The dangers of playing focus games when handling a WM_KILLFOCUS message

devblogs.microsoft.com/oldnewthing/20050808-16

August 8, 2005

Raymond Chen

I had noted last year that `WM_KILLFOCUS` is the wrong time to do field validation. Here's another example of how messing with the focus during a `WM_KILLFOCUS` message can create confusion.

Consider an edit control that displays feedback via a balloon tip. For example, password edit controls often warn you if you're typing your password while CapsLock is in effect. One of the things you probably want to do is to remove the balloon tip if the user moves focus to another control, since there's no point telling the user about a problem with something they aren't using. You might be tempted to subclass the edit control and do something like this:

```
LRESULT CALLBACK EditSubclass(
    HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
  switch (uMsg) {
  …
  case WM_KILLFOCUS:
    if (hwndBalloonTip) {
      DestroyWindow(hwndBalloonTip);
      hwndBalloonTip = NULL;
    }
    break;
  …
  }
  return CallWindowProc(prevWndProc, hwnd, uMsg, wParam, lParam);
}
```

When you give this code a shot, it works great… unless the user clicks **on the balloon tip itself** the edit control's caret (the blinking insertion point thingie) disappears. What happened?

What happened is that you gummed up the focus change process by destroying the window that focus was going to! The focus change process goes like this:

- Put focus on new focus window.

- Send WM_KILLFOCUS to old focus window (if any).
- Send WM_SETFOCUS to new focus window (if any).

But in the second step, we **destroyed the new focus window**. When the focus window is destroyed, the window manager tries to find a new focus window, and it settles upon the edit control itself. This starts a recursive focus change cycle, telling the edit control that it now has focus again.

Let's look at the flow in this nested focus change scenario when the user clicks on the tooltip window.

- Put focus on tooltip.
- Send WM_KILLFOCUS to edit control.
    - EditSubclass destroys the tooltip.
        - Window manager puts focus on the edit control.
        - Nobody to send WM_KILLFOCUS to.
        - Send WM_SETFOCUS to edit control.
            EditSubclass passes WM_SETFOCUS to the original window procedure.
    - EditSubclass passes WM_KILLFOCUS to the original window procedure.
- Send WM_SETFOCUS to tooltip – fails (tooltip was destroyed).

Do you see the problem yet?

Look at the message traffic as it reaches the original edit control window procedure:

- WM_SETFOCUS (from the nested focus change)
- WM_KILLFOCUS (from the original focus change)

As far as the edit control is concerned, it gained focus then lost it. Therefore, no caret, since the edit control displays a caret only when it has focus, and your recursive focus changing has resulted in the edit control thinking it doesn't have focus even though it does.

There are many ways out of this mess.

First, notice that you don't need to subclass the edit control; you can just react to the `EN_KILLFOCUS` notification. Second, you can respond to the `EN_KILLFOCUS` by posting yourself a message and destroying the tooltip on receipt of that posted message. By doing it via a posted message, you avoid the recursive focus change since your work is now being done outside a focus change cycle.

Raymond Chen

Follow