How do I avoid race conditions in WaitOnAddress where the wake happens before I enter the blocking state?

devblogs.microsoft.com/oldnewthing/20201214-00

December 14, 2020

Raymond Chen

A customer was struggling to close a race window with the `WaitOnAddress` function.

```
struct work_queue
{
  void produce(int value)
  {
    {
      std::lock_guard lock(m_mutex);
      m_work.push_back(value);
    }
    WakeByAddressSingle(&m_compare);
  }

  int consume()
  {
    while (true) {
      // If there is work, return it.
      {
        std::lock_guard lock(m_mutex);
        if (!m_work.empty()) {
          int value = m_work.front();
          m_work.pop_front();
          return value;
        }
      }

      // Drop the lock before waiting for work.
      WaitOnAddress(&m_compare, &m_undesirable, sizeof(m_compare), INFINITE);
    }
  }

  std::mutex m_mutex;
  std::dequeue m_work;
  int m_compare = 0;
  int m_undesireable = 0;
};
```

This is obviously a simplification but the idea is that when some work is produced (here represented by an integer), it is added to a deque, and the `m_compare` address is signaled.

To consume work, we see if the deque is nonempty, and if so, we remove the front element and return it. Otherwise, we use `WaitOnAddress` to wait for a signal.

The customer noted that this code generally worked, but sometimes the consumer would get stuck because of a race condition: After noticing that there is no work to do, the code goes to sleep. Just before the consumer goes to sleep, a producer adds a work item and wakes the address. This wake happens when nobody is waiting, so it has no effect. And then the consumer goes to sleep and never wakes up, even though there is work to be done.

The customer wanted to know how to fix this race condition. They don't want to hold the lock while waiting, because that would prevent new work from being produced.

Well, one obvious solution is to switch to a condition variable, which is well-suited to this pattern. But let's say that the customer wants to do it all with `WaitOnAddress`, say, because the example above was oversimplified, and in reality their work queue is more complicated.

The root cause of their problem is that they're holding the `WaitOnAddress` wrong.

Notice that the `m_compare` is never modified. It *always* holds the undesirable value. Therefore, the `WaitOnAddress` is within its rights to go to sleep and never wake up!

The customer was relying on an implementation detail: Waking by address wakes a waiter, even if the waiter's wake criteria aren't satisfied. This is legal behavior because `WaitOn-Address` is permitted to wake spuriously. The expectation is that upon waking, you go back and check your wake condition to confirm that it was a valid wake. (In this case, we check for a valid wake by inspecting the work queue.) The problem is that the customer was relying on the spurious wake.

The solution here is to use `WaitOnAddress` as it was intended. Let the `m_compare` represent the number of items in the queue, and the intent is to wait until the number is nonzero.

```
struct work_queue
{
  void produce(int value)
  {
    {
      std::lock_guard lock(m_mutex);
      m_work.push_back(value);
      ++m_compare;
    }
    WakeByAddressSingle(&m_compare);
  }

  int consume()
  {
    while (true) {
      // If there is work, return it.
      {
        std::lock_guard lock(m_mutex);
        if (!m_work.empty()) {
          int value = m_work.front();
          m_work.pop_front();
          --m_compare;
          return value;
        }
      }

      // Drop the lock before waiting for work.
      WaitOnAddress(&m_compare, &m_undesirable, sizeof(m_compare), INFINITE);
    }
  }

  std::mutex m_mutex;
  std::dequeue m_work;
  int m_compare = 0;
  int m_undesireable = 0;
};
```

Now we're actually waiting for a condition, namely for the `m_compare` to become nonzero. This solves the race condition, because if an item is queued just as the code is about to enter `WaitOnAddress`, the wait will not start, and the code will loop back to find the waiting item.

Bonus chatter: The `m_undesirable` doesn't need to be a member variable. It can be a local. And if you are clever, you may be able to arrange for a way to get rid of the `m_compare` variable: If your private deque data structure provide read-only access to a member variable that holds the number of elements in the queue, or at least something that contains a known value when the queue is empty, you can wait on that member variable directly. That's one of the bonus features of `WaitOnAddress`: It lets you create a synchronization object out of memory you're already using for some other purpose, thereby rendering it effectively free.

Bonus reading: Some time ago, I described how `WaitOnAddress` avoids race conditions when the value changes while it's getting ready to go to sleep, and how this can lead to spurious wakes.

Raymond Chen

Follow