WaitOnAddress lets you create a synchronization object out of any data variable, even a byte

devblogs.microsoft.com/oldnewthing/20160823-00

August 23, 2016

Raymond Chen

The `WaitOnAddress` function is a funny little guy. It lets you create a synchronization object out of 1, 2, 4, or 8 bytes of memory. And you even control the contents of the memory, so the synchronization object costs you nothing!

Basically, the `WaitOnAddress` function waits for the contents of a block of memory to change. You give it a memory buffer and an "undesirable value", and the function waits until the buffer's contents are something different from the undesirable value. (Of course, if the memory buffer already contains something different from the undesirable value, then the function returns immediately.)

You can accomplish the same thing with a spinloop, but `WaitOnAddress` puts the thread into a wait state so that it doesn't burn CPU while waiting.

Wow, this sounds amazing. But like they say, there's no such thing as a free lunch. For one thing, you have to give `WaitOnAddress` a little help when you actually change the value. After you update the value atomically, you need to call either `WakeByAddressSingle` if you want to wake up one waiting thread, or `WakeByAddressAll` if you want to wake up all waiting threads.

Note that everything must stay within a single process. You cannot wait on an address in one process and wake it in another. (Not that it would make much sense anyway, seeing as processes have separate address spaces.)

Another wrinkle is that the `WaitOnAddress` function might return spuriously; *i.e.*, it may return even if the address being watched still contains the undesirable value.

Those who are familiar with condition variables are well aware of the phenomenon of the spurious wake.[1] Spurious wakes are unavoidable because the thread may be woken due to the variable's value changing, but before the thread gets a chance to run, the value changes the back to the undesirable value. Therefore, when `WaitOnAddress` returns, you must check whether the value still has the undesirable value, and if so, you probably want to go back and wait again.

(The spurious wake also explains why you need to update the value atomically: If you use a non-atomic update, then another thread waiting on the address may experience a spurious wake at exactly the moment you are updating the value, and if you use a non-atomic update, that thread will see a torn value when it goes to check whether the wait was spurious.)

Here are some pros and cons of `WaitOnAddress` compared to, say, an event.

Pro: `WaitOnAddress` doesn't need to be initialized or destroyed. You don't have to worry about leaking memory because you forgot to destroy the synchronization object because there is nothing to destroy!

Pro: Like critical sections, `WaitOnAddress` does most of its work in user mode, so if you're lucky, you can avoid kernel transitions. (Though in practice, I don't think this buys you much, because most of the time, you probably expect `WaitOnAddress` to actually wait, which means that you're paying for a kernel transition anyway.)

Con: Since there is no synchronization object, you cannot ask the threadpool to notify you when a `WaitOnAddress` is ready. More generally, you cannot perform a simultaneous wait on an address and some other synchronization object. This means for example that you can't ask `MsgWaitForMultipleObjects` to wait for a message or on an address.

Con: As noted earlier, you cannot wait on an address in one process and wake it in another. If you need to synchronize between processes, then you cannot use `WaitOnAddress`.

Con: Internally, all of the addresses being waited on are kept in a hash table, which means that the theoretical complexity is linear in the number of pending `WaitOnAddress` calls, albeit with a low constant. This is probably not going to be a serious problem in practice because (1) it's a lock-free hash table, and (2) you won't have a huge number of pending `WaitOnAddress` calls, since it's bounded by the number of threads in your process.

Next time, by way of demonstration, we'll try implementing some existing synchronization concepts in terms of `WaitOnAddress`.

[1] Those who worked with Windows 95 device drivers may also observe a similarity between `WaitOnAddress` and Windows 95's `BlockOnID`, which had very similar semantics, including the phenomenon of spurious wakes.

Raymond Chen

Follow