has a queue of Work to be done. It is possible that some threads finish the jobs in their queues faster than other. In such situation, this algorithm will allow faster threads to steal jobs from other threads' queues.

The implementation of this algorithm requires a DEQueue (Double Ended Queue). When picking a job from a queue, we distinguish two cases. The first one, which is the most common one, is that one thread picks a job from its own queue. In that case, the thread uses the *popBottom()* method.

The other case is when a thread steals a job from another queue. In that case, the thread uses the *popTop()* method.

Let us take a look at the interesting methods.

popBottom() distinguishes between two cases. If there is a conflict between a *popBottom()* and *popTop()*, then it resets the bookmark of the top of the queue using a *compareAndSet()*. If it succeeds, then it means that our method won and it returns the element. Other wise it means that the stealer won, and we have to return null.

If there is no conflict between the two pop methods, then we simply return the element. There is no need to call a *compareAndSet()*.

```
  Runnable popBottom() {
    // is the queue empty?
    if (bottom == 0) // empty  `\label{line:steal:empty}`
      return null;
    bottom--;
    // bottom is volatile to assure all reads beyond this line see it
```

```
    Runnable r = tasks[bottom];
    int[] stamp = new int[1];
    int oldTop = top.get(stamp), newTop = 0;
    int oldStamp = stamp[0], newStamp = oldStamp + 1;
    // no conflict with thieves ‘\label{line:steal:noconflict}‘
    if (bottom > oldTop)
      return r;
    // possible conflict: try to pop it ourselves
    if (bottom == oldTop) {
      // even if stolen by other, queue will be empty, reset bottom
      bottom = 0;
      if (top.compareAndSet(oldTop, newTop, oldStamp, newStamp))
        return r;
      }
    return null;
  }
```

11.5.3 Experiment Description

Two test cases are provided with this program:

- testSequential. 16 threads are spawned. Each thread pushes a value in our DEQueue. After that, even threads call *popTop()* and odd threads call *popBottom()*. Depending on the pop'd value, an associated slot in an array is marked as true. The test checks that no slot in the array is marked true twice, since that would mean that a conflict between the pop methods was not resolved correctly.

- testConcurrent. Again, 16 threads are spanwed and each thread first pushes a value into our DEQueue. After that, all threads are in competition to pop the values. The idea is that some threads will call *popTop()* to steal from other's queues. However, at the end we should still see the same invariant as in the previous test case.

11.5.4 Sample Results and Interpretation

The result of the execution of the test cases was as follows:

```
[oraadm@gdlaa008 Steal]$ junit steal.BDEQueueTest
.sequential pushBottom and popBottom
```

```
.concurrent pushBottom and popBottom

Time: 0.055

OK (2 tests)
```

The tests passed every time

11.6 FibTaskTest

11.6.1 Particular Case (problem)

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

11.6.2 Solution

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

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

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

11.6.3 Experiment Description

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

11.6.4 Observations and Interpretations

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

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

Time: 0.128

OK (1 test)
```

11.7 FibThreadTest

11.7.1 Particular Case

Note: We noticed that this problem is repeated. Actually, it should not go here. It corresponds to the chapter of Futures. Including it here just to follow the order of the programs in the web site.

Here we are dealing of computing the Fibonacci numbers in parallel. In particular we want to do so by spawning slave threads.

11.7.2 Solution

The way to implement a parallel generator of Fibonacci number is by creating two threads to calculate $F(n-1)$ and $F(n-2)$. This process is repeated recursively. When each thread finishes calculating its values, the parent thread adds the value of the children. Here is the code:

```java
public FibThread(int n) {
  arg = n;
  result = -1;
}

public void run() {
  FibThread left, right;
  if (arg < 2) {
    result = arg;
  } else {
    left = new FibThread(arg-1);
    right = new FibThread(arg-2);
    left.start();
    right.start();
    try {
     left.join();
     right.join();
    } catch (InterruptedException e) {};
    result = left.result + right.result;
  }
}
```

11.7.3 Experiment Description

The test provided for this exercise consists of calculating $F(16)$. This result must be equal to 987. If that is the case, the test passes. Otherwise, it fails.

11.7.4 Sample Results

11.7.5 Interpretation

As it is mentioned in the book, for this kind of applications where a lot of short living threads is very inefficient. Instead, we should consider using a thread pool.

11.7.6 Sample Results and Interpretation

This is the result of the execution of the test:

```
[oraadm@gdlaa008 Steal]$ junit steal.FibThreadTest
.run

Time: 0.825

OK (1 test)
```

This indicates that the concurrent implementation of the Fibonacci sequence by using threads works correctly

11.8 PolynomialTaskTest

11.8.1 Particular Case

Note: We noticed that this problem is repeated. Actually, it should not go here. It corresponds to the chapter of Futures. Including it here just to follow the order of the programs in the web site.

The particular problem we want to solve in this exercise is how can we implement an efficient polynomial parallel adder and multiplier.

11.8.2 Solution

The solution to this problem requires the use of a pool of threads, which are called *executor services* in Java. In this case, we will use Runnable objects. These Runnable objects return what are called Futures. Futures of Runnables provide the *get()* method. This method blocks the caller till the computation finishes.

Let us examine the code to understand how the Futures are used to solve the Polynomial problem.

Here is the *MulTask* Runnable. It requires 3 references to polynomials: the multipliers and the result. When it is executed, it will split each of the multipliers into two, say p_1, p_2 and q_1, q_2. The concatenation of p_1 and p_2 is the original p polynomial. Given this partition, the result of a multiplication should be $p_1 * q_1 + p_1 * q_2 + p_2 * q_1 + p_2 * q_2$. So, the task of multiplying two polynomials is decomposed into 4 multiplications and 3 sums.

```java
static class MulTask implements Runnable {
  Polynomial p, q, r;
  Polynomial[] temp;
  public MulTask(Polynomial p, Polynomial q, Polynomial r) {
    this.p = p; this.q = q; this.r = r;
    int newDegree = 2 * p.getDegree();
    temp = new Polynomial[6];
    for (int i  = 0; i < temp.length; i++) {
      temp[i] = new Polynomial(newDegree);
    }
  }
  public void run() {
    try {
      if (p.getDegree() == 0) {
```

```
            r.set(0, p.get(0) * q.get(0));
        } else {
            Polynomial[] pp = p.split();
            Polynomial[] qq = q.split();
            Future<?>[] future = (Future<?>[]) new Future[7];
            // launch parallel multiplications
            future[0] = exec.submit(new MulTask(pp[0], qq[0], temp[0]));
            future[1] = exec.submit(new MulTask(pp[0], qq[1], temp[1]));
            future[2] = exec.submit(new MulTask(pp[1], qq[0], temp[2]));
            future[3] = exec.submit(new MulTask(pp[1], qq[1], temp[3]));
            // wait for them to finish
            for (int i = 0; i < 4; i++)
              future[i].get();
            // do partial sums
            future[5] = exec.submit(new AddTask(temp[0], temp[1], temp[4]));
            future[6] = exec.submit(new AddTask(temp[2], temp[3], temp[5]));
            // wait for them to finish
            future[5].get();
            future[6].get();
            // final sum
            future[7] = exec.submit(new AddTask(temp[4], temp[5], r));
            future[7].get();
          }
        } catch (Exception ex) {
          ex.printStackTrace();
        }
      }
    }
```

Here we have now the *AddTask*. This Runnable is simpler. It again requires 3 references to polynomials. The execution consists of splitting each polynomial into two. After that, sub-polynomials are added one to one.

```
static class AddTask implements Runnable {
  Polynomial p, q, r;
  public AddTask(Polynomial p, Polynomial q, Polynomial r) {
    this.p = p; this.q = q; this.r = r;
  }
  public void run() {
    try {
      int n = p.getDegree();
      if (n == 1) {
        r.set(0, p.get(0) + q.get(0));
      } else {
```

```
            Polynomial[] pp = p.split(), qq = q.split(), rr = r.split();
            Future<?>[] future = (Future<?>[]) new Future[2];
            // create asynchronous computations
            future[0] = exec.submit(new AddTask(pp[0], qq[0], rr[0]));
            future[1] = exec.submit(new AddTask(pp[1], qq[1], rr[1]));
            // wait for them to finish
            future[0].get();
            future[1].get();
          }
        } catch (Exception ex) {
          ex.printStackTrace();
        }
      }
    }
```

11.8.3 Experiment Description

Now let us discuss the experiment. It is actually a sum of the polynomial $x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$ by itself. The result of this sum should be $2x^7 + 2x^6 + 2x^5 + 2x^4 + 2x^3 + 2x^2 + 2x + 2$.

11.8.4 Sample Results and Interpretation

First, we had to fix the test case. The right one is as follows:

```
    /**
     * Test of run method, of class steal.PolynomialTask.
     */
    public void testRun() throws InterruptedException, ExecutionException {
      System.out.println("run");

      int[] a = {1, 1, 1, 1, 1, 1, 1, 1};
      int[] b = {1, 1, 1, 1, 1, 1, 1, 1};
      Polynomial aa = new Polynomial(a, 0, 8);
      Polynomial bb = new Polynomial(b, 0, 8);
      Polynomial cc = PolynomialTask.add(aa, bb);
      for (int i = 0; i < cc.getDegree(); i++) {
        assertEquals(2, cc.get(i));
      }
      Polynomial dd = PolynomialTask.multiply(aa,bb);
    }
```

After this fix, we had the following failure:

```
[oraadm@gdlaa008 Steal]$ junit steal.PolynomialTaskTest
.run
java.lang.ArrayIndexOutOfBoundsException: 7
  at steal.PolynomialTask$MulTask.run(PolynomialTask.java:102)
  at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471)
  at java.util.concurrent.FutureTask$Sync.innerRun(FutureTask.java:334)
  at java.util.concurrent.FutureTask.run(FutureTask.java:166)
  at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
  at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
  at java.lang.Thread.run(Thread.java:724)
...
java.lang.ArrayIndexOutOfBoundsException: 7
  at steal.PolynomialTask$MulTask.run(PolynomialTask.java:102)
  at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471)
  at java.util.concurrent.FutureTask$Sync.innerRun(FutureTask.java:334)
  at java.util.concurrent.FutureTask.run(FutureTask.java:166)
  at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
  at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
  at java.lang.Thread.run(Thread.java:724)
java.lang.ArrayIndexOutOfBoundsException: 7
  at steal.PolynomialTask$MulTask.run(PolynomialTask.java:102)
  at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471)
  at java.util.concurrent.FutureTask$Sync.innerRun(FutureTask.java:334)
  at java.util.concurrent.FutureTask.run(FutureTask.java:166)
  at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
  at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
  at java.lang.Thread.run(Thread.java:724)

Time: 0.13

OK (1 test)
```

The problem is that the size of the *futures* array was not big enough. It is 7, but it actually required 8 elements. After that fix, we were able to run the test successfully every time:

```
[oraadm@gdlaa008 Steal]$ junit steal.PolynomialTaskTest
.run

Time: 0.072

OK (1 test)
```

11.9 MMThreadTest

11.9.1 Particular Case (problem)

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

11.9.2 Solution

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

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

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

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

```
        }
    }
```

11.9.3 Experiment Description

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

11.9.4 Observations and Interpretations

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

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

Time: 0.004

OK (1 test)
```

Chapter 12

Priority Queues

12.1 BitReversedCounterTest

12.1.1 Particular Case (problem)

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

12.1.2 Solution

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

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

12.1.3 Experiment Description

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

12.1.4 Observations and Interpretations

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

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

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

Time: 0.022

OK (1 test)
```

12.2 SimpleTreeTest

12.2.1 Particular Case

The problem we want to solve now is how we can implement a bounded priority queue that can be accessed concurrently by multiple threads.

Remember that priority queue is said to be *bounded* when the range of the priorities are taken from the range $0, ..., m - 1$.

12.2.2 Solution

One possible solution to this problem is given by the *Tree-Based Bounded Priority Queue* data structure. This structure is essentially a binary tree where the leaf nodes have a bin holding items of priority i. Internal nodes maintain a shared counter that indicates the number of elements stored to the left of a given node.

The interesting methods in this data structure are the *add()* and *removeMin()*, so let us talk about them.

The *add()* method first takes the bin associated with the given priority. It puts the item into that bin and then starts increasing the counter of the parent nodes all the way to the root. Note that it only increases the counter if the current node happens to be a left child of the parent.

```
/**
 *  add item to priority queue
 * @param item new item
 * @param priority item''s priority
 */
public void add(T item, int priority) {
  TreeNode node = leaves.get(priority);
  node.bin.put(item);
  while(node != root) {
    TreeNode parent = node.parent;
    if (node == parent.left) { // increment if ascending from left
      parent.counter.getAndIncrement();
    }
    node = parent;
  }
}
```

The *removeMin()* method has to retrieve the item with the lowest priority. In order to do that, the method starts from the root and traverses the tree. If the current counter of the inner node is 0, then it knows that the min must be on the left. Otherwise, it goes through the right child.

```java
public T removeMin() {
  TreeNode node = root;
  while(!node.isLeaf()) {
    if (node.counter.getAndDecrement() > 0 ) {
      node = node.left;
    } else {
      node = node.right;
    }
  }
  return node.bin.get(); // if null pqueue is empty
}
```

12.2.3 Experiment Description

Three test cases were provided to exercise this code:

- testAdd. This test adds elements with random priority to the priority queue one by one. At the end, it extracts each element and it must be the case that the previously removed element had a lower priority than the current element.
- testParallelAdd. This test spawns 8 threads. Each of the test adds 8 elements to the queue concurrently. At the end, it performs the same checking as in the previous test case. Notice that each thread is in charge of adding elements from a fixed range in ascending order.
- testParallelBoth. Does the same as the previous test case but also removes elements in parallel. Each remover thread will perform the same checking. Since adders add elements in ascending order, it is guaranteed that any removal from any thread will retrieve an element with a lower priority than the next one.

12.2.4 Sample Results

Here is the result of the execution of the test cases:

```
[oraadm@gdlaa008 ch15]$ junit priority.SimpleTreeTest
.sequential test
OK.
.testParallelBoth
OK.
.testParallelAdd
OK.

Time: 0.01

OK (3 tests)
```

The tests passed every time

12.3 Sequential Heap

12.3.1 Particular Case

In this experiment we implement a linearisable priority queue that supports priorities from an unbounded range. It uses fine-grained locking for synchronization.

12.3.2 Solution

According to the theory a sequential heap implementation represent a binary heap by using an array of nodes, where the trees root is array entry 1, and the right and left children of array entry i are entries $2i$ and $(2i)+1$, respectively. Each node has an item and a score field. To add an item, the *add()* method sets child to the index of the first empty array slot.

```
public void add(T item, int priority) {
  int child = next++;
  heap[child].init(item, priority);
  while (child > ROOT) {
    int parent = child / 2;
    int oldChild = child;
    if (heap[child].priority < heap[parent].priority) {
      swap(child, parent);
      child = parent;
    } else {
      return;
    }
  }
}
```

To remove and return the highest-priority item, the *removeMin()* method records the roots item, which is the highest-priority item in the tree.

```
public T removeMin() {
  int bottom = --next;
  T item = heap[ROOT].item;
  swap(ROOT, bottom);
  if (bottom == ROOT) {
    return item;
  }
  int child = 0;
  int parent = ROOT;
  while (parent < heap.length / 2) {
```

```
      int left = parent * 2; int right = (parent * 2) + 1;
      if (left >= next) {
        break;
      } else if (right >= next || heap[left].priority < heap[right].priority) {
        child = left;
      } else {
        child = right;
      }
      // If child higher priority than parent swap then else stop
      if (heap[child].priority < heap[parent].priority) {
        swap(parent, child); // Swap item, key, and tag of heap[i] and heap[child]
        parent = child;
      } else {
        break;
      }
    }
    return item;
  }
```

12.3.3 Experiment Description

The test adds 512 elements to the heap, does a check of the elements and removes the values with min priority. We verify the priorities of the removed elements and assert a fail if we have non ascending priorities.

12.3.4 Observations and Interpretations

The test execute as expected and no problems are thrown.

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.413 sec

------------- Standard Output ---------------
testSequential
OK.
------------- ---------------- ---------------
test-single:
BUILD SUCCESSFUL (total time: 1 second)
```

12.4 Skiplist-Based Unbounded Priority Queue

12.4.1 Particular Case

In this experiment we implement a skip-list priority queue which in contrast to the *FineGrainedHeap* priority queue algorithm, that the underlying heap structure requires complex, coordinated rebalancing. We requires no rebalancing.

12.4.2 Solution

According to the theory the *PrioritySkipList* class sorts items by priority instead of by hash value, ensuring that high-priority items (the ones we want to remove first) appear at the front of the list. Removing the item with highest priority is done lazily. A node is logically removed by marking it as removed, and is later physically removed by unlinking it from the list.

```
public T removeMin() {
    Node<T> node = skiplist.findAndMarkMin();

    if (node != null) {
      skiplist.remove(node);
      return node.item;
    } else{
      return null;
    }
}
```

The *add()* and *remove()* calls take skiplist nodes instead of items as arguments and results. These methods are straightforward adaptations of the corresponding *LockFreeSkipList* methods. This class nodes differ from *LockFreeSkipList* nodes in two fields, an integer score field, and an *AtomicBoolean* marked field used for logical deletion from the priority queue (not from the skiplist).

```
boolean add(Node node) {
  int bottomLevel = 0;
  Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
  Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
  while (true) {
    boolean found = find(node, preds, succs);
    if (found) { // if found it's not marked
      return false;
```

```
    } else {
      for (int level = bottomLevel; level <= node.topLevel; level++) {
        Node<T> succ = succs[level];
        node.next[level].set(succ, false);
      }
      // try to splice in new node in bottomLevel going up
      Node<T> pred = preds[bottomLevel];
      Node<T> succ = succs[bottomLevel];
      node.next[bottomLevel].set(succ, false);
      if (!pred.next[bottomLevel].compareAndSet(succ, node, false, false)) {// lin point
        continue; // retry from start
      }
      // splice in remaining levels going up
      for (int level = bottomLevel+1; level <= node.topLevel; level++) {
        while (true) {
          pred = preds[level];
          succ = succs[level];
          if (pred.next[level].compareAndSet(succ, node, false, false))
            break;
          find(node, preds, succs); // find new preds and succs
        }
      }
      return true;
    }
  }
}

boolean remove(Node<T> node) {
  int bottomLevel = 0;
  Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
  Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
  Node<T> succ;
  while (true) {
    boolean found = find(node, preds, succs);
    if (!found) {
      return false;
    } else {
      // proceed to mark all levels
      // some levels could stil be unthreaded by concurrent add() while being marked
```

```java
          // other find()s could be modifying node's pointers concurrently
          for (int level = node.topLevel; level >= bottomLevel+1; level--) {
            boolean[] marked = {false};
            succ = node.next[level].get(marked);
            while (!marked[0]) { // until I succeed in marking
              node.next[level].attemptMark(succ, true);
              succ = node.next[level].get(marked);
            }
          }
          // proceed to remove from bottom level
          boolean[] marked = {false};
          succ = node.next[bottomLevel].get(marked);
          while (true) { // until someone succeeded in marking
            boolean iMarkedIt = node.next[bottomLevel].compareAndSet(succ, succ, false, true);
            succ = succs[bottomLevel].next[bottomLevel].get(marked);
            if (iMarkedIt) {
              // run find to remove links of the logically removed node
              find(node, preds, succs);
              return true;
            } else if (marked[0]) return false; // someone else removed node
            // else only succ changed so repeat
          }
        }
      }
    }
```

12.4.3 Experiment Description

The test creates 8 threads, eight that need to be coordinate in order to *add()* and *removeMin()* an array of values. All threads have to cooperate to add and remove elements from the queue and a validation of the priorities is done to validate non-ascending priorities. If that is not the case, a fail will be raised.

12.4.4 Observations and Interpretations

The test needs some changes in order to execute and avoid starvation.

```
------------- Standard Output ---------------
add
OK.
testParallelAdd
OK.
testParallelBoth
```

12.4.5 Proposed changes to fix the problem

The *findAndMarkMin()* method must choose the coresponding element that will be removed in order to mark it and later on the *remove()* method we verify the marked node and remove it, but the atomicity of this operations are not guarantee, thus when a node is marked it can be marked by other thread causing the remove process to hand, since the internal logic tries to remove the node and has no escape sequence when the mark is affected.

```java
  public T removeMin() {
    synchronized(this) {
      Node<T> node = skiplist.findAndMarkMin();

      if (node != null) {
        skiplist.remove(node);
        return node.item;
      } else{
        return null;
      }
    }
  }
```

By doing so, the execution finishes as expected and no errors are found.

```
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
    0.358 sec

------------- Standard Output ---------------
add
OK.
testParallelAdd
OK.
testParallelBoth
```

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

12.5 FineGrainedHeapTest

12.5.1 Particular Case

Now we look at the problem of implementing an Unbounded Priority Queue in a Fine-Grained fashion.

Again, remember that Unbounded in this context means that the range is unbounded, for example the set of 32-bits integers or floating point values.

12.5.2 Solution

The solution proposed in this exercise is the Fine-Grained heap. The particular characteristic of this implementation is that percolation happens in a sequence of discrete atomic steps that can be interleaved with other such steps. The same happens with the *removeMin()* method.

The *add()* method first locks the heap and then takes the lock to the next available child and puts the new value there. After that, it releases the heap lock. After this point, it must percolate this new node. To do this, it has to swap child and parent after locking them. This only happens if the parent is available and the child has a higher priority. Otherwise, the node is already where it belongs. At the end of this step, child and parent are unlocked. Then it goes up one node in the heap, and does the same.

```
/**
 * Add item to heap.
 * @param item Uninterpreted item.
 * @param priority item priority
 */
public void add(T item, int priority) {
  heapLock.lock();
  int child = next++;
  heap[child].lock();
  heapLock.unlock();
  heap[child].init(item, priority);
  heap[child].unlock();
  while (child > ROOT) {
    int parent = child / 2;
    heap[parent].lock();
    heap[child].lock();
    int oldChild = child;
    try {
```

```
      if (heap[parent].tag == Status.AVAILABLE && heap[child].amOwner()) {
        if (heap[child].score < heap[parent].score) {
          swap(child, parent);
          child = parent;
        } else {
          heap[child].tag = Status.AVAILABLE;
          heap[child].owner = NO_ONE;
          return;
        }
      } else if (!heap[child].amOwner()) {
        child = parent;
      }
    } finally {
      heap[oldChild].unlock();
      heap[parent].unlock();
    }
  }
  if (child == ROOT) {
    heap[ROOT].lock();
    if (heap[ROOT].amOwner()) {
      heap[ROOT].tag = Status.AVAILABLE;
      heap[child].owner = NO_ONE;
    }
    heap[ROOT].unlock();
  }
}
```

The *removeMin()* method first locks the heap, the bottom and the root nodes. If the root is empty, then we are done. Otherwise, it swaps root and bottom and unlocks bottom. After that, it has to percolate the new root element. It follows a similar approach as the one mentioned in the *add()* method. One difference is that this percolation goes from top to bottom.

```
/**
 * Returns and removes lowest-priority item in heap.
 * @return lowest-priority item.
 */
public T removeMin() {
  heapLock.lock();
  int bottom = --next;
  heap[bottom].lock();
  heap[ROOT].lock();
  heapLock.unlock();
```

```
        if (heap[ROOT].tag == Status.EMPTY) {
            heap[ROOT].unlock();
            heap[bottom].lock();
            return null;
        }
        T item = heap[ROOT].item;
        heap[ROOT].tag = Status.EMPTY;
        swap(bottom, ROOT);
        heap[bottom].owner = NO_ONE;
        heap[bottom].unlock();
        if (heap[ROOT].tag == Status.EMPTY) {
            heap[ROOT].unlock();
            return item;
        }
        int child = 0;
        int parent = ROOT;
        while (parent < heap.length / 2) {
            int left = parent * 2;
            int right = (parent * 2) + 1;
            heap[left].lock();
            heap[right].lock();
            if (heap[left].tag == Status.EMPTY) {
                heap[right].unlock();
                heap[left].unlock();
                break;
            } else if (heap[right].tag == Status.EMPTY || heap[left].score < heap[right].s
                heap[right].unlock();
                child = left;
            } else {
                heap[left].unlock();
                child = right;
            }
            if (heap[child].score < heap[parent].score) {
                swap(parent, child);
                heap[parent].unlock();
                parent = child;
            } else {
                heap[child].unlock();
                break;
            }
        }
        heap[parent].unlock();
        return item;
    }
```

12.5.3 Experiment Description

The same three test cases were provided in this experiment, so let us go directly to the results.

12.5.4 Sample Results and Interpretation

Here is a sample output of the test demonstrating that the implementation of this concurrent heap works properly:

```
[oraadm@gdlaa008 ch15]$ junit priority.FineGrainedHeapTest
.testSequential
OK.
.testParallelBoth
OK.
.testParallelAdd
OK.

Time: 3.175

OK (3 tests)
```

One interesting thing about this exercise is that it takes longer than other tests. Actually, other students shared the results with us and all of them experimented different execution times. For example:

- Our machine, that has 24cores, executed the test in 3.175 seconds.
- On a 6-cores machine, it took 1-6 secs.
- On a 2-cores machine, it took 8-9 secs.
- On a 100-cores machine it took > 5min and it did not finish.

As a future work, we might want to understand why we had these execution times.

12.6 SimpleLinearTest

12.6.1 Particular Case (problem)

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

12.6.2 Solution

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

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

  ...

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

  ...

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

```
    }
}
```

12.6.3 Experiment Description

The test program consists of the following test cases:

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

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

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

12.6.4 Observations and Interpretations

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

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

Time: 0.116

OK (3 tests)
```

Chapter 13

Futures, Scheduling and Work Distribution

13.1 ArraySumTest

13.1.1 Particular Case (problem)

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

13.1.2 Solution

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

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

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

13.1.3 Experiment Description

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

13.1.4 Observations and Interpretations

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

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

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

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

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

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

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

13.2 BDEQueueTest Test

13.2.1 Particular Case

The problem we want to solve in this exercise is how do we perform Work Distribution among threads in an effective way.

13.2.2 Solution

The solution proposed in this exercise is by using Work-Stealing Dequeues. Specifically, we will use a bounded version.

The idea is that we will have a pool of tasks and each task has a queue of Work to be done. It is possible that some threads finish the jobs in their queues faster than other. In such situation, this algorithm will allow faster threads to steal jobs from other threads' queues.

The implementation of this algorithm requires a DEQueue (Double Ended Queue). When picking a job from a queue, we distinguish two cases. The first one, which is the most common one, is that one thread picks a job from its own queue. In that case, the thread uses the *popBottom()* method.

The other case is when a thread steals a job from another queue. In that case, the thread uses the *popTop()* method.

Let us take a look at the interesting methods.

popBottom() distinguishes between two cases. If there is a conflict between a *popBottom()* and *popTop()*, then it resets the bookmark of the top of the queue using a *compareAndSet()*. If it succeeds, then it means that our method won and it returns the element. Other wise it means that the stealer won, and we have to return null.

If there is no conflict between the two pop methods, then we simply return the element. There is no need to call a *compareAndSet()*.

```
  Runnable popBottom() {
    // is the queue empty?
    if (bottom == 0)  // empty  `\label{line:steal:empty}`
      return null;
    bottom--;
    // bottom is volatile to assure all reads beyond this line see it
    Runnable r = tasks[bottom];
    int[] stamp = new int[1];
    int oldTop = top.get(stamp), newTop = 0;
```

```
    int oldStamp = stamp[0], newStamp = oldStamp + 1;
    // no conflict with thieves ‘\label{line:steal:noconflict}‘
    if (bottom > oldTop)
      return r;
    // possible conflict: try to pop it ourselves
    if (bottom == oldTop) {
      // even if stolen by other, queue will be empty, reset bottom
      bottom = 0;
      if (top.compareAndSet(oldTop, newTop, oldStamp, newStamp))
        return r;
      }
    return null;
  }
```

13.2.3 Experiment Description

Two test cases are provided with this program:

- testSequential. 16 threads are spawned. Each thread pushes a value in our DEQueue. After that, even threads call *popTop()* and odd threads call *popBottom()*. Depending on the pop'd value, an associated slot in an array is marked as true. The test checks that no slot in the array is marked true twice, since that would mean that a conflict between the pop methods was not resolved correctly.

- testConcurrent. Again, 16 threads are spawned and each thread first pushes a value into our DEQueue. After that, all threads are in competition to pop the values. The idea is that some threads will call *popTop()* to steal from other's queues. However, at the end we should still see the same invariant as in the previous test case.

13.2.4 Sample Results and Interpretation

The result of the execution of the test cases was as follows:

```
[oraadm@gdlaa008 Steal]$ junit steal.BDEQueueTest
.sequential pushBottom and popBottom
.concurrent pushBottom and popBottom

Time: 0.055

OK (2 tests)
```

The tests passed every time

13.3 FibThreadTest Test

13.3.1 Particular Case

Here we are dealing of computing the Fibonacci numbers in parallel. In particular we want to do so by spawning slave threads.

13.3.2 Solution

The way to implement a parallel generator of Fibonacci number is by creating two threads to calculate $F(n-1)$ and $F(n-2)$. This process is repeated recursively. When each thread finishes calculating its values, the parent thread adds the value of the children. Here is the code:

```
public FibThread(int n) {
  arg = n;
  result = -1;
}

public void run() {
  FibThread left, right;
  if (arg < 2) {
    result = arg;
  } else {
    left = new FibThread(arg-1);
    right = new FibThread(arg-2);
    left.start();
    right.start();
    try {
     left.join();
     right.join();
    } catch (InterruptedException e) {};
    result = left.result + right.result;
  }
}
```

13.3.3 Experiment Description

The test provided for this exercise consists of calculating $F(16)$. This result must be equal to 987. If that is the case, the test passes. Otherwise, it fails.

13.3.4 Sample Results

13.3.5 Interpretation

As it is mentioned in the book, for this kind of applications where a lot of short living threads is very inefficient. Instead, we should consider using a thread pool.

13.3.6 Sample Results and Interpretation

This is the result of the execution of the test:

```
[oraadm@gdlaa008 Steal]$ junit steal.FibThreadTest
.run

Time: 0.825

OK (1 test)
```

This indicates that the concurrent implementation of the Fibonacci sequence by using threads works correctly

13.4 FibTaskTest

13.4.1 Particular Case (problem)

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

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

13.4.2 Solution

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

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

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

13.4.3 Experiment Description

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

13.4.4 Observations and Interpretations

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

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

Time: 0.176

OK (1 test)
```

13.5 PolynomialTaskTest

13.5.1 Particular Case

The particular problem we want to solve in this exercise is how can we implement an efficient polynomial parallel adder and multiplier.

13.5.2 Solution

The solution to this problem requires the use of a pool of threads, which are called *executor services* in Java. In this case, we will use Runnable objects. These Runnable objects return what are called Futures. Futures of Runnables provide the *get()* method. This method blocks the caller till the computation finishes.

Let us examine the code to understand how the Futures are used to solve the Polynomial problem.

Here is the *MulTask* Runnable. It requires 3 references to polynomials: the multipliers and the result. When it is executed, it will split each of the multipliers into two, say p_1, p_2 and q_1, q_2. The concatenation of p_1 and p_2 is the original p polynomial. Given this partition, the result of a multiplication should be $p_1 * q_1 + p_1 * q_2 + p_2 * q_1 + p_2 * q_2$. So, the task of multiplying two polynomials is decomposed into 4 multiplications and 3 sums.

```java
static class MulTask implements Runnable {
  Polynomial p, q, r;
  Polynomial[] temp;
  public MulTask(Polynomial p, Polynomial q, Polynomial r) {
    this.p = p; this.q = q; this.r = r;
    int newDegree = 2 * p.getDegree();
    temp = new Polynomial[6];
    for (int i  = 0; i < temp.length; i++) {
      temp[i] = new Polynomial(newDegree);
    }
  }
  public void run() {
    try {
      if (p.getDegree() == 0) {
        r.set(0, p.get(0) * q.get(0));
      } else {
        Polynomial[] pp = p.split();
        Polynomial[] qq = q.split();
```

```java
          Future<?>[] future = (Future<?>[]) new Future[7];
          // launch parallel multiplications
          future[0] = exec.submit(new MulTask(pp[0], qq[0], temp[0]));
          future[1] = exec.submit(new MulTask(pp[0], qq[1], temp[1]));
          future[2] = exec.submit(new MulTask(pp[1], qq[0], temp[2]));
          future[3] = exec.submit(new MulTask(pp[1], qq[1], temp[3]));
          // wait for them to finish
          for (int i = 0; i < 4; i++)
            future[i].get();
          // do partial sums
          future[5] = exec.submit(new AddTask(temp[0], temp[1], temp[4]));
          future[6] = exec.submit(new AddTask(temp[2], temp[3], temp[5]));
          // wait for them to finish
          future[5].get();
          future[6].get();
          // final sum
          future[7] = exec.submit(new AddTask(temp[4], temp[5], r));
          future[7].get();
        }
      } catch (Exception ex) {
        ex.printStackTrace();
      }
    }
  }
```

Here we have now the *AddTask*. This Runnable is simpler. It again requires 3 references to polynomials. The execution consists of splitting each polynomial into two. After that, sub-polynomials are added one to one.

```java
  static class AddTask implements Runnable {
    Polynomial p, q, r;
    public AddTask(Polynomial p, Polynomial q, Polynomial r) {
      this.p = p; this.q = q; this.r = r;
    }
    public void run() {
      try {
        int n = p.getDegree();
        if (n == 1) {
          r.set(0, p.get(0) + q.get(0));
        } else {
          Polynomial[] pp = p.split(), qq = q.split(), rr = r.split();
          Future<?>[] future = (Future<?>[]) new Future[2];
          // create asynchronous computations
          future[0] = exec.submit(new AddTask(pp[0], qq[0], rr[0]));
```

```java
            future[1] = exec.submit(new AddTask(pp[1], qq[1], rr[1]));
            // wait for them to finish
            future[0].get();
            future[1].get();
          }
        } catch (Exception ex) {
          ex.printStackTrace();
        }
      }
    }
```

13.5.3 Experiment Description

Now let us discuss the experiment. It is actually a sum of the polynomial $x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$ by itself. The result of this sum should be $2x^7 + 2x^6 + 2x^5 + 2x^4 + 2x^3 + 2x^2 + 2x + 2$.

13.5.4 Sample Results and Interpretation

First, we had to fix the test case. The right one is as follows:

```java
    /**
     * Test of run method, of class steal.PolynomialTask.
     */
    public void testRun() throws InterruptedException, ExecutionException {
      System.out.println("run");

      int[] a = {1, 1, 1, 1, 1, 1, 1, 1};
      int[] b = {1, 1, 1, 1, 1, 1, 1, 1};
      Polynomial aa = new Polynomial(a, 0, 8);
      Polynomial bb = new Polynomial(b, 0, 8);
      Polynomial cc = PolynomialTask.add(aa, bb);
      for (int i = 0; i < cc.getDegree(); i++) {
        assertEquals(2, cc.get(i));
      }
      Polynomial dd = PolynomialTask.multiply(aa,bb);
    }
```

After this fix, we had the following failure:

```
[oraadm@gdlaa008 Steal]$ junit steal.PolynomialTaskTest
.run
java.lang.ArrayIndexOutOfBoundsException: 7
  at steal.PolynomialTask$MulTask.run(PolynomialTask.java:102)
  at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471)
  at java.util.concurrent.FutureTask$Sync.innerRun(FutureTask.java:334)
  at java.util.concurrent.FutureTask.run(FutureTask.java:166)
  at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
  at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
  at java.lang.Thread.run(Thread.java:724)
...
java.lang.ArrayIndexOutOfBoundsException: 7
  at steal.PolynomialTask$MulTask.run(PolynomialTask.java:102)
  at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471)
  at java.util.concurrent.FutureTask$Sync.innerRun(FutureTask.java:334)
  at java.util.concurrent.FutureTask.run(FutureTask.java:166)
  at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
  at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
  at java.lang.Thread.run(Thread.java:724)
java.lang.ArrayIndexOutOfBoundsException: 7
  at steal.PolynomialTask$MulTask.run(PolynomialTask.java:102)
  at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471)
  at java.util.concurrent.FutureTask$Sync.innerRun(FutureTask.java:334)
  at java.util.concurrent.FutureTask.run(FutureTask.java:166)
  at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
  at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
  at java.lang.Thread.run(Thread.java:724)

Time: 0.13

OK (1 test)
```

The problem is that the size of the *futures* array was not big enough. It is 7, but it actually required 8 elements. After that fix, we were able to run the test successfully every time:

```
[oraadm@gdlaa008 Steal]$ junit steal.PolynomialTaskTest
.run

Time: 0.072

OK (1 test)
```

13.6 MMThreadTest

13.6.1 Particular Case (problem)

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

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

13.6.2 Solution

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

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

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

class Worker extends Thread {
  int row, col;
  Worker(int row, int col) {
    this.row = row; this.col = col;
  }
  public void run() {
```

```
        double dotProduct = 0.0;
        for (int i = 0; i < n; i++) {
          dotProduct += a[row][i] * b[i][col];
        }
        c[row][col] = dotProduct;
      }
    }
```

13.6.3 Experiment Description

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

13.6.4 Observations and Interpretations

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

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

Time: 0.004

OK (1 test)
```

Chapter 14

Barriers

14.1 SenseBarrierTest

14.1.1 Particular Case (problem)

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

14.1.2 Solution

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

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

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

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

```
  }

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

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

14.1.3 Experiment Description

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

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

14.1.4 Observations and Interpretations

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

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

Time: 0.012

OK (1 test)
```

14.2 Reusable Barrier

14.2.1 Particular Case

In this experiment we implement an n-thread reusable barrier from an n-wire counting network and a single Boolean variable.

14.2.2 Solution

To implement a reusable barrier, we use a Combining Tree Barrier implementation and change the constructor in order to support the reuse of the barrier.

```java
public RevBarrier(int _size, int _radix) {
  radix = _radix;
  leaves = 0;
  this.leaf = new Node[_size / _radix];
  int depth = 0;
  threadSense = new ThreadLocal<Boolean>() {
    protected Boolean initialValue() { return true; };
  };
  // compute tree depth
  while (_size > 1) {
    depth++;
    _size = _size / _radix;
    assert _size > 0;
  }
  Node root = new Node();
  build(root, depth - 1);
}

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

14.2.3 Experiment Description

The test creates 8 threads that need to use the barrier *await()* in order to write values into the log and let the writer finish. If the round doesn't correspond to the value in the log then a System.out showing the expected value and found are displayed.

14.2.4 Observations and Interpretations

The tests executed as expected and no errors where found.

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.511 sec

------------- Standard Output ---------------
Testing 8 threads, 8 rounds
------------- ---------------- ---------------
test-single:
BUILD SUCCESSFUL (total time: 0 seconds)
```

14.3 TreeBarrierTest

14.3.1 Particular Case (problem)

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

14.3.2 Solution

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

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

```
    }
    Node root = new Node();
    build(root, depth - 1);
  }

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

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

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

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

14.3.3 Experiment Description

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

The test above is basically the same as that described for *DisBarrierTest* and *SenseBarrierTest*; except that we have a couple of tests instead of one (each one tries a different radix [1] and number of threads; the combinations of these two parameters are $threads = 8$,$radix = 2$ and $threads = 27$,$radix = 3$ respectively.

14.3.4 Observations and Interpretations

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

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

Time: 0.023

OK (2 tests)
```

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

14.4 StaticTreeBarrierTest

14.4.1 Particular Case

Here we want to implement a Barrier algorithm that does not suffer of either contention or excessive communication cost.

Just as a reminder, a barrier is a way of forcing asynchronous threads to act almost as if they were synchronous by adding synchronization points.

14.4.2 Solution

The solution for this problem is given by the Static Tree Barrier. The way it works is: each thread is assigned to a node in a tree. A thread at a node has to wait until all nodes bellow it in the tree have finished and then let its parent know that he is done. After that, it keeps spinning since the root of the tree signals a global sense bit change. So, one the root knows that all its children are done, this bit is flipped.

Here is the interesting code. Notice that the three is constructed within an array.

When a thread finishes its work, it calls the *await()* method below. Notice that it sits there till the sense is changed.

```
public void await() {
  boolean mySense = threadSense.get();
  while (childCount.get() > 0) {};  // spin until children done
  childCount.set(children);          // prepare for next round
  if (parent != null) { // not root?
    parent.childDone();              // indicate child subtree completion
    while (sense != mySense) {}; // wait for global sense to change
  } else {
    sense = !sense;    // am root: toggle global sense
  }
  threadSense.set(!mySense); // toggle sense
}
```

14.4.3 Experiment Description

Two test cases are provided:

- testBarrier. Creates a tree barrier of radix 2. 7 threads are created, meaning that we will have 3 levels in the tree. What is thread does is confirm that all siblings are executing at the same time.

- testBarrier3. Does the same bit using a tree barrier of radix 3. This means that we will have only 2 leves in the tree.

14.4.4 Sample Results

In our 24-cores machine, we got the following result:

```
[oraadm@gdlaa008 Barrier]$ junit barrier.StaticTreeBarrierTest
.Testing 13 threads, 1 rounds, radix 3
radix 3 done
.Testing 7 threads, 1 rounds, radix 2
radix 2 done

Time: 0.015

OK (2 tests)
```

However, some people in the group mentioned that they found a hang in this test which was fixed by adding the volatile qualifier to the global sense attribute.

So, in this exercise we learnt another way of implementing a barrier mechanism.

14.5 GTreeBarrierTest

14.5.1 Particular Case

The problem we are trying to solve is how to implement a Barrier that mitigates the problem of memory contention of the Sense-Reversing Barrier.

14.5.2 Solution

We present a Generic Tree Barrier, which actually uses the Sense Reversing Barrier, leading again to what the book called the Combining Tree Barrier. However, the implementation is done in such a way that each node's barrier can be changed to any other barrier. In fact, this exercise is a solution for problem 201 in the book.

As implied by the name, this Barrier is implemented in a tree, notice its similarity with the Simple Barrier Tree. The only difference is that we factorized the call to the node's barrier allowing it to actually use any type of barrier.

```
public void await(int me) {
  boolean sense = threadSense.get();
  barrier.await();
  if (me % radix == 0) {
    if (parent != null) { // root?
      parent.await(me / radix);
    }
    done = sense;
  } else {
    while (done != sense) {};
  }
  threadSense.set(!sense);
}
```

14.5.3 Experiment Description

The experiment consists of 8 threads and 8 rounds. The test will create a GTreeBarrier with radix 2. Each thread will verify that its siblings are performing the same round. This is all the work they have to do. If siblings happen to be in a different round, it means that they are not honoring the imposed barrier.

14.5.4 Sample Results and Interpretation

Here we present the result of the execution:

```
[oraadm@gdlaa008 Barrier]$ junit barrier.GTreeBarrierTest
.Testing 8 threads, 8 rounds

Time: 0.013

OK (1 test)
```

The implementation worked as expected since the test passed. So here we effectively have a solution for the proposed problem 201 in the book.

14.6 DisBarrierTest

14.6.1 Particular Case (problem)

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

14.6.2 Solution

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

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

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

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

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

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

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

14.6.3 Experiment Description

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

14.6.4 Observations and Interpretations

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

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

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

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

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

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

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

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

Chapter 15

Transactional Memory

15.1 ListTest (ofree)

15.1.1 Particular Case (problem)

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

15.1.2 Solution

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

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

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

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

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

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

15.1.3 Experiment Description

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

15.1.4 Observations and Interpretations

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

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

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

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

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

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

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

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

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

15.2 List (Lock-Based) Test

15.2.1 Particular Case

Here we are again experimenting with the STM implementation of Herlihy, which is called TinyTM. In this case, we are implementing a concurrent list data structure using STM.

However, this time we are using another approach. Here we are going to use Lock-Based Atomic objects.

15.2.2 Solution

One problem with the obstruction-free implementation that has been described in another experiment, is that writes allocate locators continuously. Also, reads have to go through two levels of indirection.

The Lock-Based approach is called so because it locks the object when in performs a write. When writing to the object, the *openWrite()* method will create a new tentative version and adds it to the so called write set.

Reading an object on the other hand is done optimistically and then it checks for conflicts. These conflicts are detected by using a version clock. The *openRead()* method will check whether the object is in the write set. If it is, then it creates a new tentative version. If not, it checks whether the object is locked. If so, the transaction aborts, otherwise the version is added to the read set.

As for the commit, the following steps are performed:

1. Lock the objects in the write set
2. Increment the global version clock in a thread-local stamp
3. The transaction checks that objects in the read set is not locked by another thread and that each object's timestamp is not greater than the transaction's read stamp. If this validation succeeds, then it commits.
4. Update the stamp field of each object in the write set.
5. Release any lock.

For using these Atomic objects in our list implementation, all we have to do is us the TNode class which internally uses the LockObject version of the AtomicObject interface.

15.2.3 Experiment Description

Three test cases are provided for this exercise:

- testParallel. 1024 elements are added initially to the list, then 32 threads are spawned. On even turns, the threads will insert a new element to the list. On odd turns, the thread removes an element from the list. The test checks that the values in the list at the end of the test are in order, that there are no duplicate values, that the number of commits matches the number of operations on the list, and that the size of the list is as expected.

- testSequential. 1024 elements are added initially to the list. Then a thread is spawned with the same behavior for even and odd turns described above. After 6 seconds, the thread is interrupted and the same check as above is performed.

- testIterator. It adds 100 consecutive elements to the list and then it checks that the list added these elements in the correct order and that the number of elements in the list is as expected.

15.2.4 Sample Results and Interpretation

Here is the result of the execution:

```
[oraadm@gdlaa008 ch18]$ junit TinyTM.list.locking.ListTest
.TestSequential
inserts: 12658, removes: 12142, missed: 24587, delta: 516
commits: 49387, aborts: 1
length = 1540 (expected 1540)
Commits: 49387 (expected 49387)
List OK
length = 1540
.TestParallel(32)
inserts: 12, removes: 0, missed: 10, delta: 12
commits: 22, aborts: 1521782
length = 1036 (expected 1036)
Commits: 22 (expected 22)
List OK
length = 1036
.
Time: 12.119

OK (3 tests)
```

This demonstrates that the implementation of the List by using TinyTM with Lock-Based Atomic objects works correctly

15.3 SkipList (Obstruction-Free) Test

15.3.1 Particular Case

In this exercise we are experimenting with the STM implementation mentioned in the book, which is called TinyTM.

The problem we are trying to solve is how to implement a concurrent Skip List by using STM with Obstruction-Free Atomic Objects.

15.3.2 Solution

Since the solution consists pretty much on using the TinyTM package and it has been described before (both in the Obstruction-Free and Lock-Based approach) we will not do it again here.

Please refer to previous exercises in this chapter for a better understanding of how TinyTM works.

15.3.3 Experiment Description

In this exercise, three test cases are provided:

- testParallel. 1024 elements are added initially to the list, then 32 threads are spawned. On even turns, the threads will insert a new element to the list. On odd turns, the thread removes an element from the list. The test checks that the values in the list at the end of the test are in order, that there are no duplicate values, that the number of commits matches the number of operations on the list, and that the size of the list is as expected.

- testSequential. 1024 elements are added initially to the list. Then a thread is spawned with the same behavior for even and odd turns described above. After 6 seconds, the thread is interrupted and the same check as above is performed.

- testIterator. It adds 100 consecutive elements to the list and then it checks that the list added these elements in the correct order and that the number of elements in the list is as expected.

15.3.4 Sample Results

The following is the output that we observe:

```
[oraadm@gdlaa008 ch18]$ junit TinyTM.skiplistset.ofree.SkipListSetTest
.TestSequential
TinyTM.exceptions.PanicException: java.lang.NullPointerException
  at TinyTM.TThread.doIt(TThread.java:51)
  at
TinyTM.skiplistset.ofree.SkipListSetTest$TestThread.run(SkipListSetTest.java:128)
Caused by: java.lang.NullPointerException
  at TinyTM.skiplistset.ofree.TSkipNode.getNext(TSkipNode.java:43)
  at TinyTM.skiplistset.ofree.SkipListSet.find(SkipListSet.java:108)
  at TinyTM.skiplistset.ofree.SkipListSet.add(SkipListSet.java:57)
  at
TinyTM.skiplistset.ofree.SkipListSetTest$TestThread$1.call(SkipListSetTest.java:130)
  at
TinyTM.skiplistset.ofree.SkipListSetTest$TestThread$1.call(SkipListSetTest.java:128)
  at TinyTM.TThread.doIt(TThread.java:41)
  ... 1 more
inserts: 0, removes: 0, missed: 0, delta: 0
commits: 0, aborts: 0
E.TestParallel(32)
TinyTM.exceptions.PanicException: java.lang.NullPointerException
  at TinyTM.TThread.doIt(TThread.java:51)
  at
TinyTM.skiplistset.ofree.SkipListSetTest$TestThread.run(SkipListSetTest.java:128)
Caused by: java.lang.NullPointerException
TinyTM.exceptions.PanicException: java.lang.NullPointerException
  at TinyTM.TThread.doIt(TThread.java:51)
  at
...
Caused by: java.lang.NullPointerException
TinyTM.exceptions.PanicException: java.lang.NullPointerException
  at TinyTM.TThread.doIt(TThread.java:51)
  at
TinyTM.skiplistset.ofree.SkipListSetTest$TestThread.run(SkipListSetTest.java:128)
Caused by: java.lang.NullPointerException
TinyTM.exceptions.PanicException: java.lang.NullPointerException
  at TinyTM.TThread.doIt(TThread.java:51)
  at
TinyTM.skiplistset.ofree.SkipListSetTest$TestThread.run(SkipListSetTest.java:128)
Caused by: java.lang.NullPointerException
inserts: 0, removes: 0, missed: 0, delta: 0
commits: 0, aborts: 0
E.
Time: 12.102
There were 2 errors:
1)
testSequential(TinyTM.skiplistset.ofree.SkipListSetTest)java.lang.NullPointerException
  at TinyTM.skiplistset.ofree.TSkipNode.getNext(TSkipNode.java:43)
  at TinyTM.skiplistset.ofree.SkipListSet$1.hasNext(SkipListSet.java:126)
  at
TinyTM.skiplistset.ofree.SkipListSetTest.sanityCheck(SkipListSetTest.java:181)
  at
```

```
TinyTM.skiplistset.ofree.SkipListSetTest.testSequential(SkipListSetTest.java:97)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
  at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
2)
testParallel(TinyTM.skiplistset.ofree.SkipListSetTest)java.lang.NullPointerException

FAILURES!!!
Tests run: 3,  Failures: 0,  Errors: 2
```

So, two test were complaining about NullPointerException. We started by adding the fix we put in the ListTest problem. With that, we got the following output:

```
[oraadm@gdlaa008 ch18]$ junit TinyTM.skiplistset.ofree.SkipListSetTest
.TestSequential
inserts: 220167, removes: 219660, missed: 438834, delta: 507
commits: 878661, aborts: 1
SkipListSet OK
length = 1531 (expected 1531)
Commits: 878661 (expected 878661)
.TestParallel(32)
inserts: 22, removes: 0, missed: 34, delta: 22
commits: 56, aborts: 66
SkipListSet OK
length = 1046 (expected 1046)
Commits: 56 (expected 56)
.
Time: 12.133

OK (3 tests)
```

So, that fixed the problem.

15.4 Lazy Skiplist

15.4.1 Particular Case

In this experiment we implement a Lazy Skiplist which uses an optimistic fine-grained locking, meaning that the method searches for its target node without locking, and acquires locks and validates only when it discovers the target.

15.4.2 Solution

Acording to the theory we use a skiplist to implement a set providing the methods, *add(x)* adds x to the set, *remove(x)* removes x from the set, and *contains(x)* returns true if, and only if x is in the set. Recall that a skiplist is a collection of linked lists. Each node in the list contains an item field (an element of the set), a key field (the items hash code), and a next field, which is an array of references to successor nodes in the list. Using *TinyTM*, threads communicate via shared atomic objects, which provide synchronization, ensuring that transactions cannot see one another's uncommitted effects, and recovery, undoing the effects of aborted transactions. Fields of atomic objects are not directly accessible. Instead, they are accessed indirectly through getter and setter methods. Accessing fields through getters and setters provides the ability to interpose transactional synchronization and recovery on each field access.

15.4.3 Experiment Description

The test creates 32 thread that parallel insert values, commits them and aborts then and compare the length and commits. If values are missing an assert fail is thrown.

15.4.4 Observations and Interpretations

The tests executed as expected and no errors where found.

```
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
    12.445 sec

————————— Standard Output ——————————
TestSequential
```

```
inserts: 186147, removes: 185635, missed: 370334, delta: 512
commits: 742116, aborts: 0
SkipListSet OK
length = 1536 (expected 1536)
Commits: 742116 (expected 742116)
TestParallel(32)
inserts: 24, removes: 0, missed: 41, delta: 24
commits: 65, aborts: 66
SkipListSet OK
length = 1048 (expected 1048)
Commits: 65 (expected 65)
------------------------------------------------------------
test-single:
BUILD SUCCESSFUL (total time: 12 seconds)
```

15.5 Restricted Bloom Filter

15.5.1 Particular Case

In this experiment we implement a bloom filters to reduce expensive disk lookups for non-existent keys. If the element is not in the bloom filter, then we know for sure we don't need to perform the expensive lookup. On the other hand, if it is in the bloom filter, we perform the lookup.

15.5.2 Solution

The basic bloom filter supports two operations, *contains()* and *add()*. Contains is used to check whether a given element is in the set or not.

```
public boolean contains(Object x) {
    return filter.get(hash0(x)) && filter.get(hash1(x));
  }
```

Add simply adds an element to the set. Removal is impossible without introducing false negatives.

```
  public void add(Object x) {
    filter.set(hash0(x));
    filter.set(hash1(x));
  }
```

15.5.3 Experiment Description

The test creates three values and adds two off them to the filter, then we check if the hash contains them and only fail if the assertion fails if a non-existing value returns true.

15.5.4 Observations and Interpretations

The tests executed as expected and no errors where found.

```
Testsuite: TinyTM.locking.BloomFilterTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.393 sec

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