What happens to lost timer messages if I don’t process them fast enough?

devblogs.microsoft.com/oldnewthing/20150928-00

September 28, 2015

Raymond Chen

Some time ago, I noted that if your `WM_TIMER` handler takes longer than the timer period, your queue will not fill up with `WM_TIMER` messages. The `WM_TIMER` message is generated on demand, and if your handler doesn’t check for messages, then there is no demand for timer messages. But what happens when your thread finally starts processing messages again? What happens to the timers that elapsed while you were busy? Do they accumulate?

Here’s a sketch of how timers work. (Note that the timers under discussion here are the timers set by the `SetTimer` function.)

When a timer is created, it is initially *not ready*.

Every *n* milliseconds (where *n* is the period of the timer), the timer is marked *ready*. This is done regardless of the state of the UI thread. Note that *ready* is a flag, not a counter. If the timer is already ready, then it stays ready; there is no such thing as “double ready”. The `QS_TIMER` flag is set on the queue state, indicating that there is now a pending timer for the thread. This in turn may cause a function like `GetMessage` or `MsgWaitForMultiple-Objects` to wake up.

When the appropriate conditions are met (as discussed earlier), the window manager checks whether there are any timers for the thread that are marked *ready*. If so, then the corresponding `WM_TIMER` message is generated and the *ready* flag is cleared.

Let’s illustrate this with our scratch program. Make the following changes:

```
#include <strsafe.h>

DWORD g_tmStart;

void SquirtTime()
{
 TCHAR sz[256];
 StringCchPrintf(sz, 256, "%d\r\n", GetTickCount() - g_tmStart);
 OutputDebugString(sz);
}
```

This adds a function that prints the number of milliseconds which have elapsed since `g_tmStart`. Note that we use simple subtraction and rely on unsigned arithmetic to handle timer rollover issues.

```
void CALLBACK OnTimer(HWND hwnd, UINT, UINT_PTR, DWORD)
{
 SquirtTime();
}
```

Our timer tick handler merely prints the elapsed time to the debugger.

```
BOOL
OnCreate(HWND hwnd, LPCREATESTRUCT lpcs)
{
 g_tmStart = GetTickCount();
 SetTimer(hwnd, 1, 500, OnTimer);
 Sleep(1750);
 return TRUE;
}
```

Finally, we create a 500ms timer on our window, but we also intentionally stall the thread for 1750ms.

Can you predict the output of this program?

Here’s what I got when I ran the program:

```
1797
2000
2500
3000
3500
4000
4500
...
```

Let’s see if we can explain it.

Since the timer is set to fire at 500ms intervals, every 500ms, the timer gets marked ready.

- At T = 500ms, the timer is marked ready.
- At T = 1000ms, the timer is marked ready. This is redundant, since the timer is already ready.
- At T = 1500ms, the timer is marked ready. Again, this is redundant.
- At T = 1750ms, the program finally wakes up from its `Sleep` and eventually gets around to processing messages.
- Hey look, there is a ready timer, so we generate a `WM_TIMER` message and clear the *ready* flag.
- At T = 1797ms, the timer message is processed.
- The program returns to its message loop, where there are no further messages to process, so it sits and waits.
- At T = 2000ms, the timer is marked ready. This causes the `GetMessage` to wake up generate a `WM_TIMER` message and clear the *ready* flag.
- At T = 2000ms, the timer message is processed.
- At T = 2500ms, the timer is marked ready. This causes the `GetMessage` to wake up generate a `WM_TIMER` message and clear the *ready* flag.
- At T = 2500ms, the timer message is processed.
- And so on, with a new timer message every 500ms that is processed immediately.

Observe that when the program begins processing messages at T = 1750ms, it receives only one timer message right away, even though three timer periods have elapsed.

I guess another way of looking at this is to think of your timer as setting a frame rate. If your thread is busy when it's time to render the next frame, then the next frame is late. And if your thread is really busy, it may drop frames entirely.

Raymond Chen

Follow