Reading a contract from the other side: SHSetInstanceExplorer and SHGetInstanceExplorer

devblogs.microsoft.com/oldnewthing/20080528-00

May 28, 2008

Raymond Chen

Shell extensions that create worker threads need to call the `SHGetInstanceExplorer` function so that Explorer will not exit while the worker thread is still running. When your worker thread finishes, you release the `IUnknown` that you obtained to tell the host program, "Okay, I'm done now, thanks for waiting."

You can read this contract from the other side. Instead of thinking of yourself as the shell extension running inside a host program, think of yourself as the host program that has a shell extension running inside of it. Consider a simple program that displays the properties of a file, or at least tries to:

```
#include <windows.h>
#include <shellapi.h>
#include <tchar.h>
int __cdecl _tmain(int argc, TCHAR **argv)
{
  SHELLEXECUTEINFO sei = { sizeof(sei) };
  sei.fMask = SEE_MASK_FLAG_DDEWAIT | SEE_MASK_INVOKEIDLIST;
  sei.nShow = SW_SHOWNORMAL;
  sei.lpVerb = TEXT("properties");
  sei.lpFile = TEXT("C:\\Windows\\Explorer.exe");
  ShellExecuteEx(&sei);
  return 0;
}
```

Oh dear! When you run this program, nothing happens. Well, actually, something did happen, but the program exited too fast for you to see it. To slow things down, add the line

```
  MessageBox(NULL, TEXT("Waiting"), TEXT("Title"), MB_OK);
```

right before the `return 0`. Run the program again, and this time the properties dialog appears, as well as the message box. Aha, the problem is that our program is exiting while the property sheet is still active. (Now delete that call to `MessageBox` before something stupid happens.)

The question now is how to know when the property sheet is finished so we can exit. That's where `SHSetInstanceExplorer` comes in. The name "Explorer" in the function name is really a placeholder for "the host program"; it just happens to be called "Explorer" because the function was written from the point of view of the shell extension, and the host program is nearly always Explorer.exe.

In this case, however, we are the host program, not Explorer. The `SHSetInstanceExplorer` lets you register a free-threaded `IUnknown` that shell extensions can obtain by calling `SHGetInstanceExplorer`. Following COM reference counting conventions, the `SHGetInstanceExplorer` performs an `AddRef()` on the `IUnknown` that it returns; the shell extension's worker thread performs the corresponding `Release()` when it is finished.

All that is required of the `IUnknown` that you pass to `SHSetInstanceExplorer` is that it be free-threaded; in other words, that it support being called from multiple threads. This means managing the "process reference count" with interlocked functions rather than boring `++` and `--` operators. Of course, in practice, you also need to tell your main program "Okay, all the shell extensions are finished; you can exit now" when the reference count drops to zero.

There are many ways to accomplish this task. Here's one that I threw together just now. I didn't think too hard about this class; I'm not positioning this as the best way of implementing it, or even as a particularly good one. The purpose of this article is to show the *principle* behind process references. Once you understand that, you are free to go ahead and solve the problem your own way. But here's *a* way.

```
#include <shlobj.h>
class ProcessReference : public IUnknown {
public:
  STDMETHODIMP QueryInterface(REFIID riid, void **ppv)
  {
    if (riid == IID_IUnknown) {
      *ppv = static_cast<IUnknown*>(this);
      AddRef();
      return S_OK;
    }
   *ppv = NULL; return E_NOINTERFACE;
  }
  STDMETHODIMP_(ULONG) AddRef()
    { return InterlockedIncrement(&m_cRef); }
  STDMETHODIMP_(ULONG) Release()
  {
    LONG lRef = InterlockedDecrement(&m_cRef);
    if (lRef == 0) PostThreadMessage(m_dwThread, WM_NULL, 0, 0);
    return lRef;
  }
  ProcessReference()
    : m_cRef(1), m_dwThread(GetCurrentThreadId())
    { SHSetInstanceExplorer(this); }
  ~ProcessReference()
  {
    SHSetInstanceExplorer(NULL);
    Release();
    MSG msg;
    while (m_cRef && GetMessage(&msg, NULL, 0, 0)) {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
  }
private:
  LONG m_cRef;
  DWORD m_dwThread;
};
```

The idea behind this class is that the main thread (and only the main thread) creates it on the stack. When constructed, the object registers itself as the "process `IUnknown`"; any shell extensions that call `SHGetInstanceExplorer` will get a pointer to this object. When the object is destructed, it unregisters itself as the process reference (to avoid dangling references) and waits for the reference count to drop to zero. Notice that the `Release` method posts a dummy thread message so that the "waiting for the reference count to go to zero" message loop will wake up.

In a sense, this is backwards from the way normal COM objects work, which operate under the principle of "When the reference count drops to zero, the object is destructed." We turn it around and code it up as "when the object is destructed, it waits for the reference count to

drop to zero." If you wanted to do it the more traditional COM way, you could have the main thread go into a wait loop and have the object's destructor signal the main thread. I did it this way because it makes using the class very convenient.

Now that we have a process reference object, it's a simple matter of adding it to our main thread:

```
int __cdecl _tmain(int argc, TCHAR **argv)
{
  ProcessReference ref;
  ...
```

With this modification, the program displays the property sheet and patiently waits for the property sheet to be dismissed before it exits.

Exercise: Explain how the object behaves if we initialized the reference count to zero and deleted the call to `Release` in the destructor.

Bonus reading: Do you know when your destructors run? Part 2.

Raymond Chen

Follow