ORACLE®

Multiprocessor Programming Course

Dario Bahena Tapia - Ricardo Zavaleta Vazquez - Isai Barajas Cicourel

October 2015

1 LockFreeQueueTest

1.1 Particular Case (problem)

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

1.2 Solution

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

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

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

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

1.3 Experiment Description

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

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

1.4 Observations and Interpretations

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

1.4.1 Non atomic dequeue: referencing invalid registers

The symptom for this problem is a NullPointerException:

```
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.

1.4.2 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.

1.4.3 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.

1.5 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 class, so at any moment in time, only one synchronized method can actually run. In order to overcome this limitation, we can use synchronized blocks against two different lock objects (one for each operation).

[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.*

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 cite scenario that we try to dequeue with an empty queue, we would expect to simply retry again (giving the change to parallel enqueue threads to produce something for us to pop). The final code which incorporates these fixes is listed below:

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

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

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

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

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