Swamping the thread pool with work faster than it can drain

devblogs.microsoft.com/oldnewthing/20110603-00

June 3, 2011

Raymond Chen

This scenarios is real, but details have been changed to protect the guilty.

Consider a window showing the top of a multi-page document. The developers found that when the user clicks the down-arrow button on the scroll bar, the program locks up for 45 seconds, over a dozen threads are created, and then suddenly everything clears up and the window displays the final paragraph of the document (i.e., it scrolled all the way to the bottom).

The problem was traced to queueing tasks to the thread pool faster than they can drain.

The document is an object which, unlike a window, has no thread affinity. (Naturally, there are critical sections around the various document methods so you don't have corruption if two threads try to modify the document at the same time.) The way to display a different portion of the document is to call a method which changes the viewport location.

When the user clicks the down-arrow button on the scroll bar, the main thread creates a periodic timer at four fifths of the double-click speed, and each time the timer elapses, it does the equivalent of `document.ScrollDown()`. The code cancels the timer once the user releases the mouse button.

The catch is that the document was so complicated that it takes a long time to change the view top and redraw the new view. (I don't remember exactly how long, but let's say it was 700ms. The important thing is that it takes longer than 400ms.)

Given that set-up, you can see what happens when the user clicks the scroll down-arrow. The initial scroll is initiated, and before it can complete, another scroll is queued to the thread pool. The document view keeps trying to update its position, but the periodic timer generates scroll requests faster than the document view can keep up.

If that description was too terse, here's a longer version.

The code for scrolling went something like this:

```
OnBeginScrollDown()
{
 // Start a timer to do the scrolling
 CreateTimerQueueTimer(&htimer, NULL, ScrollAgain, NULL,
     0, GetDoubleClickTime() * 4 / 5, WT_EXECUTEDEFAULT);
}
OnEndScrollDown()
{
 if (htimer != NULL) {
  DeleteTimerQueueTimer(NULL, htimer, INVALID_HANDLE_VALUE);
  htimer = NULL;
 }
}
ScrollAgain(...)
{
 document.ScrollDown();
}
```

(In reality, the program didn't use the `CreateTimerQueueTimer` function—it had a custom timer queue and a custom thread pool—but the effect is the same.)

At time *T* = 0, the user clicks on the scroll bar down-arrow. The UI thread starts the timer with an initial delay of zero and a period of 400ms. The timer fires immediately, and a thread pool thread is asked to run `ScrollAgain`. The `ScrollAgain` function calls `ScrollDown`, which begins the process of scrolling the document.

At time *T* = 400ms, the periodic timer fires, and a new thread pool thread is created to service it. Pool thread 2 calls `ScrollDown()` and blocks.

At time *T* = 700ms, the `ScrollDown` call on pool thread 1 completes, and now pool thread 2 can begin its call to `ScrollDown()`.

At time *T* = 800ms, the periodic timer fires again, and pool thread 1 (now idle) is asked to handle it. Pool thread 1 calls `ScrollDown()` and blocks.

At time *T* = 1200ms, the periodic timer fires yet again. This time, there are no idle threads in the thread pool, so the thread pool manager creates yet another thread to service the timer. Pool thread 3 calls `ScrollDown()` and blocks.

At time *T* = 1400ms, the `ScrollDown()` call issued by pool thread 2 completes. Pool thread 2 now returns to idle. Now the call to `ScrollDown()` from pool thread 1 (issued at time *T* = 800ms) can start.

At time *T* = 1600ms, the periodic timer fires *again*, and pool thread 2 is chosen to service it. Pool thread 2 calls `ScrollDown()` and blocks.

At time *T* = 2000ms, the periodic timer fires again, and a new pool thread is created to service it. Pool thread 4 calls `ScrollDown()` and blocks.

You can see where this is going, I hope. Work is being generated by the periodic timer at a rate of one work item per 400ms, but it takes 700ms to carry out each work item, and the tasks are serialized on the document. It's like Lucy in the chocolate factory. The document is frantically trying to carry out all the work, and it never manages to catch up. Eventually, the document scrolls all the way to the bottom, and the mass of pent-up calls to `ScrollDown()` all return immediately since there is no more scrolling possible.

Now that the document is idle, it can paint, and that's where the user finally sees the document, scrolled all the way to the bottom.

There are a number of possible solutions here.

One way is not to queue up another scroll while an old one is still running. Instead, just wait for it to finish, and then issue a new scroll that accumulates all the scrolling that had taken place while you were waiting for the first to complete. This results in jerky scrolling, however, and it creates a lag of up to 700ms between the user releasing the mouse button and scrolling actually stopping.

Another approach is to disable repainting the entire document when you detect that you are in the *document is too complex to scroll quickly* case and just scroll the scrollbar thumb. When the user stops scrolling, re-enable painting and *boom* the document appears at the user's chosen location. This preserves responsiveness, but you lose the ability to see the document as you scroll it.

I don't know what solution the customer finally went with. I was just there to help with the debugging.

Bonus example: Larry Osterman describes another situation with the same underlying cause.

Hidden take-away: Observe that both of these examples illustrate one of the subtle consequences of a design which moves all processing off the UI thread.

Update: Note that `SetTimer` wouldn't have helped here.

```
case WM_TIMER:
  if (wParam == SCROLLTIMER) {
    QueueUserWorkItem(ScrollAgain, NULL, WT_EXECUTEDEFAULT);
  }
  ...
```

Since the processing has been moved off the UI thread, the `WM_TIMER` messages are free to keep flowing in and queue up work faster than the background thread can keep up.

Raymond Chen

Follow