Gotcha: A threadpool periodic timer will not wait for the previous tick to complete

devblogs.microsoft.com/oldnewthing/20200217-00

February 17, 2020

Raymond Chen

Some time ago, we learned that if your `WM_TIMER` handler takes longer than the timer period, your queue will not fill up with `WM_TIMER` messages.

However, that behavior does not extend to thread pool timers.

This is called out in the documentation for `CreateTimerQueueTimer`:

> The callback is called every time the period elapses, whether or not the previous callback has finished executing.

However, corresponding verbiage is missing from the documentation for `SetThreadpool-Timer` and `SetThreadpoolTimerEx`. But it applies there too.

This means that if your timer callback ends up taking time longer than the period, you may find multiple callbacks running at the same time, leading to confusion, because those multiple callbacks are probably trying to manipulate the same state, and they need to be careful not to confuse each other. If your callback holds a lock, you've starved out your main thread while all this is going on.

If the "long callback" situation is temporary, you may be able to catch up, but if the situation lasts a long time, you're in a Lucy in the chocolate factory situation, and things will spiral out of control.

I encountered this phenomenon back in the very early days of what today goes by the name of Windows Presentation Framework. I had a visual tree inside a scroller, and I observed that when the user clicked the down-arrow on the scrollbar, the contents scrolled by one line, and then froze for about ten seconds. During that time, the process spawned a dozen threads and pegged the CPU. Finally, the scroller repainted itself, having scrolled all the way to the bottom.

This is not how a scroller is supposed to behave.

What happened was that clicking the down-arrow did two things: The first thing was that it immediately scrolled the document by one line. That's what resulted in the contents scrolling by one line on the screen.

The second thing was that it set up an autorepeat timer to perform autoscrolling for as long as the user held the mouse button down. The autorepeat timer callback also scrolled the document by another line. The problem was that scrolling by a line took about a half a second, but the autorepeat timer was set to trigger every 100ms. In the time it took to scroll by one line, four additional requests to autoscroll were generated. The callbacks very quickly fell behind, and the thread pool tried to catch up by throwing more threads at the problem.

Now you had a dozen threads all trying to scroll the document by one line, and they're all competing with each other for CPU, so the calculations are even slower than normal. The UI thread is so busy with the document updates that it hasn't had a chance to notice that the user has released the mouse button, because input messages are relatively low priority.

Finally, the document scrolls all the way to the bottom, and the autoscroll callbacks start returning immediately with "Nope, no scrolling possible, nothing to do." This stops the candy conveyor belt, and the existing backlog of callbacks gradually gets retired, the UI thread finally notices that the mouse button was released, so it cancels the autorepeat timer, and the madness finally ends.

But what the user observed was that everything froze for ten seconds, and then they ended up at the bottom of the document.

If you want only one instance of the callback at a time, you can switch from a periodic timer to a one-time timer, and have your callback schedule the next timer callback when it finishes.

Raymond Chen

Follow