Waiting until the dialog box is displayed before doing something

devblogs.microsoft.com/oldnewthing/20060925-02

September 25, 2006

Raymond Chen

Last time, I left you with a few questions. Part of the answer to the first question was given in the comments, so I'll just link to that. The problem is more than just typeahead, though. The dialog box doesn't show itself until all message traffic has gone idle. If you actually ran the code presented in the original message, you'd find that it didn't actually work!

```
#include <windows.h>
INT_PTR CALLBACK
DlgProc(HWND hwnd, UINT uiMsg, WPARAM wParam, LPARAM lParam)
{
  switch (uiMsg) {
  case WM_INITDIALOG:
    PostMessage(hwnd, WM_APP, 0, 0);
    return TRUE;
  case WM_APP:
    MessageBox(hwnd,
              IsWindowVisible(hwnd) ? TEXT("Visible")
                                    : TEXT("Not Visible"),
               TEXT("Title"), MB_OK);
    break;
  case WM_CLOSE:
   EndDialog(hwnd, 0);
   break;
  }
  return FALSE;
}
int WINAPI WinMain(HINSTANCE hinst, HINSTANCE hinstPrev,
                   LPSTR lpCmdLine, int nShowCmd)
{
    DialogBox(hinst, MAKEINTRESOURCE(1), NULL, DlgProc);
    return 0;
}
```

When you run this program, the message box says "Not Visible", and in fact when it appears, you can see that the main dialog is not yet visible. It doesn't show up until after you dismiss the message box.

Mission: Not accomplished.

Along the way, there was some dispute over whether the private message should be `WM_USER` or `WM_APP`. As we saw before, window messages in the `WM_USER` range belong to the window class, and in this case, the window class is the dialog window class, *i.e.*, `WC_DIALOG`. Since you are not the implementor of the dialog window class (you didn't write the window procedure), the `WM_USER` messages are not yours for the taking. And in fact, if you had decided to use `WM_USER` you would have run into all sorts of problems, because it so happens that the dialog manager already defined that message for its own purposes:

```
#define DM_GETDEFID         (WM_USER+0)
```

When the dialog manager sends the dialog a `DM_GETDEFID` message to obtain the default control ID, you will think it's your `WM_USER` message and show your dialog box. It turns out that the dialog manager uses the default control ID rather often, and as a result, you're going to display an awful lot of dialog boxes. (Even worse, your second dialog box will probably use the dialog itself as the owner, which then leads to the problem of having a dialog box with multiple modal children, which then leads to disaster when they are dismissed by the user in the wrong order.)

Okay, so we're agreed that we should use `WM_APP` as the private message.

Some people suggested using a timer, on the theory that timer messages are lower priority than paint messages, so the timer won't fire until all painting is done. While that's true, it also doesn't help. The relative priority of timer and paint messages comes into play only if the window manager has to choose between timers and paint messages when deciding which one to deliver first. But if there are no paint messages needed in the first place, then timers are free to go ahead.

And when the window is not visible, it doesn't need any paint messages. In a sense, the timer approach misses the point entirely: It's trying to take advantage of paint messages being higher priority precisely in the scenario where there are no paint messages!

Let's demonstrate this by implementing the timer approach, but I'm going to add a twist to make the race condition clearer:

```
...
INT_PTR CALLBACK
DlgProc(HWND hwnd, UINT uiMsg, WPARAM wParam, LPARAM lParam)
{
  switch (uiMsg) {
  case WM_INITDIALOG:
    SetTimer(hwnd, 1, 1, 0);
    Sleep(100); //simulate paging
    return TRUE;
  case WM_TIMER:
    if (wParam == 1) {
      KillTimer(hwnd, 1);
      MessageBox(hwnd,
                IsWindowVisible(hwnd) ? TEXT("Visible")
                                      : TEXT("Not Visible"),
                 TEXT("Title"), MB_OK);
    }
    break;
  case WM_CLOSE:
   EndDialog(hwnd, 0);
   break;
  }
  return FALSE;
}
```

If you run this program, you'll see the message "Not Visible". I inserted an artificial `Sleep(100)` to simulate the case where the code takes a page fault and has to wait 100ms for the code to arrive from the backing store. (Maybe it's coming from the network or a CD-ROM, or maybe the local hard drive is swamped with I/O and you have to wait that long for your paging request to become satisfied after all the other I/O requests active on the drive.)

As a result of that `Sleep()`, the dialog manager doesn't get a chance to empty the message queue and show the window because the timer message is already in the queue. Result: The timer fires and the dialog is still hidden.

Some people waited for `WM_ACTIVATE`, but that tells you when the window becomes active, which is not the same as being shown, so it doesn't satisfy the original requirements.

Others suggested waiting for `WM_PAINT`, but a window can be visible without painting. The `WM_PAINT` message arrives if the window's client area is uncovered, but the caption bar might still be visible even if the client area is covered. Furthermore, while this addresses the problem if you interpret "visible" as "results in pixels on the screen", as opposed to `IsWindowVisible`, you need to look behind the actual request to what the person was really looking for. (This is an important skill to have because people rarely ask for what they want, but rather for what they think they want.) The goal was to create a dialog box and have it look like the user automatically clicked a button on it to call up a secondary dialog. In order to get this look, the base dialog needs to be visible before the secondary dialog can be displayed.

One approach is to show the second dialog on receipt of the `WM_SHOWWINDOW`, but even that is too soon:

```
// In real life, this would be an instance variable
BOOL g_fShown = FALSE;
INT_PTR CALLBACK
DlgProc(HWND hwnd, UINT uiMsg, WPARAM wParam, LPARAM lParam)
{
  switch (uiMsg) {
  case WM_INITDIALOG:
    return TRUE;
  case WM_SHOWWINDOW:
    if (wParam && !g_fShown) {
      g_fShown = TRUE;
      MessageBox(hwnd,
                 IsWindowVisible(hwnd) ? TEXT("Visible")
                                       : TEXT("Not Visible"),
                 TEXT("Title"), MB_OK);
    }
    break;
  case WM_CLOSE:
   EndDialog(hwnd, 0);
   break;
  }
  return FALSE;
}
```

(Subtlety: Why do I set `g_fShown = TRUE` before displaying the message box?)

If you run this program, you will still get the message “Not Visible” because `WM_SHOWWINDOW` is sent as part of the entire window-showing process. At the time you receive it, your window is **in the process of being show** but it’s not quite there yet. The `WM_SHOWWINDOW` serves a similar purpose to `WM_INITDIALOG`: To let you prepare the window while it’s still hidden so the user won’t see ugly flashing which would otherwise occur if you had done your preparation after the window were visible.

Is there a message that is sent after the window has been shown? There sure is: `WM_WINDOWPOSCHANGED`.

```
INT_PTR CALLBACK
DlgProc(HWND hwnd, UINT uiMsg, WPARAM wParam, LPARAM lParam)
{
  switch (uiMsg) {
  case WM_INITDIALOG:
    return TRUE;
  case WM_WINDOWPOSCHANGED:
    if ((((WINDOWPOS*)lParam)->flags & SWP_SHOWWINDOW) &&
        !g_fShown) {
      g_fShown = TRUE;
      MessageBox(hwnd,
                 IsWindowVisible(hwnd) ? TEXT("Visible")
                                       : TEXT("Not Visible"),
                 TEXT("Title"), MB_OK);
    }
    break;
  case WM_CLOSE:
   EndDialog(hwnd, 0);
   break;
  }
  return FALSE;
}
```

This time, we get the message “Visible”, because `WM_WINDOWPOSCHANGED` is sent after the window positioning negotiations are complete. (The “ED” at the end emphasizes that it is delivered after the operation has been done, as opposed to the “ING” which is delivered while the operation is in progress.)

But wait, we’re not out of the woods yet. Although it’s true that the window position negotiations are complete, the message is nevertheless sent as part of the whole window positioning process, and there may be other things that need to be done as part of the whole window-showing bookkeeping. If you show the second dialog directly in your `WM_WINDOWPOSCHANGED` handler, then that clean-up won’t happen until after the user exits the second dialog.

For example, the window manager notifies Active Accessibility of the completed window positioning operation after all the window positions have settled down. This reduces the likelihood that the accessibility tool will be told “Okay, the window is shown” followed by “Oh no wait, it moved again, ha ha!” If you display the second dialog inside your `WM_WINDOWPOSCHANGED` handler, the screen reader will receive a bizarro sequence of events:

- Second dialog shown.
- (User interacts with second dialog and dismisses it.)
- Second dialog destroyed.
- (Your `WM_WINDOWPOSCHANGED` handler returns.)
- Main dialog shown.

Notice that the “Main dialog shown” notification arrives out of order because you did additional UI work before the previous operation was complete.

As another example, the window may have been shown as part of a multiple-window window positioning operation such as one created by `DeferWindowPos`. All the affected windows will get their `WM_WINDOWPOSCHANGED` notifications one at a time, and if your window happened to go first, then those other windows won’t know that they were repositioned until after the user finishes with the nested dialog. This may manifest itself in those other windows appearing to be “stuck” since your dialog is holding up the subsequent notifications with your nested dialog. For example, a window might be trying to do **exactly what you’re trying to do here**, but since you’re holding up the remainder of the notifications, that other window won’t display its secondary dialog until the user dismisses yours. From the user’s standpoint, that other window is “stuck” for no apparent reason.

Therefore, we need one more tweak to our solution.

```
INT_PTR CALLBACK
DlgProc(HWND hwnd, UINT uiMsg, WPARAM wParam, LPARAM lParam)
{
  switch (uiMsg) {
  case WM_INITDIALOG:
    return TRUE;
  case WM_WINDOWPOSCHANGED:
    if ((((WINDOWPOS*)lParam)->flags & SWP_SHOWWINDOW) &&
        !g_fShown) {
      g_fShown = TRUE;
      PostMessage(hwnd, WM_APP, 0, 0);
    }
    break;
  case WM_APP:
      MessageBox(hwnd,
                 IsWindowVisible(hwnd) ? TEXT("Visible")
                                       : TEXT("Not Visible"),
                 TEXT("Title"), MB_OK);
      break;
  case WM_CLOSE:
   EndDialog(hwnd, 0);
   break;
  }
  return FALSE;
}
```

When we learn that the dialog is being shown for the first time, we post a message to ourselves to display the secondary dialog and return from the `WM_WINDOWPOSCHANGED` handler. This allows the window positioning operation to complete. Everybody gets their notifications, they are all on board with the state of the windows, and only after everything has stabilized do we display our message box.

This is a common thread to many types of window management. Many window messages are notifications which are delivered **while the operation is still in progress**. You do not want to display new UI while handling those notifications because that holds up the completion of the original UI operation that generated the notification in the first place. Posting a message to yourself to complete the user interaction after the original operation has stabilized is the standard solution.

Raymond Chen

Follow