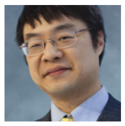
Implementing a synchronization barrier in terms of WaitOnAddress

devblogs.microsoft.com/oldnewthing/20160824-00

August 24, 2016



Raymond Chen

Last time, I promised to reimplement existing synchronization objects in terms of `WaitOn-Address`. Note that these are academic exercises rather than recommendations, because you should probably just use the existing synchronization objects instead of trying to roll your own. (Also, because the existing synchronization objects are optimized for their special cases.)

Today, let's implement the synchronization barrier. We won't implement the part that lets you customize how the synchronization barrier waits, though, since that's not really relevant to the exercise. (Allowing customization of the wait is left as the proverbial exercise for the reader.)

```
struct ALTBARRIER
{
  LONG TotalThreads;
  LONG WaitingThreads;
  LONG Unique;
};

void InitializeAltBarrier(
    ALTBARRIER* Barrier,
    LONG TotalThreads)
{
    Barrier->TotalThreads = TotalThreads;
    Barrier->WaitingThreads = 0;
    Barrier->Unique = 0;
}
```

The alternate synchronization barrier keeps track of these things:

- The total number of threads participating in the barrier. When this many threads enter the barrier, the barrier releases all the waiting threads.
- The number of threads waiting in the barrier.
- A unique number that changes for each barrier wait cycle. This is how waiting threads detect that they have been released.

```
BOOL EnterAltBarrier(
    ALTBARRIER* Barrier)
{
  LONG Unique = InterlockedCompareExchangeAcquire(
                         &Barrier->Unique, -1, -1);
  LONG WaitingThreads = InterlockedIncrement(&Barrier->WaitingThreads);

  if (WaitingThreads < Barrier->TotalThreads) {
    while (Barrier->Unique == Unique) {
       WaitOnAddress(&Barrier->Unique,
                     &Unique, sizeof(Unique), INFINITE);
    }
    return FALSE;
  } else if (WaitingThreads == Barrier->TotalThreads) {
    Barrier->WaitingThreads = 0;
    InterlockedIncrementRelease(&Barrier->Unique);
    WakeByAddressAll(&Barrier->Unique);
    return TRUE;
  } else {
    // The caller entered too many times.
    FatalError();
  }
}
```

Let’s look at how `EnterAltBarrier` works:

Right off the bat, it captures the current barrier wait cycle number. This is important, because the wait cycle number may change while the method is running, and we want to use the wait cycle number at the time the function was called. To ensure we don’t get a stale barrier wait cycle number, we use an interlocked function, and we use the trick of performing an `InterlockedCompareExchange` which tries to exchange a value with itself (which is a nop), and using an unlikely value to try to avoid generating a write.

We add ourselves to the count of waiting threads. If we are not the last thread, then we wait on the barrier’s wait cycle number until it changes to a value different from the wait cycle number at the time we entered the barrier. It’s important that we captured the wait cycle number *before* incrementing the waiting thread count, because our increment may have been enough to cause some other thread to decide that the barrier requirements have been met, and it might have incremented the wait cycle number while we were still in the process of getting ready to wait.

We keep waiting until the wait cycle number changes. This means that the previous wait cycle is complete. The rule for synchronization barriers is that exactly one thread returns `TRUE`; our barrier will have the thread that causes the waiting thread count to hit `TotalThreads` be the one to return `TRUE`. All the other threads return `FALSE`.

If the waiting through count reaches `TotalThreads`, then that means that the currently wait cycle is complete. We atomically increment the wait cycle number, and then use `WakeByAddressAll` to ask that everybody waiting for the wait cycle number to change be woken.

Using a 32-bit value for our wait cycle number means that a thread could be stuck in the scheduler for 4 billion barrier wait cycles, and when it finally gets a chance to run, it will still see that the wait cycle number has changed, because we don't recycle wait cycle numbers until 4 billion have passed.[1]

We use `InterlockedIncrementRelease` to ensure that the assignment of `WaitingThreads = 0` is visible to other processors before we increment the wait cycle number.

The last case is if the number of threads in the barrier exceeds the total number allowed. If this happens, then there is a bug in the caller, because the caller promised that it wouldn't send more than `TotalThreads` threads into the barrier at a time.

One interesting side effect of this alternate implementation is that it doesn't have the causality limitation of the real synchronization barrier. Once the barrier releases its threads, new threads can enter the barrier even before the old threads exit. This works because even if the old threads get stuck in the scheduler for a long time, when they finally get a chance to run, they will see that the wait cycle number has changed, and they will know that their wait is over.[2]

Next time, we'll demonstrate `WakeByAddressSingle`.

[1]If you are super-paranoid, then you could use a 64-bit counter for the wait cycle number. That gives you 18 pentillion barrier wait cycles. This means that even if you had a 4GHz processor entering the barrier every cycle, it would have to be locked out of the barrier for 136 years before the possibility that the wait cycle counter would get reused. Of course, if you have a thread waiting 136 years to be scheduled, then you have a bigger problem than a synchronization barrier getting stuck.

[2]You might say that the system implementation of synchronization barriers has just a single bit for the "cycle number". If a thread gets stuck in the scheduler for two cycles, then it will not realize that it's supposed to have woken.

Raymond Chen

Follow