Lock free many-producer/single-consumer patterns: A work queue of distinct events, order not important, follow-up question

devblogs.microsoft.com/oldnewthing/20180627-00

June 27, 2018

Raymond Chen

When I discussed the work queue of distinct events, order not important, commenter JDG noted that there's a sequence of events that can result in a spurious wake-up. "Is this spurious wake just a necessary evil to keep the algorithm simple? It looks harmless; the way the consumer function is written looks threadsafe to me."

Spurious wakes tend to come with the territory of lock-free algorithms. Sure, you can work really hard to remove all the tiny little race conditions that can lead to spurious wakes, but that usually makes your algorithm significantly more complicated without much real benefit.

Of course, one way to get rid of spurious wakes is to make your algorithm intentionally inefficient, but that's probably not what you're after either. In the linked example, you can get rid of spurious wakes by removing the loop from `ConsumeWork`. Instead of draining the pool of available work, the consumer drains only one item. You then make the `RequestWork` function always wake the consumer using something that has an internal counter, like a semaphore. Congratulations, you now have no spurious wakes. But you replaced it with a design that calls `WakeConsumer` far, far more often than the original code, so it's a net performance loss.

In the example I gave, the race window is open when the consumer has dequeued the last work item and is the process of retiring it. During that time, the next queued work item will generate a wake. I guess you could get into a persistent spurious wake case if the producer queues items at the same rate that the consumer is retiring them, so that the race window is open for most of the loop.

I guess you could fix this by having another flag that says "I'm busy retiring work items."

```
SLIST_HEADER WorkQueue;
LONG busy = 0;

void RequestWork(WorkItem* work)
{
 if (InterlockedPushEntrySList(&WorkQueue, work)
                                                 == nullptr) {
  if (!InterlockedCompareExchange(&busy, -1, -1)) {
   // You provide the WakeConsumer() function.
   WakeConsumer();
  }
 }
}

// You call this function when the consumer receives the
// signal raised by WakeConsumer().
void ConsumeWork()
{
 InterlockedExchange(&busy, 1);

 PSLIST_ENTRY entry;
 while ((entry = InterlockedPopEntrySList(&WorkQueue))
                                                 != nullptr) {
back_into_the_loop:
  ProcessWorkItem(static_cast<WorkItem*>(entry));
  delete entry;
 }

 InterlockedExchange(&busy, 0);

 // Final race condition: Maybe we were too aggressive
 // in suppressing spurious wakes.
 entry = InterlockedPopEntrySList(&WorkQueue);
 if (entry) {
  InterlockedExchange(&busy, 1);
  goto back_into_the_loop;
 }
}
```

Maybe there's a more elegant way of phrasing this, but I think it illustrates that the work to remove the last vestiges of spurious wakes can result in an algorithm that is harder to read.

Raymond Chen

Follow