Give ActiveX-based Web pages a boost with the apartment threading model. (Microsoft's application development software) (Technology Tutorial)(Tutorial)

David Platt
8,709 words
1 February 1997
Microsoft Systems Journal
MSJN
17
Vol. 12, No. 2, ISSN: 0889-9932
English

You can significantly can significantly boost the performance of your Activex[TM] controls in multi-threaded containers by making a few small modifications to your existing code. Why is this important? Because the first Web browser to support Activex controls, Microsoft[R] Internet Explorer 3.0 (IE 3.0), is a multithreaded container, and the forthcoming IE 4.0 (also known as Nashville) makes even more extensive use of threading. The performance profits that you can gain are not confined to these apps, but will apply to any multithreaded container that uses your controls.

This article will tell you how to make a significant performance gain without a whole lot of work--fairly large bang, not too many bucks. As I take you through the process step by step, I will also explain how COM deals with threads, particularly as they apply to Activex controls. COM has a reputation for being hard to learn. This reputation stems primarily from the prevailing instruction model, which presents theory first, then examples long after your eyes glaze over. I've written an entire book dedicated to the opposite approach (The Essence of OLE with ActiveX, Prentice-Hall), showing the simplest possible example and using it to explain the theory. Nobody I've ever met, myself included, can stay awake for the theory without having an example to pin it on.

I strongly suggest that you download the sample code (see page 5 for instructions) and work along as you read this article. The sample programs are each in their own sample directory (provided you unzip the file with the -d switch). Throughout this article I refer to sample directories; these are in the download file. The first sample control requires the MFC 4.0 DLLs, and the others require MFC 4.2.

Container Environment

Before examining controls in multithreaded containers, it would help to look at the container environment in which they will be running. Beginning with Windows NT[R] 3.51 and continuing in Windows[R] 95, COM has supported the use of objects on different threads in a manner dissected in detail later in this article. The container follows three simple rules and COM makes it all work.

First, every container thread that wants to use COM objects must call the function CoInitializeEx when it starts up and CoUninitialize when it shuts down. The first thread in the app that calls CoInitializeEx is called "COM's main thread" in that app. It doesn't have to be the app's main thread, the one on which WinMain was called, but for convenience it usually is. Unless I state otherwise, whenever I use the term "main thread" in the rest of this article, I am referring to COM's main thread. As the first thread to initialize COM, the main thread must also be the last thread to uninitialize COM.

Second, each thread that calls CoInitializeEx using the COINIT_APARTMENTTHREADED flag must have a message loop and service it frequently. (Note that calling the CoInitialize API is equivalent to calling CoInitializeEx with the COINIT_APARTMENTTHREADED flag.) COM uses private messages to hidden windows for its own internal purposes, as described later. If you fail to get and dispatch the messages that appear in a thread's message queue, you will break COM. You probably don't have to change your code a bit. The normal message loop you've been using in your main thread has been doing exactly this since day one, and you probably haven't heard a thing about COM and window messages. They've always been there; you just haven't had any reason to notice. If your new threads are going to contain windows, you'll have to do it anyway. Just make sure you don't plan a design with a thread that wants to use COM but doesn't check a message queue. (The message-loop requirement does not apply to multithreaded apartments--threads on which CoInitializeEx was called with the COINIT_MULTITHREADED flag.)

The final rule that the container must follow is that, when a thread obtains a pointer to a COM interface via any legal mechanism, that interface's methods may only be called from that thread. For example, if a thread calls CoCreateInstance and obtains a pointer to an interface, that interface's methods may only be called from within the thread that called CoCreateInstance. If this seems restrictive, fear not; there are legal workarounds described later in this article. But keep this rule in mind and you'll have the right mental model.

If you follow the three simple rules I just discussed you'll have a single-threaded apartment (STA) model container. Your threads are single-threaded apartment threads. That really reduces to following a few simple, consistent, easy rules. All of COM is like that, once you get to know it.

Nonthreaded Control

Suppose I have the simplest possible full-featured Activex control. I used the MFC 4.0 Control Wizard to generate a new Activex control project, accepting all the default options. I then added two lines to COleControl::OnDraw, using the API GetCurrentThreadId to get the ID of the thread on which this function was called, and outputting that ID in the control's window. This code is shown in Figure 1, and it's about as simple as it gets. You will find the control and code in the sample directory \demo40. You will have to register the control demo40.ocx by using a command-line utility like regsvr32.exe.

I used the Activex Control Pad to place it on a page called demo40.htm. I then opened this page with IE 3.0. The control was created and shown in the IE 3.0 window. I then chose File New Window from the IE 3.0 main menu, bringing up another window showing the same page and containing another instance of my simple control. IE 3.0 does this by launching a new thread, not a whole new process, which you can verify by using PView. My screen then looked like Figure 2. The thread ID was the same for both controls, even though IE 3.0 created a new thread to handle the new page and this new thread called CreateControl internally. What is going on?

COM has not always supported multiple threads. It didn't in 16-bit Windows, obviously, and it didn't in Windows NT 3.5. In the latter, only one thread in a process was allowed to use COM and all COM function and object calls had to be made from this thread. If another thread tried to call CoInitialize, the call would fail.

With the release of Windows NT 3.51, COM added support for multiple threads. The approach used requires some cooperation from the object's server to keep controls on different threads from stepping on each other's toes, but few control servers provided this cooperation at the time. To keep from breaking every control in existence, COM had to provide a mechanism for detecting whether or not a server knew how to handle itself in a multithreaded environment and for protecting the ones that couldn't while still allowing them to be used.

A control that has taken the necessary precautions to allow itself to be accessed by multiple threads identifies itself as such by making an entry in the registry (a process I'll describe later). When a container app calls CoCreateInstance to create a control, COM checks for the presence of this entry. In its absence, COM assumes that the control's DLL server has no knowledge of threads whatsoever. Controls built with MFC 4.1 or earlier, such as this example, fall into this category. If an object method that accessed global data was called on one thread, and another object method that accessed the same global data was called on another thread, the threads could conflict with each other and chaos would result.

For example, consider the code listing in Figure 3. Suppose I have a COM class called CSomeObject that has two member functions, AllocIfNeededAndUseMemory and FreeMemory. The first one checks to see if a pointer is NULL and, if so, allocates and initializes memory before using the pointer. The second one frees the memory and sets the pointer value back to NULL. The comment in the code points out the exact sequence of events needed for a conflict. While this example looks pretty silly, it was a reasonable way to write code back in the world of 16-bit COM and Windows NT 3.5, where you didn't have to think about preemptive multithreading.

To solve this problem, COM sets up marshaling code conceptually similar to that used for marshaling calls to another process, which ensures that all controls provided by that DLL are created on the container's main thread, the one that first called CoInitialize (or CoInitializeEx). This marshaling code also ensures that all calls to all methods of these controls are made only from the container's main thread. This way, all calls to the control are serialized in the manner that the control is expecting.

Since the thread handling the second window in the previous example wasn't the main thread, COM transparently marshaled the object creation into the main thread to avoid any possible conflict with other objects from the same DLL. The interface pointer returned to the second thread did not point directly to the code inside the object, as was the case in a single-thread situation. Instead, it pointed to a marshaling proxy object on the second thread. All of this happened within the second thread's call to CoCreateInstance. When the second thread calls a method on this object, the proxy marshals the call to the main thread and marshals the results back to the second thread. This thread doesn't know or care whether it has a pointer to the actual object or to a marshaling proxy; it just calls the methods locally and the right thing happens. Each side follows

the relatively simple COM rules, and COM intercedes as necessary to make it work transparently. The flow is shown in Figure 4.

While this may seem like an exotic and unusual thing for COM to do, it really isn't. Window message handling works the same way. A window's response function will always be called from the thread that created the window. Ever wonder why you don't have to serialize access to window message handlers? It's because the operating system marshals window messages the same way that COM does. In fact, COM accomplishes its marshaling by creating hidden windows and posting messages to them. You can see these windows in Spy++. They're the ones belonging to class OleMainThreadWndClass and having the text OleMainThreadWndName. For a full discussion of the blood and guts of multithreaded marshaling, see Don Box's column in the April 1996 issue of MSJ.

My container app can run with any control and vice versa, regardless of the threading used by either one, and I get my app out the door quickly. So what's the bad news? This marshaling takes time and burns CPU cycles. How bad a hit? It's tricky to measure; enough that you don't want it if you can avoid it. A ballpark figure is that marshaling a call from one thread into another takes about as much time as marshaling to another process. This means a call overhead of a few milliseconds each. Obviously, this can vary widely. It's worse in Windows 95 than in Windows NT. The more data you are passing, the longer it takes to copy, and if you are passing so much that the copy involves disk swapping, the delay can skyrocket. Worse, it's not just the extra call overhead you have to deal with. The main thread to which the call is being marshaled might be busy doing a calculation or reading a long file, in which case the thread that originates the call will have to wait until the main thread finishes. Imagine five or six threads each creating ten controls, all needing all of their method calls marshaled back to the main thread. You've lost the prime benefit of multithreading. Life becomes a weary burden.

Apartment Model-aware Control

Is there a way around this? Fortunately, yes, and it's relatively easy. I generated and built the same control as the previous example, only this time using MFC 4.2, and put it on an HTML page called demo42.htm. You will find this page and the control demo42.ocx in the sample code directory \demo42. If you register the control, open the page with IE 3.0, and then open another IE 3.0 window showing the same page, you will find that the thread IDs are now different as shown in Figure 5. If you use Pview, you will find that the thread ID reported by the new control is the same as the new thread that IE 3.0 created to manage the new window. Unlike the previous example, the control has somehow been created on the new thread. What happened?

Every server that provides an object to COM needs to specify how the object's functions may be called from multiple threads so that COM will know how to intercede between the object and its user. There are three choices. First, the server can refrain from specifying anything, as in the first example. In this case, COM sets up marshaling code to force all calls to all objects from that server to be made from a single thread (the main thread). Second, the server can specify that any of the object's functions may be called from any thread at any time; that the object is reentrant or internally synchronized to whatever extent it requires, so it's perfectly safe for any thread to do anything to it at any time. This is the free threading model--also called the multithreaded apartment (MTA) model which is new in Windows NT 4.0 and is now available in Windows 95 with the DCOM for Windows 95 update. Because writing fully thread-safe code is complicated and not widely used, I'll deal with it at the end of this article.

The middle ground that COM has provided in Windows 95 and Windows NT (since version 3.51) is called the apartment threading model (more correctly called the STA model). A server that specifies its support for this model is promising COM that the server has serialized access to all of its global data, such as its object count. An apartment model object has not serialized access to the object's internal data, so if one thread calls a method on an object and gets swapped out in the middle, then another thread calls a method on the same object, the two calls could conflict. In the absence of the workaround described later in this article, COM requires you to call an object's methods only from the thread on which that object was created. Since this is the case, the object does not need to serialize access to its member variables, as these will always be called from the same thread. The objects do need to serialize access to the server's globals, as different objects may try to access these from different threads.

A control that supports this model is saying to COM, "Hey, I've thought about threading, so I've already taken care of serializing access to my own global data. You don't have to make all calls into me on a single thread the way you did with the previous idiot. Just make sure you call each of my objects on the thread that created it, and I've made sure they'll never step on each other's toes."

Activex controls created with MFC 4.2 or later support the single-threaded apartment model by default. A control that supports the single-threaded apartment model signals this to COM by making an entry in the registry as shown in Figure 6. The control's InProcServer32 registry key contains an additional named value, ThreadingModel. If this value contains the string data "Apartment," the control is promising that it supports the single-threaded apartment model. MFC has already taken care of the global object count for your control. If

you are creating controls with MFC, then all you have to do is to serialize access to your globals as shown in the example coming up shortly.

An MFC control registers itself as using the apartment model in the method COleControl::COleObjectFactory::UpdateRegistry, as shown in Figure 7. Control Wizard automatically generates this method for you. Its primary feature is a call to the function AfxOleRegisterControlClass, which makes the actual registry entries. The sixth parameter to this function is the constant afxRegApartment-Threading, which tells the function to register the control as supporting the apartment model. If you are writing a control with MFC version 4.2 or later and do not want to support apartment model threading, you must change this parameter to zero. Conversely, if you are upgrading a control that was originally generated with MFC version 4.1 or earlier and now want to support the apartment model, this parameter as generated in your original code will be FALSE and you must change it to afxRegApartment-Threading.

Consider the code listing in Figure 3 that I used as an example in the previous section. It isn't hard to write code that makes sure that different threads never conflict over the same global variable. The code listing in Figure 8 is similar to Figure 3, but this time properly serialized. The quickest, easiest, and cheapest way of serializing access is with a critical section. You must write code to serialize access to any other globals that your control DLL contains. This also applies to class statics, which are really just a politically correct form of global.

When a thread in the container calls CoCreateInstance to create an instance of control that supports the STA model, COM will create the object on the calling thread instead of marshaling everything over to the main thread as was done in the previous section. Since the creating thread has been initialized as STA, COM knows that it promised to call the object's functions only from the thread that created the object. COM therefore does not set up the marshaling code as it did in the previous example, so the interface pointer returned is a direct pointer to the object's code, as you would otherwise expect from an in-proc object. The two kids promise to play together nicely, so Mom leaves them unsupervised. This is the performance profit I referred before.

What if the creating thread wasn't initialized in the STA model? At the time of this writing, almost all of them are, and most of them will remain so for the foreseeable future. But again, you don't really have to care. Because of the control's registry entries, COM knows what it requires and will set up marshaling code if necessary so that your control sees what it expects. I deal with this issue towards the end of this article. The short answer is, don't worry about it; COM takes care of it.

Design Considerations

Your controls may want to create auxiliary threads to help them accomplish their work. If you do this, you should keep a few design guidelines in mind. In this section, I discuss the simple case of a generic background thread that does not access COM in any way. Then I will discuss legal ways in which this background thread can make calls on COM objects created by other threads.

Consider a simple control that creates a simple background thread. I wrote a control called BounceThread.ocx and placed it on an HTML page called bounce.htm, both of which you will find in the directory \BounceThread. If you register the former and open the latter with IE 3.0, you will see a window in which a bouncing ball ricochets around drawing a colored tail (see Figure 9).

To create an auxiliary thread in the MFC, I derived a class called CBounceThread from the MFC base class CWinThread. The code for creating the thread is shown in Figure 10. When the control receives its WM_CREATE message, it uses the function AfxBegin-Thread to create an object of this class. It's created in the suspended state so the code can initialize its member variables before it goes charging off, bouncing its ball around the control's window. After this, it calls ResumeThread and goes off on its merry way.

Creating the thread is easy; destroying it is trickier. When you get tired of the thread, you can't just blow it away via the function TerminateThread. This leads to all kinds of nasty consequences, such as not deallocating the thread's stack, not releasing mutexes owned by the thread, and not notifying any DLLs of the thread termination--just not a good thing to do at all. It might just be OK if you knew that your entire app was going down, but that isn't the case in an Activex control. The control could very easily be destroyed because the user surfed to another page. You have to find another approach.

To get rid of a thread, it is a much better idea to have its cooperation. The MFC base class CWinThread already provides this in its handling of the WM_QUIT message. When this message appears in the message queue of a CWinThread, it nicely exits its message loop, calls its Exit-Instance method, cleans up after itself, and quietly vanishes. In the sample control, this is done in the control's WM_DESTROY message handler, as shown in Figure 11. After posting this message, the thread's priority is raised to help it finish up whatever it has to do before it dies. The control is probably being destroyed as a result of a direct command from the

user, so this is a good use of a high priority. Since the thread accesses the control's member variables in its processing, notably the control's m_hWnd for getting a DC, I don't want the control to disappear before the thread. I therefore use the API function WaitForSingleObject to wait for the thread to die, finish my control's cleanup, and leave.

The thread needs to make sure that it keeps checking its message queue so it responds to the WM_QUIT message when one arrives. If it needs to wait on a synchronization object, the thread should do so via the function MsgWait-ForMultipleObjects so that the wait will return if a message comes into the thread's message queue. If you are not using the CWinThread MFC base class, you will have to write your own mechanism for signaling the thread to shut itself down.

The other major design consideration is to make sure that the auxiliary thread doesn't have such a high priority that it starves other threads of needed CPU cycles. If you rightclick on the control in IE 3.0, you will find that I've put a context menu on it allowing you to experiment with different priority levels. Anything higher than THREAD_PRIORITY_NORMAL will make the user interface very unresponsive. Don't try this with unsaved data in any of your other apps.

Marshaling Interface Pointers

Nowhere are the intricacies of multithreaded COM objects more delicate than in the interaction between an Activex control and its container. In a control-container situation, the distinction between client and server becomes meaningless. It is customary to think of a control as a server and its container as a client. A full-featured control provides at least seven interface pointers that the container may call. However, the control's container provides at least six interface pointers that the control can call and frequently does. So each side is the client of a handful of interfaces and the server of another handful.

Suppose I have a control that performs an operation that could take a long time to complete, for example, searching a large or slow database. I would like my control to have a method that starts the operation and returns immediately, rather than hanging the calling thread by not returning until the long operation completes. When the operation finally does complete, I want it to signal its completion asynchronously via an event.

An easy way to do this is for the control to create an auxiliary thread that performs the search operation, and for this thread to signal the completion of its search by firing an event. It seems simple enough, but there's a hidden problem. Events are signaled by calling the Invoke method of an IDispatch interface supplied by the container. This interface lives in the apartment of its own thread, which is also the one containing the control itself. To comply with the STA model, I can only safely call this interface's methods from the thread in which it was created. If the control spins off another thread, when the new thread wants to signal back, it will be calling an object pointer on a thread other than the one that created it. Doing this would violate the rules of the apartment threading model.

Fortunately, as usual, there's a legal workaround that isn't too painful. Look back at the first simple example where COM set up marshaling code to make sure that an object's methods were only called on the thread from which the object was expecting it. In that case, all object calls were marshaled to the main thread. I can set up marshaling code myself, which will allow the container's event IDispatch to be called from the database searching thread. When I do this, the call will be marshaled from the search thread into the control's thread, from which it can be legally made to the container, and the result marshaled back to the search thread. This is the same thing that COM did in the first simple example, just to and from different threads.

Consider the code in the directory sample \ThreadDB, which contains a control called threaddb.ocx and an HTML page called threaddb.htm. If you register the former and open the latter with IE 3.0, you will see the screen shown in Figure 12. The idea is to simulate a search program that takes a Social Security number and returns an IDispatch object representing a customer, having such properties as first and last name. The VBScript listing is shown in Figure 13. You enter a Social Security number and click Start Search, thereby calling that method on the ThreadDB control. For simplicity, I didn't bother reading the Social Security number out of the edit control; I just arbitrarily pass a value of 444. When the operation completes (very quickly in this example), it fires back an event called SearchFinished. This event passes as a parameter an automation object of class CustomerObj, from which VBScript fetches the properties that represent the customer's name and address and places them in the edit controls on the page. The VBScript listing actually lives in the file threaddb.alx, a layout control that I built with Activex Control Pad to get the layout to look nice. Enter anything or nothing into the Social Security number box, click Start Search, and presto, that number has identified a client named John Smith.

The control exposes a single method called StartSearch (see Figure 14), which VBScript calls when you click the Start Search button. I first use the function AfxBeginThread to create a new thread, which pretends to perform a database search. The thread belongs to the class CSearchThread, defined in Figure 15. As before, I create the thread in a suspended state so I can initialize its member variables before it goes charging off into

the database. In addition to its normal state variables, such as the Social Security number of the customer I want to search for, I also need to give it the IDispatch interface pointer that I want it to use to signal its completion. The former task is trivial. The latter is trickier.

Marshaling an interface to another thread is accomplished via two API functions. The thread that owns the legal interface pointer calls CoMarshalInterThreadInterfaceInStream, which creates a memory structure containing all the information that COM needs to set up marshaling into the owning thread. This function provides that information in the form of a Structured Storage stream, represented by an IStream interface pointer. Once you have this stream, you must somehow get it to the destination thread that wants to call the interface owned by the original thread. The destination thread calls the API function CoGetInterfaceAndReleaseStream, passing a pointer to the stream containing the marshaling information. This function returns a pointer to the original interface that the new thread may now call. This is a proxy that actually marshals the call into the original thread rather than a direct connection.

To find the IDispatch pointers used for signaling a control's events to its container, I traced down into the MFC source code to the function COleControl::FireEventV, the base function responsible for firing all events. When it fires an event, it uses the IDispatch pointers that it has stored in a member variable called COleControl::m_xEventConnPt. It shows that more than one container IDispatch might want to be notified of an event. I lifted the relevant portions of the code and bingo, there was a list of all IDispatch pointers that are listening for events from this control.

In the sample program, StartSearch iterates over all the IDispatch pointers from the container and calls CoMarshalInterThreadInterfaceInStream for each of them, receiving an IStream interface pointer in return. I've never seen there be more than one, but it's better to be safe than sorry. The CSearchThread class contains a member variable called m_StreamArray, which represents an array of pointers to IStream interfaces. The control places each of the IStream interface pointers into this array.

The thread that performs the simulated search is an object of class CSearchThread. When the CSearchThread starts up, the MFC framework calls the thread's Initinstance method, as shown in Figure 16. Since the new thread wants to access COM, it must first call CoInitialize. It then wants to get the interface pointers to the IDispatch interfaces that it will use to signal completion of its search. The CSearchThread contains the aforementioned array of streams containing the information it needs to create these. It iterates over the array of IStream interface pointers provided by its creator, calling CoGetInterfaceAndReleaseStream to convert each into an IDispatch pointer. As previously stated, these pointers are proxies, not direct connections.

I then create a CCustomer automation object with the name of John Smith to simulate the result of the search. This is an automation object that I used Class Wizard to derive from CCmdTarget. It contains the properties of FirstName and LastName. Finally, I use each IDispatch pointer to fire the SearchFinished event to each event sink that might be waiting, using the method COleDispatchDriver::InvokeHelper. I lifted the code straight from COleControl::FireEventV.

One thing looks strange here. The CCustomer automation object that the search thread provides as a parameter to the event is a COM object, but I don't seem to be marshaling it even though its properties are being accessed by IE 3.0 on another thread. Have I omitted something, or is this legal? It's legal. COM knows how to marshal method parameters. Because the parameter being passed to the event function is an IDispatch interface pointer and identified as such by the flags in the VARIANT structure that contains it, COM knows what type of marshaling is needed and sets it up automatically. The event function in VBScript actually receives not a direct connection to the CCustomer automation object, but a proxy that is valid in VBScript's thread. When VBScript calls the Invoke method to get the FirstName and LastName properties, the call is marshaled to the search thread and the result marshaled back to VBScript. That's one of the beauties of COM--the dirty details are hidden away. You just follow a few simple and consistent rules for negotiating with COM and everything works more or less the way you think it ought to.

If calls on the CCustomer proxy are marshaled back to the thread that created it, does this mean that the thread has to stay around as long as the object does? Yes, it does. To prove that to yourself, rebuild the control, adding a call to PostQuitMessage at the end of the CSearchThread::InitInstance method, thereby causing the thread to terminate as soon as it returns from signaling the event. When you click Start Search, John Smith's name appears in the edit controls because the code that fills the controls is executed during the event, while the thread is still alive. If you then click the Refresh button, attempting to access the C Customer automation object again, you will see an error box from VBScript announcing an error of type 0x80010012L. If you look that up in the header files, you will find this constant defined as RPC_E_SERVER_DIED_DNE. You can't make a call into an object if the thread that created the object isn't around to service the call.

To solve this problem in the sample control, I overrode the OnFinalRelease method of my CCustomer object. This is called when the last user of the object calls Release and the object's reference count drops to zero. In

it, I call PostQuitMessage, thereby sending the WM_QUIT message to the search thread. In the thread's Exitinstance method, I uninitialize COM via the function CoUninitialize.

How do I know if the interface really needed to be marshaled at all? It might have been a free-threaded interface that wouldn't care if it was called from another thread. I don't know, but fortunately I don't have to know or care. If I simply make the function calls to marshal the interface as shown above, and the interface actually belongs to an object that doesn't require marshaling, COM will simply not set up the marshaling code behind the scenes. Instead of a proxy, CoGetInterfaceAndReleaseStream will return a direct pointer to the object's code. I follow a few simple rules and COM intercedes to the extent necessary. I don't have to know what that extent is in order to use it, and I'd just as soon not have to think about it.

Non-MFC Single-Apartment Threaded Controls

The previous examples have focused on controls implemented with MFC and its base class COleControl, which is where most full-featured controls have historically come from. Today's rush to the Internet is leading developers to such tools as the Activex Template Library and the SDK CBaseCtrl class to keep the size of their code to a minimum. What design considerations are important if you are rolling your own apartment-threaded control instead of using MFC? Most of it turns out to be fairly simple, but there's one thorny problem, which I'll describe after I show you the easy stuff.

You have to place the "Threading Model" data entry into the registry yourself, and that's simple. All of your objects have to serialize their access to your control DLL's global data, as was done in the previous examples. That's also relatively simple, and is the same as for an MFC-based control anyway.

You now have to ensure proper handling of the named global functions that your DLL exports to COM, specifically DllGetClassObject and DllCanUnloadNow. Any thread in the container that calls CoCreateInstance, CoGetClassObject, or their kin will eventually wind up in the former function, so you have to make it thread-safe. This, too, is fairly simple. You can either write a single class factory and make that reentrant, or you can create a new class factory every time DllGetClassObject is called. The example shown in Figure 17 uses the latter.

The function DllCanUnloadNow is where the problem lies. This function gets called when the container calls CoFreeUnusedLibraries. COM is asking each server DLL whether it has objects outstanding and therefore cannot be unloaded. Your function returns S_OK if it can be unloaded because it does not have any objects currently outstanding, or S_FALSE if it cannot be unloaded because it does. Your DLL needs to maintain a global object count to answer this question properly, which is simple enough. You have a global variable as shown in Figure 17. Every time your DLL creates an object, you increment it, and every time an object is destroyed, you decrement it. Make sure you use the thread-safe functions InterlockedIncrement and InterlockedDecrement for manipulating this count, as shown in the implementation of the class factory in Figure 18. The simple C++ statement

```
g_ObjCount++ ;
```

is not thread-safe. Look at the assembler listing of it if you don't believe me.

Consider the listing of DllCanUnloadNow shown in Figure 17. Suppose the thread that called the function got swapped out just before the return statement and another thread got swapped in and called CoGetClassObject (which calls DllGetClassObject) to get a class factory. You've already checked the object count and decided it was zero. When DllCanUnloadNow eventually gets swapped back in and returns S_OK, you would think that the server DLL would be unloaded and the other thread's call to the class factory would reference code that wasn't there any more. Still no problem; COM serializes the calls to these two functions so that one has to return before the other can be called. In this situation, COM will block the thread calling DllGetClassObject until the call to DllCanUnloadNow returns.

So let's look at the thorny problem. Consider the class factory destructor shown in Figure 18. Suppose this was the last object provided by my server and it had just been released by the thread that owned it. The destructor's code calls InterlockedDecrement, the global object count goes to zero, but the return statement hasn't been executed yet. Now suppose another thread gets swapped in and calls CoFreeUnusedLibraries, causing DllCanUnloadNow to be called. This function checks the global object count, finds that it's zero, returns S_OK, and the DLL gets unloaded. The first thread gets swapped back in, tries to execute the return statement, but the code isn't there anymore because the DLL got unloaded. Boom, you crash.

It's difficult for a control to protect itself against this without being badly behaved. Don Box, in his January 1997 column, proposed two solutions. DllCanUnloadNow can always return S_FALSE, so there will never be any conflict, but that's not a nice thing. Or you could set a timer in the object's destructor, and change DllCanUnloadNow to require that both the object count be zero and the timer expire before returning S_OK. This assumes that whatever timeout interval you selected would give the destructor the chance to finish returning. Neither option is fantastic. In practice, the control cannot definitively protect itself against this

situation without violating other rules. It is up to the container to pay attention and only call CoFreeUnusedLibraries in idle moments, when it isn't creating or destroying objects.

The Multithreaded Apartment Model

What about the MTA model (also called the free threading model), which has gotten so much press these last six months or so? A full discussion of it would require another article of this length or more. I'll try to give you an idea in the next few paragraphs of what it means for controls and their containers.

A control that supports the MTA model is saying to COM that it has already internally serialized all of its methods to whatever extent they require, and therefore any method on any of its objects can be called from any thread at any time. Like most things, this approach has advantages and drawbacks. MTA controls have the potential to run faster when being called by threads other than the ones on which they were created. STA controls are easier to write.

Each thread in an app may identify itself as an STA thread or an MTA thread. Single-threaded apartment is the default provided by CoInitialize. A thread that wants to identify itself as MTA must initialize COM by calling the new function CoInitializeEx using the COINIT_MULTITHREADED flag (this flag was not supported prior to the DCOM update for Windows 95). You can mix and match single and multithreaded threads in the same process.

Based on the value of the ThreadingModel registry entry, a control can identify itself as nonthreaded (None or absent), single-threaded (Apartment), multithreaded (Free), or able to provide direct connections in either STA or MTA models (Both). You can use any of them anywhere and COM will intercede to the extent needed to make it work. It's not a matter of getting it to work at all, but of getting it to work optimally.

If a single-threaded apartment thread creates an instance of a control marked only as Free, COM will marshal all calls into or out of that control. In the absence of the Both key, the MTA control has not promised to make all callbacks into the container (for example, firing events), only on the thread that provided the interface, which the single-threaded apartment model requires. The performance advantages in IE 3.0 and IE 4.0 obtained by a direct connection apply only to controls that support STA or both models, not to those controls that support the MTA model alone.

If a multithreaded apartment thread creates a control marked as Free, COM will not set up marshaling. Methods on these controls may be freely called by any thread in the process without marshaling. An MTA-aware control has promised COM that it contains whatever serialization code it requires. It's saying, "Go ahead, do anything to me from anywhere. I'm tough! I can take it!"

If an MTA thread creates an STA control, COM will set up a marshaling proxy. The interface pointer can be called from anywhere, but the call will be marshaled back into the thread that originally created the control because COM knows that's what an STA control requires. The one place where you make a performance profit with the MTA model that isn't available in the STA model is when you have an MTA control in an MTA thread and call the control's methods from another thread. In the STA model, the call from the second thread would have to be marshaled, with attendant performance penalties. In the MTA model, it proceeds directly.

Which model should your control support? It basically depends on who your customers are going to be. If you're writing controls that you intend to place on HTML pages for the use of browsers, these are multithreaded apps that always initialize their threads as STA, and will continue to be for the foreseeable future. The MTA model won't buy you anything here because the container doesn't know how to take advantage of it. Multithreaded apartments are used most in DCOM scenarios. If your components will be serving incoming DCOM calls, you can achieve significant performance and scalability by making them fully thread-safe and marking them with Both.

If your control supports the STA model, COM provides you with object-level serialization for no development effort on your part, and you will still have a direct connection to STA threads. If most of your object's methods need this serialization, you might as well write an STA control and get it out the door quickly and bug free. For example, if your control is doing lots of user interface programming, such as in-place activation required of full-featured controls, you would have to write code to serialize most of your control's methods anyway, so why not take what COM gives you?

The place where the fully thread-safe model makes you a profit is if your control has methods that don't need very much serialization and if the container wants to access the same control from multiple MTA threads. Consider the IDataObject interface, an interface so simple that I use it for the very first COM example in my book. It contains nine member functions in addition to IUnknown. Most of this interface's methods are simple enough that they don't require any type of serialization; they can be made completely reentrant. For a one-way data transfer object of the type used for dragging and dropping text from one window to another, six of them are usually stubbed out, simply returning a hardwired error code saying, in effect, "No, I don't do this." There's no reason that these couldn't be called from any thread at any time, thereby saving the effort of

marshaling these calls. If the data object represents a static snapshot, as is usually the case, the other three methods don't need serialization either. If they did, you could write it yourself, taking the performance hit on only three methods instead of all nine. Of course, this only works if the container supports MTA controls.

To summarize, full-featured controls to be used by browsers want the STA model. Fortunately, that's the easier of the two to write. If you are going to write an MTA control, consider supporting both threading models, or you'll take a performance hit in an STA-threaded container.

Miscellaneous Gotchas Here, in no particular order, are miscellaneous tips from the trenches in writing STA controls. It probably won't make much sense the first time, but after you've worked with this stuff for a while it'll start to gel.

When you are testing your controls, make sure that you do so with several different pages, each containing several instances of your control. The interaction effects tend to bring to the surface bugs that are not easily found by other methods. Use the debugging versions of MFC and the operating system to check for leaks, which is the favorite kind of bug for a control.

If your control is exposing any kind of custom interface, you will need to supply a marshaler for it. You are used to thinking of in-proc interfaces as not requiring marshaling, and in the case of a single-threaded container this is true. Once you are in a multithreaded container, interface pointers need to get marshaled between threads all the time, and this won't happen without a marshaler. All of the standard interfaces already have these marshalers in the operating system, but your custom interfaces will not. Writing an interface marshaler is quite easy using MIDL.

In an STA control, maintaining per-object storage is trivial as the member variables of your control are thread-safe in this model. Access to global variables (per-DLL storage) needs to be serialized, but that's not a big problem. A somewhat more intricate problem is knowing when to allocate and release objects that you have one of per thread. For example, since GDI does not serialize access to its objects such as fonts or brushes, it is sometimes advantageous to keep one of these per thread. Or perhaps you created a hidden window (the ultimate thread-centric object) to use for communication between your controls. You could keep one per object, but that would be wasteful, particularly in Windows 95. "No problem," you say, "I'll just use the DLL_THREAD_ATTACH and DLL_THREAD_ DETACH notifications that I get in my control DLL." Not really. You don't get these when you need them.

Suppose you have the IE 3.0 main window showing a page that doesn't contain your control. You open another IE 3.0 window and surf to a page that does contain the control. Your control's DLL is loaded, you get the DLL_PROCESS_ATTACH notification (which implies the first thread notification as well), and you allocate a brush for that thread. No problem. Now suppose the user goes back to the first window and opens another page containing your control. Another instance of your control will be created on the first window's thread, but you won't get a DLL_ PROCESS_ATTACH notification because the thread already existed prior to your control's DLL being loaded.

The place to put your new thread-detection code is in DllGetClassObject. Any thread that wants to create an object will have to go there to get the class factory pointer. You can call GetCurrentThreadId to find the thread on which you are being called and go from there.

You have a similar problem in knowing when to release your per-thread data. You might have all your controls on a thread be destroyed without the thread going down--the user just surfs to another page. So each of your objects that allocate per-thread data will have to check in its destructor to see if it was the last survivor on that thread, and release the per-thread data if so.

Conclusion

You can make your ActiveX controls run a whole lot faster in multithreaded containers such as IE 3.0 and IE 4.0 if you use the single-threaded apartment model. It's not hard to do this; simply rebuild under MFC 4.2 or later and guard access to your global data variables with critical sections. Remember, in the single-threaded apartment model you may only call an object's method from the thread on which the object was created unless you do some serious negotiation with COM.

If you want to find out more about the threading models available in COM, the best place to start is with the Microsoft KnowledgeBase article, "Descriptions and Workings of OLE Threading Models," number Q150777.

[TABULAR DATA OMITTED]

THE QUICK ANSWER

If you don't feel like reading this entire article to learn why, the how is extremely simple. To obtain the performance benefits listed in this article, simply do the following with your Activex controls:

* If you're building new controls, generate their projects using MFC 4.2 or later. If you have existing MFC controls from earlier versions, find the call to AfxOleRegisterClass and change its sixth parameter from FALSE to afxRegApartmentThreading. This will make your control use the apartment threading model.

* Serialize access to your global variables and class statics with critical sections. In the apartment model, different objects may try to access these from different threads. Don't worry about the global object count; MFC takes care of it. Don't worry about per-object data, such as member variables. COM promises that member functions of individual objects will only be called from the thread on which that object was created, so serialization is not necessary.

* If your control creates auxiliary threads to help it carry out its functions, or if you want to pass COM objects to or access COM objects from other threads, you need to read the entire article.

David Platt is president and founder of Rolling Thunder Computing (www.rollthunder.com). He is also the author of The Essence of OLE with Activex (Prentice Hall, 1996). David can be reached at dplatt@rollthunder.com.

illustration chart program

Document msjn000020011007dt210000a

