Multiprocessor Programming Course

Dario Bahena Tapia - Ricardo Zavaleta Vazquez - Isai Barajas Cicourel

October 2015

Contents

Chapter 1

Mutual Exclusion

1.1 Peterson Algorithm

1.1.1 Particular Case

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

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

1.1.2 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.

1.1.3 Experiment Description

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

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

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

1. Processor: Intel Core i5 @2.5 GHz. 2 Cores.

2. L2 Cache per Core: 256 KB
3. L3 Cache: 3 MB
4. System Memory: 16 GB

1.1.4 Sample Results

Figure 1.2 shows the output that we observed in the netbeans window:

```
compile-test-single:
Created dir: /Users/ricardo/Desktop/ch02/Mutex/build/test/results
Testsuite: mutex.PetersonTest
parallel
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.108 sec

------------- Standard Output ---------------
parallel
------------- ---------------- ---------------
test-single:
BUILD SUCCESSFUL (total time: 5 seconds)
```

Figure 1.1: Output of the junit test

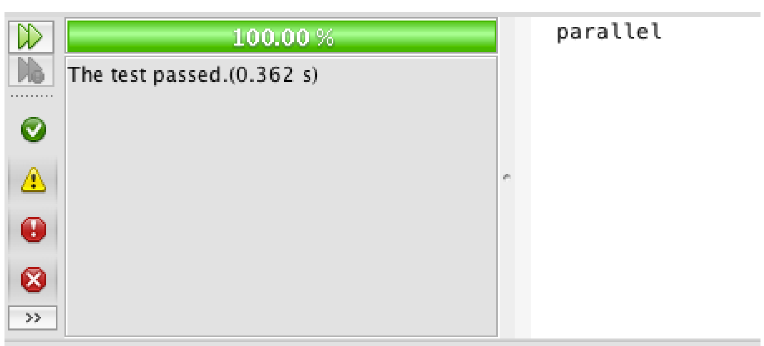

Figure 1.2: The Peterson test passed

1.1.5 Interpretation

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

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

1.2 LockFreeQueueTest

1.2.1 Particular Case (problem)

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

1.2.2 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;
  }
}
```

1.2.3 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).

1.2.4 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:

```
There was 1 error:
1) testParallelEnq(mutex.LockFreeQueueTest)java.lang.NullPointerException
        at mutex.LockFreeQueueTest.testParallelEnq(LockFreeQueueTest.java:69)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        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 20.
- T2: Executes method up to line 19.
- T1: Executes line 21, getting *items[0]*.
- T2: Executes line 20, but as *head* was changed already, it gets *items[1]*.
- 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:

```
.parallel both
.sequential push and pop
.parallel enq
E.parallel deq
Exception in thread "Thread-7" junit.framework.AssertionFailedError:
DeqThread: duplicate pop
        at junit.framework.Assert.fail(Assert.java:47)
        at mutex.LockFreeQueueTest$DeqThread.run(LockFreeQueueTest.java:132)
```

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 the test *testParallelBoth*. When produced several thread dumps of the Java program, we can see two hanging threads:

```
"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.

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

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

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

- T1: executes *tmp = tail.*
- T2: executes *tmp = tail.*
- T1: executes *tmp = tmp + 1.*
- T2: executes *tmp = tmp + 1.*
- T1: executes *tail = tmp.*
- T2: executes *tail = tmp.*

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

1.2.5 Proposed changes to fix the problems

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:

[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*.

```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).

1.3 Bakery Test

1.3.1 Particular Case

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

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

1.3.2 Solution

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

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

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

One thing that weougth 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.

1.3.3 Experiment Description

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

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

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

1.3.4 Sample Results

Figure 1.3 and 1.4 shows the output that we observed in the netbeans window:

```
compile-test-single:
Testsuite: mutex.BakeryTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.093 sec

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

Figure 1.3: Output of the junit test

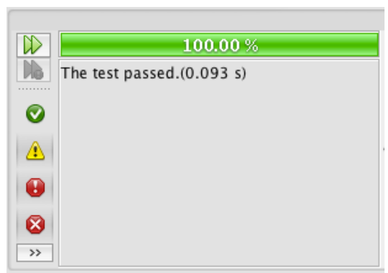

Figure 1.4: The Bakery test passed

1.3.5 Interpretation

The results in the proposed system were all OK. In all cases, the 8 threads were able to cooperate to increase the counter to 1024.

Chapter 2

Foundations of Shared Memory

2.1 Safe Boolean MRSW Register

2.1.1 Particular Case

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

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

2.1.2 Solution

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

2.1.3 Experiment Description

This program provides two test cases. The first one is a sequential test and the second one is a parallel test. The former simply calls a write and then a read from the same thread. This is not rocket science. The reader must retrieve what the writer wrote.

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

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

- Processor: Intel Core i5 @2.5 GHz. 2 Cores.

- L2 Cache per Core: 256 KB
- L3 Cache: 3 MB
- System Memory: 16 GB

2.1.4 Sample Results

We found out that the tests failed ocassionally. The manifestation of such failures were two:

1. The jUnit framework reports a failure in one of the tests as shown in figure 2.1
2. The jUnit doesn't report a failure, however the test output shows an *Out of Bounds exception*. This is shown in figure 2.2

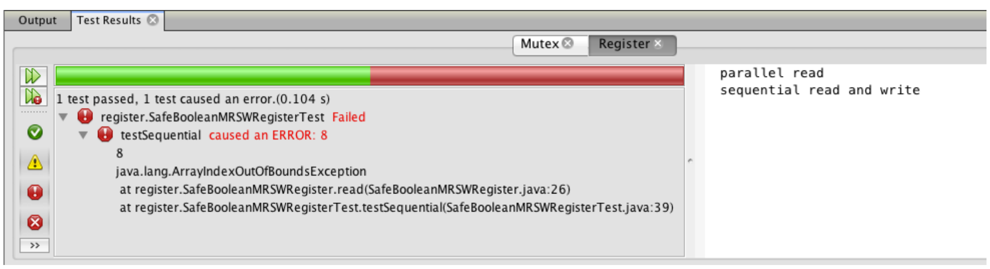

Figure 2.1: First type of failure

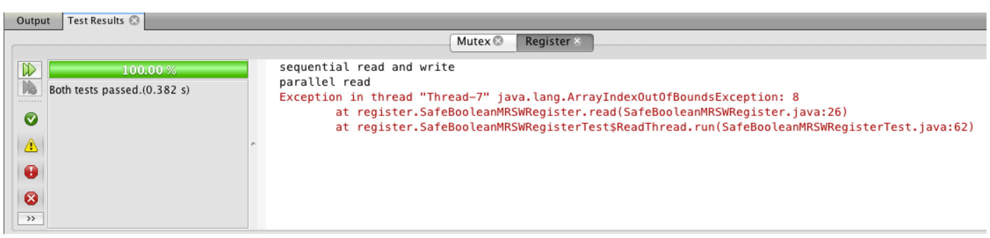

Figure 2.2: Second type of failure

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 (See figure 2.3).

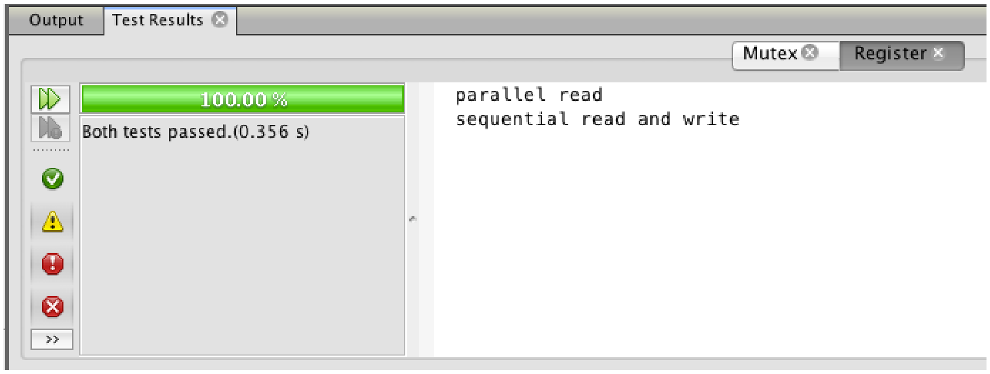

Figure 2.3: Output after fix

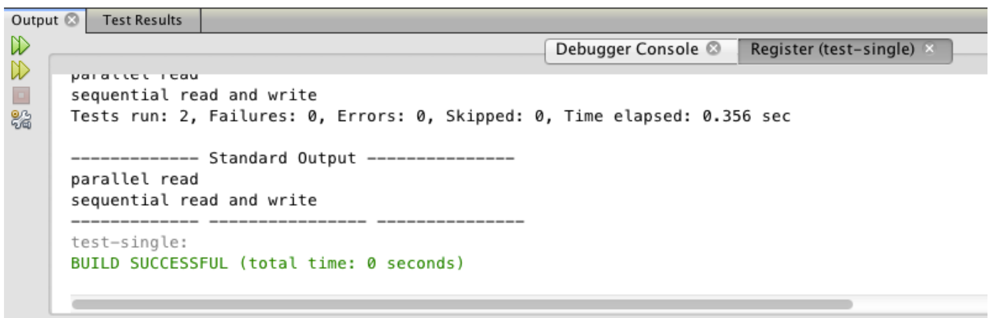

Figure 2.4: Output after fix of threadIds

2.1.5 Interpretation

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

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

2.2 Atomic MRSW Register

2.2.1 Particular Case

The characteristics of this type of register are:

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

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

2.2.2 Solution

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

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

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

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

2.2.3 Experiment Description

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

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

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

2.2.4 Sample Results

As in previous examples, we identified that the test fails from time to time. Figure 2.5 shows the output when the failure is seen.

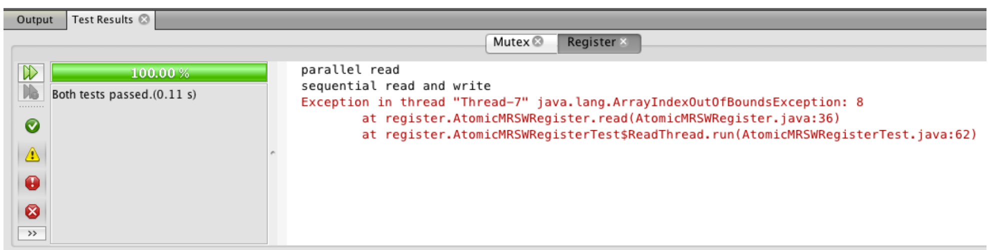

Figure 2.5: First type of failure

Again, the fix for these failures was to reset the thread ids.

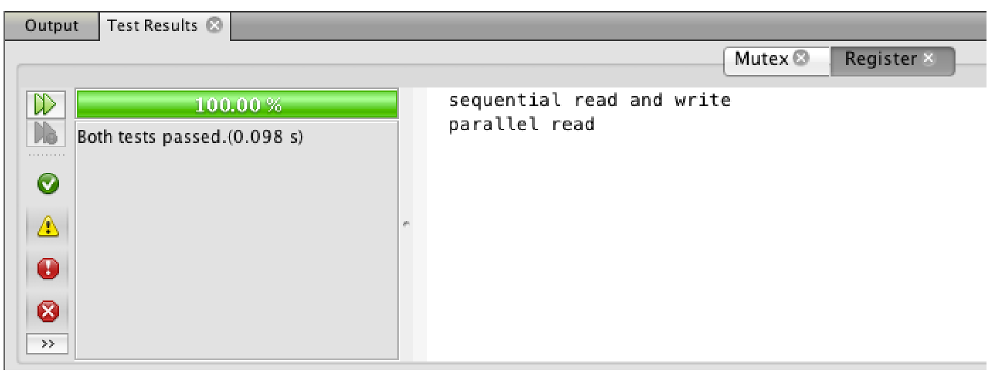

Figure 2.6: Output after fix

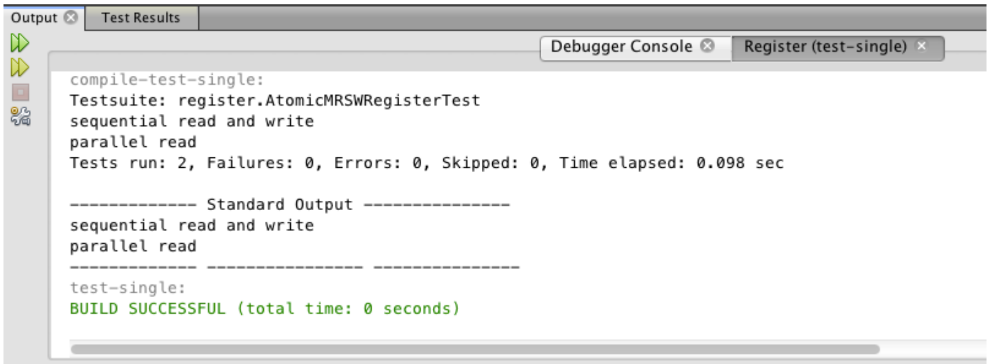

Figure 2.7: Output after fix of threadIds

2.2.5 Interpretation

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

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

2.3 Wait-Free Snapshots

2.3.1 Particular Case

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

2.3.2 Solution

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

One of the ideas behind the Wait-Free Snapshot is that 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.

2.3.3 Experiment Description

In this experiment, the test looks a bit different. Let us explain how it is different. 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 some times the first entry is that all entries are equal and we allow the first to sometimes be either greater or smaller than the second one.

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

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

2.3.4 Sample Results

For this test, we saw that in every try, both test cases passed.

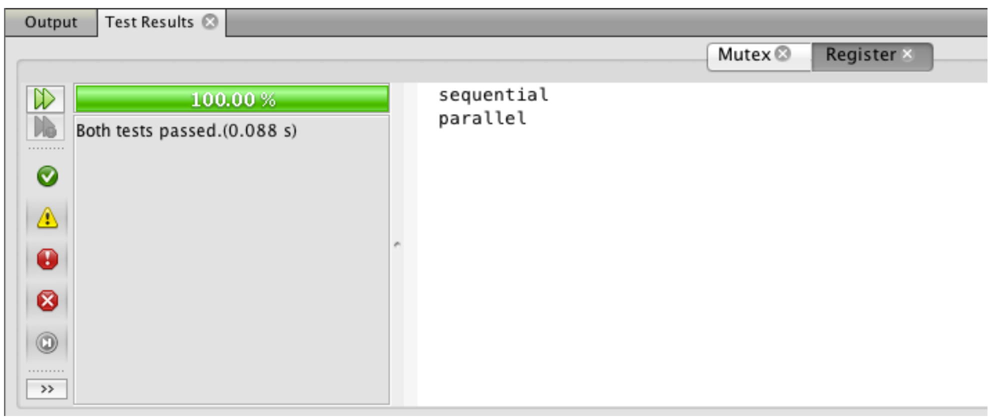

Figure 2.8: Successful execution of the tests for Wait-Free Snapshot

2.3.5 Interpretation

In this experiment we were able to observe how a Wait-Free Snapshot can be constructed.

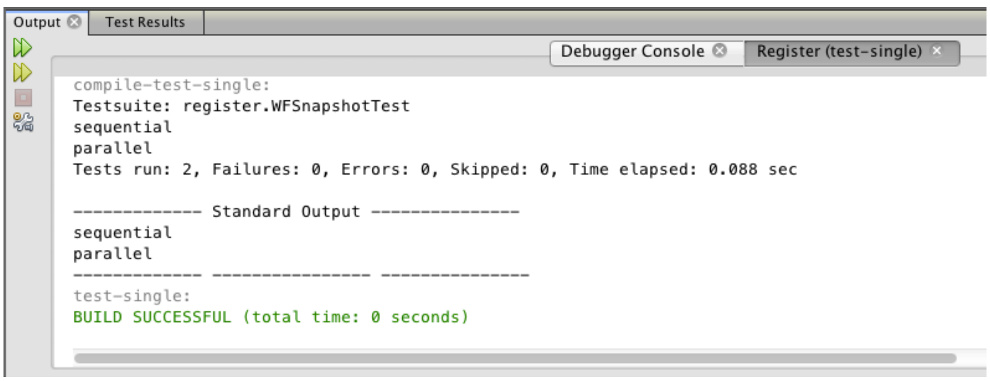

Figure 2.9: Successful execution of the tests for Wait-Free Snapshot

Chapter 3

Spin Locks and Contention

3.1 Problem Definition

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

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

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

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

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

3.2 Test-And-Set Locks

3.2.1 Particular Case

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

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

The second reason that has to do with the contradiction of sequential consistent memory is due to the hardware design which in most multiprocessor systems allow writes to be written to store buffers. This 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 a *memory barrier* or *memory fence*. These instructions are able to prevent reordering operations due to write buffering. The idea is that these instructions can be called in specific parts of the code to guarantee our proposed order of execution.

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

3.2.2 Solution

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

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

In the java code, the $testAndSet()$ is done with the $getAndSet()$ method. And setting the value to false is achieved with the method $set()$.

3.2.3 Experiment Description

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

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

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

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

3.2.4 Sample Results

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

3.2.5 Interpretation

It was shown that the algorithm seems to work and allows synchronization of two threads (consensus number was two) to access a shared resource.

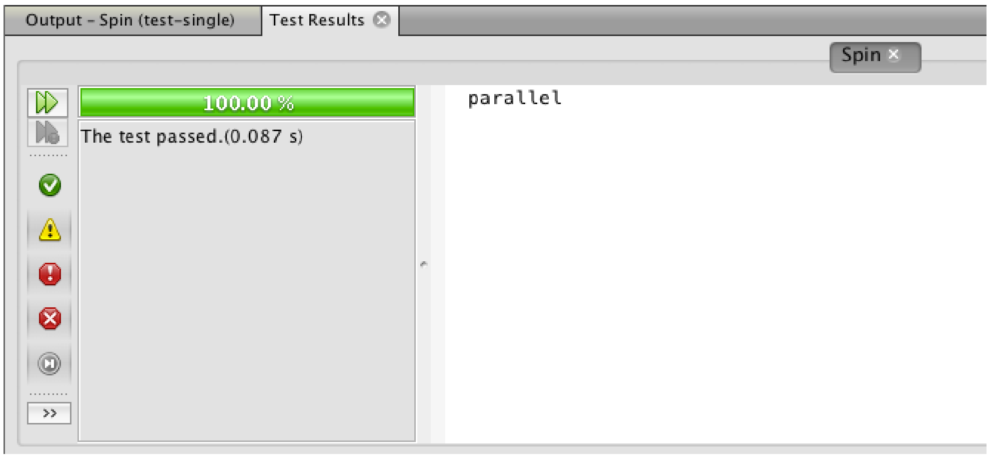

Figure 3.1: Successful execution of the TAS Lock test

```
compile-test-single:
Testsuite: spin.TASLockTest
parallel
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.087 sec

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

Figure 3.2: Successful execution of the tests for TAS Lock

3.3 Hierarchical Backoff Lock

3.3.1 Particular Case

The specific case we are trying to solve here is that of mutual exclusion with long waiting times on architectures that make use of hierarchical caching. Let us describe more deeply the characteristics of this scenario.

Firstly, it has already been discussed the difference between *spinning* and *backing-off*. The first case consists on doing an active wait in the case where a $lock()$ operation is not successful. On the other hand, a $backoff$ means that the thread who failed to acquire the lock refraing from retrying for some duration giving competing threads a chance to finish. As mentioned in the book, the idea is to avoid bus traffic caused by threads asking over and over again for the state of the lock.

The term *Hierarchical* comes from the fact that modern cache-coherent architectures organize processors in clusters. These clusters are organized in such a way that intra-cluster communication is faster than inter-cluster communication. So, the idea in this experiment is to find a lock implementation that takes advantage of these access times differences.

3.3.2 Solution

One of the solutions proposed in the book consists on adapting the *test-and-test-and-set* lock that was described in a previous chapter to exploit the characteristics of the cluster. The idea behind this implementation is that backoff times of local threads should be shorter than backoff times of remote threads.

3.3.3 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 32 threads
3. Each thread has to increment the counter by one 32 times
4. At the end, the counter must hold a value of $32 * 32$

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

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

3.3.4 Sample Results

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

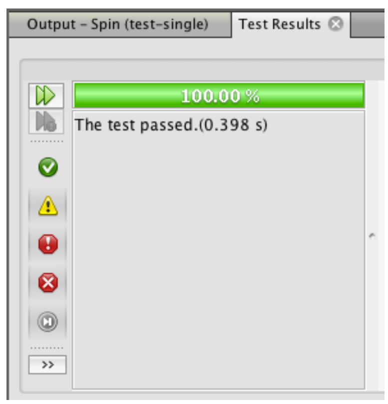

Figure 3.3: Successful execution of the tests for Hierarchichal Back-off test

```
compile-test-single:
Testsuite: spin.HBOLockTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.398 sec

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

Figure 3.4: Successful execution of the tests for Hierarchichal Back-off test

3.3.5 Interpretation

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

One problem however, has to do with the fact that this locking protocol is not fair in the sense that local threads have more chances of acquiring the lock in the next attempt.

Another thing that we ought to note is that threads spin in the same memory address which causes invalidation of remote cached copies of this value. This problem will be solved yet with another protocol.

3.4 CLH Queue Lock

3.4.1 Particular Case

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

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

3.4.2 Solution

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

3.4.3 Experiment Description

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

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

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

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

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

3.4.4 Sample Results

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

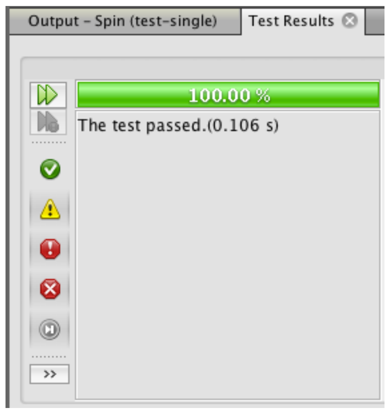

Figure 3.5: Successful execution of the tests for CLH Queue Lock test

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

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

Figure 3.6: Successful execution of the tests for CLH Queue Lock test

3.4.5 Interpretation

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

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

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

Chapter 4

Monitors and Blocking Synchronization

4.1 Count Down Latch

4.1.1 Particular Case

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

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

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

4.1.2 Solution

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

4.1.3 Experiment Description

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

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

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

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

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

4.1.4 Sample Results

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

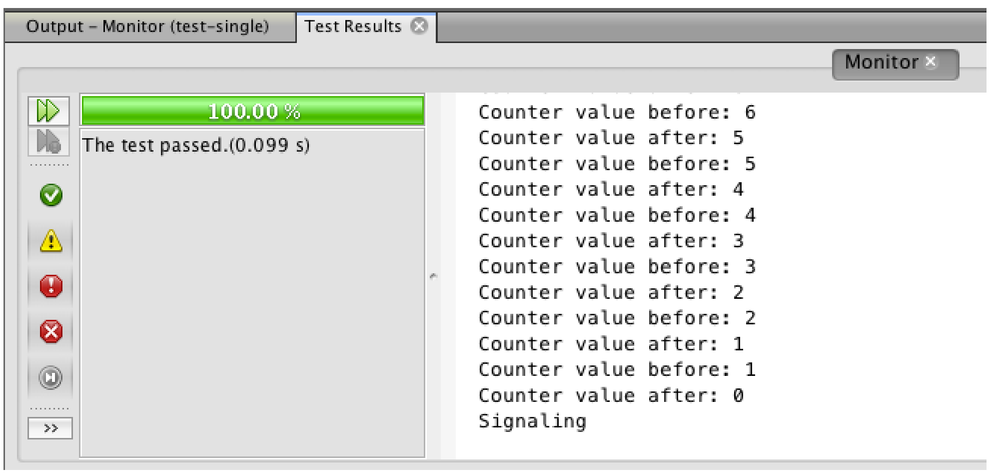

Figure 4.1: Successful execution of the tests for Count Down latch

4.1.5 Interpretation

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

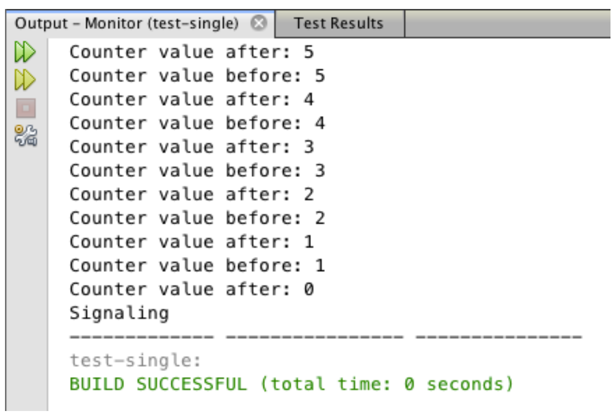

Figure 4.2: Successful execution of the tests for Count Down latch

Chapter 5

Linked Lists: The Role of Locking

5.1 LockFreeQueueTest

5.1.1 Particular Case (problem)

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

5.1.2 Solution

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

```
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;
  }
}
```

5.1.3 Experiment Description

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

- *testSequential*: calls *enq* method as many times as TEST_SIZE, and later calls *deq* method the same number of times checking that the FIFO order is preserved.
- *testParallelEnq*: enqueues in parallel (two threads) but dequeues sequentially.
- *testParallelDeq*: enqueues sequentially but dequeues in parallel (two threads)
- *testParallelBoth*: enqueues and dequeues in parallel (with a total of four threads, two for each operation).

5.1.4 Observations and Interpretations

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

Non atomic dequeue: referencing invalid registers

The symptom for this problem is a NullPointerException:

```
There was 1 error:
1) testParallelEnq(mutex.LockFreeQueueTest)java.lang.NullPointerException
        at mutex.LockFreeQueueTest.testParallelEnq(LockFreeQueueTest.java:69)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        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 20.
- T2: Executes method up to line 19.
- T1: Executes line 21, getting *items[0]*.
- T2: Executes line 20, but as *head* was changed already, it gets *items[1]*.
- 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:

```
.parallel both
.sequential push and pop
.parallel enq
E.parallel deq
Exception in thread "Thread-7" junit.framework.AssertionFailedError:
DeqThread: duplicate pop
        at junit.framework.Assert.fail(Assert.java:47)
        at mutex.LockFreeQueueTest$DeqThread.run(LockFreeQueueTest.java:132)
```

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 the test *testParallelBoth*. When produced several thread dumps of the Java program, we can see two hanging threads:

```
"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.

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

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

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

- T1: executes *tmp = tail.*
- T2: executes *tmp = tail.*
- T1: executes *tmp = tmp + 1.*
- T2: executes *tmp = tmp + 1.*
- T1: executes *tail = tmp.*
- T2: executes *tail = tmp.*

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

5.1.5 Proposed changes to fix the problems

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:

[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*.

```
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).

5.2 LazyListTest

5.2.1 Particular Case (problem)

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

5.2.2 Solution

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

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

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

The three methods ignore the locks while transversing the list, possibly passing over both logically and physically deleted nodes. The *add* and *remove* methods still lock *pred* and *curr* nodes as with *OptimisticList* solution, but the validation reduces to check that *curr* node has not been marked; as well as validating the same for *pred* node, and that it still points to *curr* (validating a couple of nodes is much better than transversing whole list though). Finally, the introduction of logical removals implies a new contract for detecting that an item still belongs to set: it does so, if still referred by an unmarked reachable node.

5.2.3 Experiment Description

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

5.2.4 Observations and Interpretations

The test works as expected, below sample execution:

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

Time: 0.03

OK (4 tests)
```

Chapter 6

Concurrent Queues and the ABA Problem

6.1 SynchronousDualQueueTest

6.1.1 Particular Case (problem)

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

6.1.2 Solution

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

This solution has some nice properties:

- Waiting threads can spin on a locally cached flag (scalability).
- Ensures fairness in a natural way. Reservations are queued in the order they arrive.
- This data structure is linearizable, since each partial method call can be ordered when it is fulfilled.

6.1.3 Experiment Description

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

6.1.4 Observations and Interpretations

The test runs normally most of the times, but in some occasions it hangs. 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. The easiest way to handle this is to modify the test to handle *null* values on dequeuers thread class:

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

With above modification, the test does not hang anymore.

6.2 LockFreeQueueRecycleTest

6.2.1 Particular Case (problem)

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

6.2.2 Solution

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

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

6.2.3 Experiment Description

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

6.2.4 Observations and Interpretations

The test does not exhibit any pitfall, which suggests that the theory works just fine on the tested machine. Sample output below:

```
.parallel enq
.parallel deq
```

```
.parallel both
.sequential push and pop

Time: 0.023

OK (4 tests)
```

6.3 BoundedQueueTest

6.3.1 Particular Case (problem)

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

6.3.2 Solution

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

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

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

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

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

6.3.3 Experiment Description

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

6.3.4 Observations and Interpretations

The original code for the test made no sense, as conditions and actions for *enq* and *deq* methods were inverted; we did not even care testing such strange version that did not match at all the one published on the book (we also needed to fix the initialization of the *size* variable, which shall be zero instead of *capacity*). After the corrections mentioned on the Solution section above, the test ran normally. Sample output below:

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

Time: 0.028

OK (4 tests)
```

Even if the original version of this test actually ran (we did not test it), it turns out quite counter-intuitive.

Chapter 7

Counting, Sorting and Distributed Coordination

7.1 TreeTest

7.1.1 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.

7.1.2 Solution

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

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

7.1.3 Experiment Description

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

7.1.4 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.At*

```
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();
```

```java
        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.