Spurious wakes, race conditions, and bogus FIFO claims: A peek behind the curtain of WaitOnAddress

devblogs.microsoft.com/oldnewthing/20160826-00

August 26, 2016

Raymond Chen

We spent the past few days looking at the `WaitOnAddress` function. Today we're going to peek behind the curtain.

Reminder: These are undocumented implementation details and are subject to change at any time. The information is presented here for educational purposes. Do not take dependencies on the implementation.[1]

Okay, let's go.

The general idea behind the implementation of the `WaitOnAddress` function is that all of the threads that have an outstanding `WaitOnAddress` call are kept on a lock-free hash table, keyed by the address. Once the thread adds itself to the hash table, the thread calls an internal kernel function to go to sleep.

When one of the `WakeByAddress` functions is called, it searches the hash table looking for any threads that are waiting for the address, and either wakes one or all of them by removing the corresponding hash table entry and calling an internal kernel function to snap the thread out of its stupor.

From this rough description, you can see a few race conditions. For example, what if the monitored address contains the undesirable value at the entry to the `WaitOnAddress` function, while the thread was adding itself to the hash table, another thread changed the value and called `WakeByAddress`? Since the waking thread didn't see the hash table entry, it didn't see anybody to wake up. Meanwhile, the first thread finishes adding the entry to the hash table and goes to sleep, but it's too late. The thread that performed the wake operation didn't see anybody to wake, and the waiting thread waits forever.

To solve this problem, the `WaitOnAddress` function adds the thread to the hash table, and then re-checks whether the value at the address is equal to the undesirable value. If not, then the `WaitOnAddress` skips the wait entirely, thereby avoiding the problem of entering a wait that will never be woken.

Even with this fix, there's still a race condition: What if the re-check says that the value at the address is equal to the undesirable value, but just before the thread is about to call the internal kernel function, it gets pre-empted, and another thread issues the corresponding wake. The waiting thread is in the hash table, so the waking thread will ask the kernel to wake the waiting thread, but the waiting thread *isn't waiting yet*, so the wake request has no effect. The waiting thread then enters the kernel and goes to sleep, never to be woken.

To solve this second race condition, the kernel steps in to assist. When the waiting thread goes to sleep, it also gives the kernel the address that it's waiting for. Similarly, when the waking thread wants to wake up the waiting thread, it gives the kernel the identity of the thread to wake up as well as the address that is being woken. If the wake call arrives but the thread is not waiting on an address, then the kernel remembers the address for that thread. When the thread finally gets around to waiting on the address, the kernel says, "Oh, hey, there's a wake waiting for you."

This wake buffer is only one entry deep. If you try to wake a non-waiting thread twice, only the most recent wake request will be remembered. But that's okay, because the way that `WaitOnAddress` works, it never needs anything but the most recent wait request to be buffered.

This last race condition does explain one of the cases of a spurious wake: Suppose a thread calls `WaitOnAddress`, and after it adds the node to the hash table, another thread wakes the thread by address. Since the thread hasn't entered the kernel yet, the wake request is buffered. But now the thread re-checks the address and sees that the value is not the undesirable value, so the wait is skipped entirely, and control returns to the caller.

But the wake is still buffered.

That same thread then calls `WaitOnAddress` for the same address. This time, it adds the node to the hash table, no race condition occurs, and the thread enters the kernel to wait. The kernel thinks that this second wait is the race condition from the first call to `WaitOnAddress`, so it completes the wait immediately. "Hey, while you were busy setting up the wait, somebody already woke you."

Result: Spurious wake.

Another note about spurious wakes: There's nothing preventing an unrelated component from calling `WakeByAddress` on the same address. And as we noted earlier, there's also the possiblity that the value changed back to the undesirable value after the thread was woken. Therefore, spurious wakes are unavoidably just the way things are, and your code needs to be able to handle them. Even if there were a way to clear the buffered wake from the kernel, applications still have to deal with spurious wakes for other reasons.

My final note for today is a response to the documentation for `WakeByAddressSingle` which says

> If multiple threads are waiting for this address, the system wakes the first thread to wait.

This is one of those times where the documentation starts documenting the implementation rather than the contract. My guess is that the current implementation wakes waiters FIFO, and somehow that got into the documentation without a clear indication that this is a descriptive statement rather than a normative one.

What's more, it's not even accurate. The presence of spurious wakes means that the order in which code calls `WaitOnAddress` may not match the order in which they remain in the queue, because a spurious wake takes a thread out of the queue, and then it re-enters the queue at the end. (Sound familiar?) And who knows, a future version of `WaitOnAddress` may choose to dispense with FIFO waking in order to add convoy resistance.

Okay, that's enough discussion of `WaitOnAddress` for now.

[1] Mind you, this warning didn't stop people from snooping in on the internals of the `CRITICAL_SECTION` structure. As a result, the `CRITICAL_SECTION` structure must continue to use an automatic-reset event, use `-1` to indicate an unowned critical section. This prevented the kernel team from switching to a keyed event for critical sections. And even though the internal bookkeeping has changed, and the `LockCount` is no longer the lock count, the implementation must nevertheless go through contortions to ensure that the value of `LockCount` is `-1` precisely when the critical section is unowned.

Raymond Chen

Follow