Modality, part 4: The importance of setting the correct owner for modal UI

devblogs.microsoft.com/oldnewthing/20050223-00

February 23, 2005

Raymond Chen

If you decide to display some modal UI, it is important that you set the correct owner for that UI. If you fail to heed this rule, you will find yourself chasing some very strange bugs.

Let's return to our scratch program and intentionally introduce a bug related to incorrect owner windows, so that we can see the consequences.

```
void OnChar(HWND hwnd, TCHAR ch, int cRepeat)
{
  switch (ch) {
  case ' ':
    // Wrong!
    MessageBox(NULL, TEXT("Message"), TEXT("Title"), MB_OK);
    if (!IsWindow(hwnd)) MessageBeep(-1);
    break;
  }
}
// Add to WndProc
    HANDLE_MSG(hwnd, WM_CHAR, OnChar);
```

Run this program, press the space bar, and instead of dismissing the message box, click the "X" button in the corner of the main window. Notice that you get a beep before the program exits.

What happened?

The beep is coming from our call to the `MessageBeep` function, which in turn is telling us that our window handle is no longer valid. In a real program which kept its state in per-window instance variables (instead of in globals like we do), you would more likely crash because all the instance variables would have gone away when the window was destroyed. In this case, the window was destroyed while inside a nested modal loop. As a result, when control returned to the caller, it is now a method running inside an object that has been destroyed. Any access to an instance variable is going to access memory that was already freed, resulting in memory corruption or an outright crash.

Here's an explanation in a call stack diagram:

```
WinMain
 DispatchMessage(hwnd, WM_CHAR)
  OnChar
   MessageBox(NULL)
    ... modal dialog loop ...
    DispatchMessage(hwnd, WM_CLOSE)
     DestroyWindow(hwnd)
      WndProc(WM_DESTROY)
       ... clean up the window ...
```

When you clean up the window, you typically destroy all the data structures associated with the window. But notice that you are freeing data structures **that are still being used** by the `OnChar` handler deeper in the stack. Eventually, control unwinds back to the `OnChar`, which is now running with an invalid instance pointer. (If you believe in C++ objects, you would find that its "this" pointer has gone invalid.)

This was caused by failing to set the correct owner for the modal `MessageBox` call, allowing the user to interact with the frame window at a time when the frame window isn't expecting to have its state changed.

Even more problematic, the user can switch back to the frame window and hit the space bar again. The result: Another message box. Repeat another time and you end up with a stack that looks like this:

```
WinMain
 DispatchMessage(hwnd, WM_CHAR)
  OnChar
   MessageBox(NULL)
    ... modal dialog loop ...
    DispatchMessage(hwnd, WM_CHAR)
      OnChar
       MessageBox(NULL)
        ... modal dialog loop ...
        DispatchMessage(hwnd, WM_CHAR)
          OnChar
           MessageBox(NULL)
            ... modal dialog loop ...
```

There are now four top-level windows, all active. If the user dismisses them in any order other than the reverse order in which they were created, you're going to have a problem on your hands. For example, if the user dismisses the second message box first, the part of the stack corresponding to that nesting level will end up returning to a destroyed window when the third message box is finally dismissed.

The fix is simple, and we'll pick up there next time.

Raymond Chen

Follow