Psychic debugging: Why your thread is spending all its time processing meaningless thread timers

devblogs.microsoft.com/oldnewthing/20081016-00

October 16, 2008

Raymond Chen

I was looking at one of those "my program is consuming 100% of the CPU and I don't know why" bugs, and upon closer investigation, the proximate reason the program was consuming 100% CPU was that one of the threads was being bombarded with `WM_TIMER` messages where the `MSG.hWnd` is `NULL`. The program was dispatching them as fast as it could, but the messages just kept on coming. Curiously, the `LPARAM` for these messages was zero.

This should be enough information for you to figure out what is going on.

First, you should refresh your memory as to what a null window handle in a `WM_TIMER` message means: These are thread timers, timers which are associated not with a window but with a thread. You create a thread timer by calling the `SetTimer` function and passing `NULL` as the window handle. Thread timer messages arrive in the message queue, and the `DispatchMessage` function calls the timer procedure specified by the message `LPARAM`. If the `LPARAM` of a thread timer message is zero, then dispatching the message consists merely of throwing it away. (If there were a window handle, then the message would be delivered to the window procedure, but there isn't one, so there's nothing else that can be done.)

The program was spending all its time retrieving `WM_TIMER` messages from its queue and throwing them away. The real question is how all these thread timers ended up on the thread when they don't do anything. Who would create a timer that didn't do anything? And who would create dozens of them?

One of the more common patterns for creating a window timer is to write `SetTimer(hwnd, idTimer, dwTimeout, NULL)`. This creates a window timer whose identifier is `idTimer`. Since the timer procedure is `NULL`, the `WM_TIMER` message is dispatched to the window procedure, which in turn will have a `case WM_TIMER` statement followed by a `switch (wParam)` to handle the timer message.

But what if `hwnd` is `NULL`, say because you forgot to check the return value of a function like `CreateWindow`? Well, then you just created a thread timer by mistake. And if you make this mistake several times in a row, you've just created several thread timers. Now you might

think that the code that created the thread timer by mistake will also destroy the thread timer by mistake when it finally gets around to calling `KillTimer(hwnd, idTimer)` and passes `NULL` for the `hwnd`. But it doesn't.

One reason is that in many cases, it's the timer that turns itself off. In other words, the `KillTimer` happens inside the `WM_TIMER` message handler. But if the `WM_TIMER` message isn't associated with that window, then that window procedure never gets a chance to turn off the timer.

Another reason is more insidious. Recall that the `idTimer` parameter to the `SetTimer` function is ignored when you create a thread timer. Since you can't predict what other thread timers may exist, you can't know which timer identifiers are in use and which are free. Instead, the `SetTimer` function creates a unique thread timer identifier and returns it, and it is that timer identifier you must use when destroying the thread timer. Of course, the code that accidentally created the thread timer thought it was creating a window timer (which uses the timer identifier you specify), so it didn't bother saving the return value. Result: Thread timer is created and becomes orphaned.

The machine I was asked to look at was running a stress scenario, so it was entirely likely that a low memory condition caused a function like `CreateWindow` to fail, and the program most likely neglected to check the return value. I never did hear back to find out if that indeed was the source of the problem, but seeing as they didn't come back for more help, I suspect I put them on the right track.

Raymond Chen

Follow