If I have a thread waiting on an event, and I call SetEvent immediately followed by ResetEvent, is the waiting thread guaranteed to be released?

devblogs.microsoft.com/oldnewthing/20170407-00

April 7, 2017

Raymond Chen

A customer had developed a producer-consumer scenario and used a manual-reset event to coordinate the threads. “If there are *n* threads waiting on an event, is it guaranteed that all *n* threads will be unblocked if the event is signaled? Specifically, is this guaranteed if the event is reset very shortly after it is set? Hypothetically, all the waiting threads may not get scheduled before the signalling thread resets the event, but is it the case that once the event is signaled, all the waiting threads will be unblocked and will eventually start receiving CPU cycles?”

Actually, you have a problem even before you asked the question. How do you know that your waiting threads really are waiting on the event? After all, the fact that your program called `WaitForSingleObject` doesn’t guarantee that the thread is actually waiting. The thread might get pre-empted immediately after the `call` instruction and before the first line of code in `WaitForSingleObject` executes. As far as your program is concerned, it called `WaitForSingleObject`, but in reality, nothing meaningful has happened yet because `WaitForSingleObject` hasn’t gotten a chance to do anything. In this scenario, the signaling thread can call `SetEvent` and `ResetEvent` even before the waiting thread gets a chance to wait. And in that case, obviously, the thread won’t wake up because it never observed a set event.

Even if you somehow manage to guarantee that the threads are definitely waiting, you’re still out of luck. Setting the event and resetting it shortly afterward is basically reinventing `PulseEvent`, and we already saw that `PulseEvent` is fundamentally flawed. All the arguments for why `PulseEvent` is broken also apply to your homemade `PulseEvent` emulator: One of the waiting threads might be temporarily taken out of the wait state to process a kernel APC, and if your `SetEvent` and `ResetEvent` occur before the thread returns to the wait state, then the thread will have missed your simulated pulse.

If you have only one waiting thread, you can use an auto-reset event rather than a manual-reset event. That way, the event resets only when the waiting thread definitely observes the wait. But this won't work if you have multiple waiting threads.

You might consider using a semaphore and releasing *n* tokens to the semaphore when you want to wake up *n* threads. There's still a race condition, though: While preparing to wait, the thread increments *n* and then waits on the event handle. Suppose that the thread gets pre-empted after the increment and before the wait. The signaling thread releases *n* tokens. All but one of the tokens are consumed by the other waiting threads, leaving one token for the thread that is about to wait. But wait, what's that over there? Another thread swooped in, incremented *n* (from 0 to 1, presumably), and waited on the semaphore. That interloper thread *stole your token*!

Rather than trying to reimplement `PulseEvent` poorly, you probably would be better off using a condition variable. Condition variables are well-suited to these sorts of custom synchronization conditions.

Raymond Chen

Follow