Producer Consumer Thread Race Conditions

Background

Recently, on StackOverFlow.com, a question was asked about why java.util.concurrent.ArrayBlockingQueue use *while* loops instead of *if* around calls to await(). The accepted answer was that it was due to spurious wakeups. I strongly disagree with this answer and instead believe that it is due to bad programming around race conditions. This is an all too common problem with producer/consumer models from threaded/reentrant applications.

The Problem

In producer/comsumer applications you have threads which are producing some items and threads which are consuming the items and working on them. This can be a website with servlet threads queueing up aynchronous database requests, a thread reading lines from a file and queueing up work that needs to be done on each line, etc..

Typically you have some sort of blocking queue so that when a producer has a item, it adds it to the end of the queue, possible blocking if the queue has too many items in it already. The consumer, when it needs a new item, gets from the front of the queue, possibly blocking if the queue is empty. It is the fact that both sides are blocking and need to signal the other side is where the complexity and confusion originate.

In the code listed at the bottom of the page, the consumer locks the lock and tests whether the items list is empty. If it is empty then it waits until a producer signals the hasItems condition. However, when an item is added to the list, and the condition is signaled, the consumer goes to re-lock the lock and may end up *behind* another consumer waiting for the same lock. That other comsumer will get the lock first and de-queue the item that the first consumer was signaled for. *This* race to the dequeue is why what we call a race condition and it is the real source of the problem.

The Race

Here is a more specific enumeration of the race condition:

1. getter A is waiting for there to be items in the queue (line #31)
2. putter B locks the lock (line #54)
3. getter C finishes the last item, calls get(), tries to lock the lock, waits for B to unlock (line #25)
4. putter B adds an item into the queue and signals that there is an item there (line #72)
5. getter A is awoken and tries to lock the lock, has to wait for B to unlock (after line #31)
6. putter B unlocks (line #74)
7. getter C is ahead of getter A waiting for lock so it is given the lock *first* (line #25)
8. getter C dequeues the item that getter A was notified for (!!), and then unlocks (line #48)
9. getter A is finally able to lock, goes forward and calls remove on an empty list (line #40)

The race is between getter A and C on who is able to get the lock first to dequeue the new item. The race condition would be solved by changing lines 26 and 51 from `if` statements to `while` statements. Once getter A gets the lock it would check again to see if items is empty and would go back to waiting if C had already dequeued. The producer would also check again to see if the list is at capacity.

This is the real problem with many producer/comsumer models instead of spurious wakeups which I'm sure do happen but not nearly as frequently. As an interesting historical note, I had submit corrections to the original O'Reilly Pthreads book which suffered from the same problems -- errata had to be issued and a new version published.

Other Reading

- Producer-Consumer - using while loop instead of if

Sample Code

Click to download sample code.

```
1  package com.j256;
2
3  import java.util.Vector;
4  import java.util.concurrent.locks.Condition;
5  import java.util.concurrent.locks.ReentrantLock;
6
7  /**
8   * Little class which demonstrates the perils of using if statements instead of while
9   * in producer/consumer loops.
10  */
11 public class Foo {
12
13         // how many maximum items should the list store
```

```
14          private static final int CAPACITY = 10;
15          private static final int NUM_PUTTERS = 10;
16          private static final int NUM_GETTERS = 10;
17
18          private Vector<String> items = new Vector<String>(CAPACITY);
19          private ReentrantLock lock = new ReentrantLock();
20          private Condition hasSpace = lock.newCondition();
21          private Condition hasItems = lock.newCondition();
22
23          // consumer method
24          public String get() {
25                  lock.lock();
26                  try {
27                          // see if the list is empty, !!DANGER!! should be a while loop
28                          if (items.isEmpty()) {
29                                  try {
30                                          // wait for the list to have items
31                                          hasItems.await();
32                                  } catch (InterruptedException e) {
33                                          e.printStackTrace();
34                                          return null;
35                                  }
36                          }
37                          String item = null;
38                          try {
39                                  // remove the item from the front of the list
40                                  item = items.remove(0);
41                                  // signal a waiting producer (if any) that the list has space
42                                  hasSpace.signal();
43                          } catch (ArrayIndexOutOfBoundsException e) {
44                                  System.err.println("Under capacity: " + items.size());
45                          }
46                          return item;
47                  } finally {
48                          lock.unlock();
49                  }
50          }
51
52          // producer method
53          public void put(String item) {
54                  lock.lock();
55                  try {
56                          // see if the list is full, !!DANGER!! this should be a while loop
57                          if (items.size() >= CAPACITY) {
58                                  try {
59                                          // wait for the list to have space
60                                          hasSpace.await();
61                                  } catch (InterruptedException e) {
62                                          e.printStackTrace();
63                                  }
64                          }
65                          // add an item to the end of the list
66                          items.add(item);
67                          int size = items.size();
68                          if (size > CAPACITY) {
69                                  System.err.println("Over capacity: " + size);
70                          }
71                          // signal a waiting consumer (if any) that the list has an item
72                          hasItems.signal();
73                  } finally {
74                          lock.unlock();
75                  }
76          }
77
78          public static void main(String[] args) {
79                  new Foo().doMain(args);
80          }
81
82          private void doMain(String[] args) {
83                  for (int i = 0; i < NUM_GETTERS; i++) {
84                          new Thread(new Getter()).start();
85                  }
86                  for (int i = 0; i < NUM_PUTTERS; i++) {
87                          new Thread(new Putter()).start();
88                  }
89                  new Putter();
90          }
91
92          private class Getter implements Runnable {
```

```
 93                 public void run() {
 94                         while (true) {
 95                                 get();
 96                         }
 97                 }
 98         }
 99
100         private class Putter implements Runnable {
101                 public void run() {
102                         while (true) {
103                                 put("foo");
104                         }
105                 }
106         }
107 }
```

More topics.