How to avoid accessing freed memory when canceling a thread pool callback

devblogs.microsoft.com/oldnewthing/20180502-00

May 2, 2018

Raymond Chen

The Windows thread pool is convenient, but one tricky part is how to remove items from the thread pool.

This discussion applies to all the thread pool objects, but I'll use thread pool timers for concreteness. You start by calling `CreateThreadpoolTimer` to establish the callback function and its context. Next, you call `SetThreadpoolTimer` to configure the timer: When the timer becomes due and its optional periodicity.

At this point, the timer is live. It will queue a callback (or callbacks, if periodic) to the thread pool according to the schedule you specified.

At some future point, you decide that you are finished with the timer. The timer may have elapsed by this point, or maybe you're cleaning up the timer before it elapsed.

Now you have a few options.

The simplest way is just to call `CloseThreadpoolTimer`. If the callback is not running, then this frees the timer immediately. Otherwise, it waits for the callback to complete before freeing the timer.

This "either/or" behavior makes `CloseThreadpoolTimer` basically useless for any callback with nontrivial context data, because you don't know when it's safe to free the context data. If you free it as soon as `CloseThreadpoolTimer` returns, then you might free it out from under an active callback.

That would be bad.

If you make the callback itself responsible for freeing the context data, you have the new problem of not knowing whether the callback is running, so the thread trying to close the timer doesn't know whether it should free the data or not. You can't have the callback set a flag saying, "Hey, I've started!" because there's still a race condition where the thread trying to close the timer checks the flag just before the callback manages to set it. You might try to fix this by making the context pointer be a pointer to a control block that in turn contains the data pointer, and having the callback and the main thread perform an atomic exchange on the data pointer, but you merely replaced the problem with an identical one: How do you know when it's safe to free the control block?

Fortunately, the documentation suggests an alternative:

- Call `SetThreadpoolTimer` to reconfigure the timer so it never comes due. This prevents new callbacks from occurring.
- Call `WaitForThreadpoolTimerCallbacks` to wait for any outstanding callbacks to complete.
- Call `CloseThreadpoolTimer`.
- Free the context data.

When `WaitForThreadpoolTimerCallbacks` returns, you know that there are no active callbacks, and your prior call to `SetThreadpoolTimer` makes sure that no new callbacks are scheduled. This means that you can call `CloseThreadpoolTimer`, and it will always be in the "callback is not running" case, so you can free the context data as soon as `CloseThreadpoolTimer` returns.

Great, we solved the context data lifetime problem, but we introduced a new problem: Deadlock.

Oh, look at the time. We'll continue this discussion next time.

Raymond Chen

Follow