PulseEvent is fundamentally flawed

devblogs.microsoft.com/oldnewthing/20050105-00

January 5, 2005

Raymond Chen

The `PulseEvent` function releases one thread (or all threads, if manual-reset) which is/are waiting for the pulsed event, then returns the event to the unset state. If no threads happen to be waiting, then the event goes to the unset state without anything happening.

And there's the flaw.

How do you know whether the thread that you think is waiting on the event really is? Surely you can't use something like

```
SignalSemaphore(hOtherSemaphore);
WaitForSingleObject(hEvent, INFINITE);
```

because there is a race between the signal and the wait. The thread that the semaphore is alerting might complete all its work and pulse the event before you get around to waiting for it.

You can try using the `SignalObjectAndWait` function, which combines the signal and wait into a single operation. But even then, you can't be sure that the thread is waiting for the event at the moment of the pulse.

While the thread is sitting waiting for the event, a device driver or part of the kernel itself might ask to borrow the thread to do some processing (by means of a "kernel-mode APC"). During that time, the thread is **not** in the wait state. (It's being used by the device driver.) If the `PulseEvent` happens while the thread is being "borrowed", then it will **not** be woken from the wait, because the `PulseEvent` function wakes only threads that were waiting **at the time the `PulseEvent` occurs**.

Not only are you (as a user-mode program) unable to prevent kernel mode from doing this to your thread, you cannot even detect that it has occurred.

(One place where you are likely to see this sort of thing happening is if you have the debugger attached to the process, since the debugger does things like suspend and resume threads, which result in kernel APCs.)

As a result, the `PulseEvent` function is useless and should be avoided. It continues to exist solely for backwards compatibility.

Sidebar: This whole business with kernel APCs also means that you cannot predict which thread will be woken when you signal a semaphore, an auto-reset event, or some other synchronization object that releases a single thread when signalled. If a thread is "borrowed" to service a kernel APC, then when it is returned to the wait list, it "goes back to the end of the line". Consequently, the order of objects waiting for a kernel object is unpredictable and cannot be relied upon.

Raymond Chen

Follow