Why can’t I create a “Please wait” dialog from a background thread to inform the user that the main UI thread is busy?

devblogs.microsoft.com/oldnewthing/20191023-00

October 23, 2019

Raymond Chen

A customer had a program which performed a long-running operation on its main UI thread. They wanted to display a “Please wait” dialog from a background thread, so they did something like this:

```
void OnClick(HWND mainWindow)
{
  PleaseWaitDialog dialog;
  dialog.Start(mainWindow);
  DoSomeReallyLongOperation();
  dialog.Stop();
}

class PleaseWaitDialog
{
  void Start(HWND mainWindow)
  {
    ...
    m_mainWindow = mainWindow;
    CreateThread(nullptr, 0,
      PleaseWaitThreadProc, this,
      0, &threadId);
    ...
  }

  static DWORD CALLBACK PleaseWaitThreadProc(void* parameter)
  {
    auto self = reinterpret_cast<PleaseWaitDialog*>(parameter);
    DialogBox(instance, MAKEINTRESOURCE(IDD_WAIT),
              self->mainWindow, DialogProc);
    return 0;
  }

  HWND m_mainWindow;
};
```

The `PleaseWaitDialog` class is incomplete, but that's the general idea: We create a separate thread to display the dialog box, and make it modal to the main window so the user can see which window it is associated with.

The problem is that this doesn't work.

When the dialog box sets the main UI window as its owner, this causes the input queues to become attached, at which point their fates become linked. In particular, the dialog box cannot show itself because doing so requires it to notify the owner window that the owner has lost activation, but that owner window is not responding to messages because it's off doing the really long operation.

There are a few ways to address this.

One way is to make the long-running operation pump messages occasionally:

```
void HandleMessages()
{
 MSG msg;
 while (PeekMessage(&msg, nullptr, 0, 0, PM_REMOVE)) {
  TranslateMessage(&msg):
  DispatchMessage(&msg):
 }
}

void DoSomeReallyLongOperation()
{
  for (auto&& item : items) {
   item.DoSomething();
   HandleMessages(); // pump messages between each item
  }
}

void Item::DoSomething()
{
 StartSomething();
 HandleMessages(); // pump messages to remain responsive
 ContinueSomething();
 HandleMessages(); // pump messages to remain responsive
 FinishSomething();
}
```

This does require you to litter `HandleMessages` calls throughout your long-running operation. If your operation is cancellable, then you could have the `HandleMessages` function return whether the user clicked the *Cancel* button in the *Please Wait* dialog, and callers could abandon the operation.

This improves the situation from *unresponsive* to *sluggishly responsive*, because the UI thread doesn't respond to actions immediately; rather, it responds to them only when it remembers to check.

A more serious problem with this design is that pumping messages may create reentrancy problems. For example, if a message arrives like `WM_SETTINGSCHANGE`, the program may start responding to the change in settings while it was in the middle of an operation, and that might confuse the operation already in progress. For example, an incoming message might trigger a change to the `items` collection, which is bad news because the `for` loop is iterating over that same collection. It might even destroy the item that the loop is actively operating on!

The best way to solve the problem is to switch the roles of the two threads. The UI thread displays the progress dialog, and the background thread performs the long-running operation.

Of course, due to architectural decisions made elsewhere in the program, this is often easier to say than to do.

Raymond Chen

Follow