A thread waiting on a synchronization object could be caught napping

devblogs.microsoft.com/oldnewthing/20060313-06

March 13, 2006

Raymond Chen

If you have a synchronization object, say a semaphore, and two threads waiting on the semaphore, and you then release two semaphore tokens with a single call to `ReleaseSempahore`, you would expect that each of the waiting threads would be woken, each obtaining one token. And in fact, that's what happens—most of the time. Recall in our discussion of why the `PulseEvent` function is fundamentally flawed that a thread in a wait state could be momentarily woken to service a kernel APC and therefore miss out on the pulse. The same thing might happen to the thread waiting for the semaphore. If the `ReleaseSemaphore` happens to occur while a thread has been taken out of the wait state to service a kernel APC, it will not claim the token immediately but rather will attempt to claim the token when the kernel APC completes and the thread is about to be returned to the wait state. Normally this is not a problem, because the token will still be there waiting for the thread. But if you have multiple threads all competing for the token, there is a small probability that in the time it took the thread to service the kernel APC, that other thread which was also waiting for a token not only got the first token, but managed to complete whatever work was associated with the token and issue a new `WaitForSingleObject` which claims the second token! The first thread was caught napping and missed out on both tokens. Fortunately, the cases where you have this sort of rampant competition for semaphore tokens are typically producer/consumer scenarios where it doesn't really matter who consumes the token, as long as somebody does.

Exercise: If there are multiple threads waiting on an auto-reset event and the event is signalled, can you predict which thread will be released?

Raymond Chen

Follow