A pathological program which ignores the keyboard, and understanding the resulting behavior based on what we know about the synchronous input

devblogs.microsoft.com/oldnewthing/20130606-00

June 6, 2013

Raymond Chen

Today, we'll illustrate the consequences of the way the window manager synchronizes input when two or more threads decide to share an input queue.

Since I need to keep separate state for the two windows, I'm going to start with the new scratch program and make the following changes:

```
#include <strsafe.h>


class RootWindow : public Window
{
public:
 virtual LPCTSTR ClassName() { return TEXT(“Scratch”); }
 static RootWindow *Create();


 void AppendText(LPCTSTR psz)
 {
  ListBox_SetCurSel(m_hwndChild,
                    ListBox_AddString(m_hwndChild, psz));
 }


 void LogMessage(const MSG *pmsg)
 {
   TCHAR szMsg[80];
   StringCchPrintf(szMsg, 80, TEXT(“%d\t%04x\t%p\t%p”),
                   pmsg->time,
                   pmsg->message,
                   pmsg->wParam,
                   pmsg->lParam);
   AppendText(szMsg);
}


protected:
 LRESULT HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam);
 LRESULT OnCreate();
private:
 HWND m_hwndChild;
};


LRESULT RootWindow::OnCreate()
{
 m_hwndChild = CreateWindow(
      TEXT(“listbox”), NULL,
      LBS_HASSTRINGS | LBS_USETABSTOPS |
      WS_CHILD | WS_VISIBLE | WS_TABSTOP | WS_VSCROLL,
      0, 0, 0,0, GetHWND(), (HMENU)1, g_hinst, 0);


 return 0;
}
```

All we did above was add a list box to the window and provide public methods `AppendText` to add a string to the list box and `LogMessage` that adds a string based on the contents of a `MSG` structure. We're going to use this list box to log what the program is doing.

```
bool ShouldLogMessage(UINT uMsg)
{
 if (uMsg >= WM_KEYFIRST && uMsg <= WM_KEYLAST) return true;
 if (uMsg >= WM_MOUSEFIRST && uMsg <= WM_MOUSELAST) return true;
 return false;
}
```

This helper function above tells us which messages we want to log. For now, let's log keyboard and mouse messages.

Now, in order to demonstrate input thread attachment, we need two threads. Here comes the second thread:

```
DWORD CALLBACK AttachedThreadProc(void *lpParameter)
{
 RootWindow *prw = RootWindow::Create();
 SetWindowText(prw->GetHWND(), TEXT(“Bad window”));
 AttachThreadInput(PtrToInt(lpParameter),
                   GetCurrentThreadId(), TRUE);
 ShowWindow(prw->GetHWND(), SW_SHOW);


 BOOL fIgnoreKeyboard = FALSE;


 while (true) {
  MSG msg;
  BOOL fMessage;
  if (fIgnoreKeyboard) {
   fMessage =
    PeekMessage(&msg, NULL, 0, WM_KEYFIRST – 1, PM_REMOVE) ||
    PeekMessage(&msg, NULL, WM_KEYLAST + 1, 0xFFFFFFFF, PM_REMOVE);
  } else {
   fMessage = PeekMessage(&msg, NULL, 0, 0, PM_REMOVE);
  }


  if (!fMessage) { WaitMessage(); continue; }


  if (msg.message == WM_QUIT) break;


  if (ShouldLogMessage(msg.message)) {
   prw->LogMessage(&msg);
  }


  if (msg.message == WM_KEYDOWN && msg.wParam == VK_SHIFT) {
   prw->AppendText(TEXT(“Stop processing keyboard messages”));
   fIgnoreKeyboard = TRUE;
  }


  TranslateMessage(&msg);
  DispatchMessage(&msg);
 }
 AttachThreadInput(PtrToInt(lpParameter),
                   GetCurrentThreadId(), FALSE);
 return 0;
}
```

This second thread is intentionally ill-behaved, so that we can see what happens when there's a bad apple in the barrel. The thread processes messages normally, until you hit the shift key. Once that happens, it goes into another mode where it starts ignoring the keyboard by stubbornly refusing to pump any keyboard messages.

Normally, this sort of recalcitrant behavior would affect only the thread itself, but since this thread is attached to the main thread, the scope of the damage expands.

```
int PASCAL
WinMain(HINSTANCE hinst, HINSTANCE, LPSTR, int nShowCmd)
{
 g_hinst = hinst;


 if (SUCCEEDED(CoInitialize(NULL))) {
  InitCommonControls();


  RootWindow *prw = RootWindow::Create();
  if (prw) {
   ShowWindow(prw->GetHWND(), nShowCmd);


  DWORD dwId;
  CreateThread(0, 0, AttachedThreadProc,
               IntToPtr(GetCurrentThreadId()), 0, &dwId);


   MSG msg;
   while (GetMessage(&msg, NULL, 0, 0)) {


   if (ShouldLogMessage(msg.message)) {
    prw->LogMessage(&msg);
   }


    TranslateMessage(&msg);
    DispatchMessage(&msg);
   }
  }
  CoUninitialize();
 }
 return 0;
}
```

We modify our main program to create the secondary thread (which attaches itself to the main thread), and then to log messages in its message pump.

Okay, now run this program and use the mouse to resize and reposition the two windows side by side with no overlap. (This will make it easier to observe what's going on.) Wave the mouse over both windows, and click on each of the windows and do some typing, but don't hit the shift key yet. So far, everything works as you expect: Focus switches back and forth, mouse and keyboard messages are delivered.

Now put focus on the bad window and tap the shift key. This puts the bad window into `fIgnoreKeyboard = TRUE` mode, where it stops pumping keyboard messages (but pumps everything else).

What we just did is leave the `WM_KEYUP` message for the shift key in the input queue, and steadfastly refused to process it. The message just sits there forever. Let's see what this does to the input retrieval algorithm.

Wave the mouse over the bad window. Notice that mouse events are still delivered to the bad window. (Keyboard events are not delivered because the bad thread is not pumping keyboard messages.) This makes sense, because the filtered `PeekMessage` for `WM_KEYLAST + 1` through `0xFFFFFFFF` includes the mouse message range but excludes the keyboard message range, so the loop that looks for a candidate message completely ignores the stuck keyboard message. All it sees are mouse messages, and they are not stuck. The code is taking advantage of the "peek into the future" feature we mentioned yesterday.

Next thing you notice is that if you wave the mouse over the main window, it does *not* receive mouse input. That's because the main window performs an unfiltered peek. The stuck keyboard message satisfies the filter, and since that message belongs to another thread and is ahead of all the mouse messages, the input manager will not return the mouse messages until the stuck keyboard message is cleared out.

This also provides an example of the paradox I alluded to yesterday: The main thread is not receiving any input because it is performing an unfiltered message retrieval, and there is a stuck keyboard message in the input queue. On the other hand, if the main thread had explicitly peeked only for mouse messages, then the stuck keyboard message would not have been taken into consideration, and it would have gotten the mouse messages. The paradox is that under these strange conditions, a filtered message retrieval actually returns messages that an unfiltered retrieval would not!

Now here's another trick: Click on the main window. (Yes, it's not processing mouse input, but do it anyway.) Now both windows stop responding to input. What happened?

Back before you clicked on anything, the only stuck input message was that keyboard message. Sure, there were mouse motions that took place, but we saw that `WM_MOUSEMOVE` messages are generated on demand rather than being posted into the queue when the mouse moves. Therefore, all that mouse-waving didn't actually leave a stuck mouse message in the

queue. On the other hand, when you click, that generates a mouse click event in the queue, and those are generated when the click happens, not on demand. Therefore, when you click on the main window, a click event goes into the input queue.

Now think about what's in the input queue: There is a stuck keyboard message (for the bad window, which is stuck because the bad window refuses to pump keyboard messages), and there is a stuck mouse message (for the main window, which is stuck because the main window is waiting for the stuck keyboard message to clear out). New keyboard input will not be processed because of the stuck keyboard message, and new mouse input will not be processed because of the stuck mouse message.

Result: Nobody gets any input.

Bonus investigation: While you're in this horrible state, open Task Manager. Observe that the scratch program has pegged a CPU. Why is it draining CPU when there is nothing to do?

There's a little extra step in the overall algorithm that describes how input is processed:

- If the input queue is waiting for another thread to finish processing an input message, and the current thread is processing an inbound sent message, then mark the input queue as no longer waiting.
- If the input queue is waiting for another thread to finish processing an input message, then stop and return no message.
- If the input queue is waiting for the current thread to finish processing an input message, then mark the input queue as no longer waiting.
- Look at the first message in the input queue which satisfies the message range filter and either belongs to some other thread or belongs to the current thread and matches the window handle filter.
    - If the message belongs to some other thread, then (New!) nudge the other thread to get it to process the message, then stop. Return no message to the caller.
    - Otherwise, mark the input queue as waiting for the current thread to finish processing an input message, and return the message we found.
- If no such message exists, then there is no input. Return no message.

Reminder: This is a peek under the hood at how the sausage is made, and the algorithm described above is not contractual.

If the algorithm cannot return an input message because there is a stuck input message that belongs to another thread, then the algorithm nudges that other thread by setting the appropriate queue state flag (for example, `QS_KEY` if it is a stuck keyboard message). If the other thread is waiting for that type of message, then the change in queue state will satisfy the wait, and the hope is that other thread will call a message retrieval function to retrieve the stuck message and unclog the input queue.

That explains why the scratch program is pegging a processor. The bad thread wants to peek out a mouse message, but it can't because of the stuck mouse click event that belongs to the main thread, so it nudges the main thread to say, "Hey, I need you to process that mouse event." The main thread wakes up and tries to pump messages, but it can't retrieve any input because of the stuck keyboard message. The main thread therefore nudges the bad thread to say, "Hey, I need you to process that keyboard event."

The two threads are therefore busy taking turns yelling at each other, saying, "Hey, you, you need to get out of my way," and together they burn a CPU.

Now, this is admittedly a pathological program, but it did do a pretty good job of highlighting some of the consequences of synchronous input caused by attaching multiple threads to the same input queue. This is why it's important that threads which share an input queue all be aware of the connection so that they don't accidentally cause trouble for each other.

Exercise: How would you modify the above program to demonstrate the "waiting for a thread to finish processing a message" part of the input message retrieval algorithm?

Raymond Chen

Follow