Modality, part 7: A timed MessageBox, the cheap version

devblogs.microsoft.com/oldnewthing/20050301-00

March 1, 2005

Raymond Chen

As we noted at the end of part 3, now that you know the conventions surrounding the `WM_QUIT` message you can put them to your advantage.

The more robust you want the `TimedMessageBox` function to be, the more work you need to do. Here's the cheap version, based on the sample in the Knowledge Base, but with some additional bug fixes.

```
static BOOL s_fTimedOut;
static HWND s_hwndMBOwnerEnable;
void CALLBACK
CheapMsgBoxTooLateProc(HWND hWnd, UINT uiMsg, UINT_PTR idEvent, DWORD dwTime)
{
    s_fTimedOut = TRUE;
    if (s_hwndMBOwnerEnable) EnableWindow(s_hwndMBOwnerEnable, TRUE);
    PostQuitMessage(42); // value not important
}
// Warning! Not thread-safe! See discussion.
int CheapTimedMessageBox(HWND hwndOwner, LPCTSTR ptszText,
    LPCTSTR ptszCaption, UINT uType, DWORD dwTimeout)
{
    s_fTimedOut = FALSE;
    s_hwndMBOwnerEnable = NULL;
    if (hwndOwner && IsWindowEnabled(hwndOwner)) {
      s_hwndMBOwnerEnable = hwndOwner;
    }
    UINT idTimer = SetTimer(NULL, 0, dwTimeout, CheapMsgBoxTooLateProc);
    int iResult = MessageBox(hwndOwner, ptszText, ptszCaption, uType);
    if (idTimer) KillTimer(NULL, idTimer);
    if (s_fTimedOut) {                  // We timed out
        MSG msg;
        // Eat the fake WM_QUIT message we generated
        PeekMessage(&msg, NULL, WM_QUIT, WM_QUIT, PM_REMOVE);
        iResult = -1;
    }
    return iResult;
}
```

This `CheapTimedMessageBox` function acts just like the `MessageBox` function, except that if the user doesn't respond within `dwTimeout` milliseconds, we return -1. The limitation is that only one timed message box can be active at a time. If your program is single-threaded, this is not a serious limitation, but if your program is multi-threaded, this will be a problem.

Do you see how it works?

The global static variable `s_fTimedOut` tells us whether we generated a fake `WM_QUIT` message as a result of a timeout. When the `MessageBox` function returns, and we indeed timed out, we use the `PeekMessage` function to remove the fake `WM_QUIT` message from the queue before returning.

Note that we remove the `WM_QUIT` message only if we were the ones who generated it. In this way, `WM_QUIT` messages generated by other parts of the program remain in the queue for processing by the main message loop.

Note also that when we decide that the timeout has occurred, we re-enable the original owner window before we cause the message box to bail out of its message loop by posting a quit message. Those are the rules for the correct order for disabling and enabling windows.

Note also that we used a thread timer rather than a window timer. That's because we don't own the window being passed in and therefore don't know what timer IDs are safe to use. Any timer ID we pick might happen to collide with a timer ID being used by that window, resulting in erratic behavior.

Recall that when you pass `NULL` as the `hwnd` parameter to the `SetTimer` function and also pass zero as the `nIDEvent` parameter, then the `SetTimer` function creates a brand new timer, assigns it a unique ID, and returns the ID. Most people, when they read that part of the specification for `SetTimer`, scratch their heads and ask themselves, "Why would anybody want to use this?"

Well, this is one scenario where this is exactly what you want.

Next comes the job of making the function a tad more robust. But before we do that, we'll need two quick sidebars.

Raymond Chen

Follow