Lock free many-producer/single-consumer patterns: A work queue of identical non-coalescable events

devblogs.microsoft.com/oldnewthing/20161123-00

November 23, 2016

Raymond Chen

Onward with our miniseries on lock-free many-producer/single-consumer patterns. Today, we're going to look at the case where you have a work queue where there can be multiple work items, and you need to perform them all, but each item is identical.

For example, you may have a *Buy It* button. Each time the user clicks the button, you want to run a transaction. But each button press is equivalent; all that's important is the number of times the user pushed the button.

Okay, that's not a very good example, but it'll have to do.

One way of doing this is with a semaphore, where the number of tokens in the semaphore is the number of work items left to be done. But let's stick with our current pattern where the producers need to manually wake the consumer, say with a message, and we want to minimize the number of times we need to perform the wake ritual.

```
LONG WorkCount;

void RequestWork()
{
 if (InterlockedIncrement(&WorkCount) == 1) {
  // You provide the WakeConsumer() function.
  WakeConsumer();
 }
}

// You call this function when the consumer receives the
// signal raised by WakeConsumer().
void ConsumeWork()
{
 while (InterlockedDecrementToZero(&WorkCount)) {
  DoSomeWork();
 }
}

bool InterlockedDecrementToZero(LONG volatile* value)
{
 LONG original, result;
 do {
  original = *value;
  if (original == 0) return false;
  result = original - 1;
 } while (InterlockedCompareExchange(value, result,
                                     original) != original);
 return true;
}
```

The `InterlockedDecrementToZero` function follows the pattern for building complex interlocked operations, in this case, decrementing a number, but not decrementing it below zero. We check if the value is zero; if so, then stop and return `false`. Otherwise, try to swap it with the value one less than the current value. If that fails, then it means that another thread changed the `WorkCount` while we were busy thinking, so we start over. If we successfully decremented, then return `true`.

The `WorkCount` variable remembers how much work there is for the consumer to do. When the first piece of work arrives, we wake the consumer, and the consumer keeps draining the work until it's all done.

Remember, there is only one consumer, so if `WakeConsumer` is called while `ConsumeWork` is still running, the wake will not start a new consumer immediately. It will wait for the existing `ConsumeWork` to complete before starting a new `ConsumeWork`.

Although this specific pattern may not be all that interesting, it can be viewed as a building block on top of which other patterns are built. We'll look at one such next time.

Exercise: Why couldn't the `InterlockedDecrementToZero` function have been written like this?

```
// Code in italics is wrong.
LONG InterlockedDecrementToZero(LONG volatile* value)
{
 LONG original = *value;
 if (original == 0) return false;
 InterlockedDecrement(value);
 return true;
}
```

Bonus chatter: We could have avoided having to write the `InterlockedDecrementTo-Zero` by writing this instead: void ConsumeWork() { LONG count = InterlockedExchange(&WorkCount); for (LONG i = 0; i < count; i++) { DoSomeWork(); } }

Discuss.

Raymond Chen

Follow