Running some UI code on a timer at a higher priority than your usual timer messages, or without coalescing

devblogs.microsoft.com/oldnewthing/20221221-00

December 21, 2022

Raymond Chen

A customer wanted something similar to a Windows `WM_TIMER` timer, but they didn't want the timer to be a low-priority message generated on demand, and they didn't want the messages to be coalesced. Is this possible?

You can do this by using some other kind of timer, but using window messages to synchronize with the UI thread. In other words, treat this as a work queue where the work is generated by a non-UI timer.

A naïve solution would be to post a custom message each time the timer elapses:

```
HWND g_hwnd;
PTP_TIMER g_timer = nullptr;

void CALLBACK TimerCallback(PTP_CALLBACK_INSTANCE instance,
    void* context, PTP_TIMER timer)
{
    PostMessage(g_hwnd, WM_DOSOMETHING, 0, 0);
}

// Must be called from UI thread
void StartTimer(
    FILETIME dueTime, DWORD period, DWORD window)
{
    g_timer = CreateThreadpoolTimer(TimerCallback,
        nullptr, nullptr);
    SetThreadpoolTimer(
        g_timer, &dueTime, period, window);
}

// Must be called from UI thread
void StopTimer()
{
    CloseThreadpoolTimer(g_timer);
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT message,
    WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    ...
    case WM_DOSOMETHING:
        DoSomething();
        break;
    ...
    }
    return DefWindowProc(hwnd, message, wParam, lParam);
}
```

It's very simple: Each time the timer fires, we post a "do something" message to the window.

The problem with this is that it can flood the message queue if the UI thread gets blocked for a long time for some reason. The timer keeps running, and new `WM_DOSOMETHING` messages are posted into the queue, but the UI thread has stopped processing messages, so the queued messages just accumulate, and you risk overflowing the message queue.

Better is to use an edge-triggered message, using a technique we saw some time ago.

```
HWND g_hwnd;
PTP_TIMER g_timer = nullptr;

// alternate: LONG g_count;
std::atomic<int> g_count;

void CALLBACK TimerCallback(PTP_CALLBACK_INSTANCE instance,
    void* context, PTP_TIMER timer)
{
    // alternate: InterlockedIncrement(&g_count)
    if (g_count.fetch_add(1,
            std::memory_order_relaxed) == 1) {
        PostMessage(g_hwnd, WM_DOSOMETHING, 0, 0);
    }
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT message,
    WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    ...

    case WM_DOSOMETHING:
        {
            // alternate: InterlockedExchange(&g_count, 0)
            int count = g_count.exchange(0,
                std::memory_order_relaxed);
            for (int i = 0; i < count; i++) DoSomething();
        }
        break;
    ...
    }
    return DefWindowProc(hwnd, message, wParam, lParam);
}
```

This time, when the timer expires, we just increment the number of outstanding operations. If the number incremented from zero to one, then we post a message to get the UI thread to drain the work.

When the UI thread receives the message, it exchanges the counter back to zero and then operates on each of the operations that were outstanding. In this simple example, we just do the "something" that many times. In a real program, you would probably write a special version `DoSomethingN(count)` which is more efficient than doing `DoSomething()` *N* times.

Note that I've been glossing over the problem of what to do if `StopTimer()` is called while there are still pending operations. As-written, what happens is that the pending operations will still be performed later. If you want them to be flushed or recalled, you have a little more work to do. I'll leave you work out the details of threadpool timers, but the idea is

- Stop the timer and wait for callbacks to complete.
- Exchange the `g_count` back to zero.
- If flushing: Perform `DoSomething()` that many times immediately.

Raymond Chen

Follow