Fashionable app designers agree: the free threading model is what's hot this fall. (Technology Tutorial)(Tutorial)

David Platt
7,779 words
1 August 1997
Microsoft Systems Journal
MSJN
33
Vol. 12, No. 8, ISSN: 0889-9932
English

Using COM in multithreaded applications is a topic fraught with fear, uncertainty, and doubt. If you are writing a COM object, do you have to worry about serializing access to the object's member functions and data? What about to the object server's global functions and data? And what about writing a client app that uses COM objects--an the app access the same object from two threads? Can it access two different objects from the same server from two different threads? What if an object's behavior changes from one version to another?

Don't worry. As with most topics in COM, the reality isn't nearly as painful as the anticipation. If you follow a few simple rules, COM will take care of the gory details.

Requiring a client to have intimate knowledge of the internals of the objects it wants to use, or having an object need to care about the internals of its clients, would violate the most fundamental principle in all of COM. The C in COM stands for component--if a client needs to care about the internals of an object, then the object can't really be called a component. COM supports threading in such a manner that neither side needs to know or care about what the other is doing. It works according to the following principles:

* The client app tells COM the rules that it intends to follow For using threads when it initializes COM via the function CoInitializeEx, the replacement function for CoInitialize.

* The object's server tells COM the rules that it intends to follow for using threads, either when it calls CoInitializeEx for an EXE server or by making registry entries for a DLL server.

When a client creates an object, COM compares the threading rules that each party has said that it follows. If the parties have promised to play by the same rules, then COM sets up a direct connection between the two and gets out of the way. If they are following different rules, then COM sets up marshaling between the two parties so that each sees only the threading rules that it has said it knows how to handle. The latter case has some performance costs but allows parties following dissimilar threading rules to work together without croaking or needing a special case.

Threading Model Choices

I know more potential COM programmers that have been turned off by the buzzwords "threading model," "apartment threading," and "free threading" than any other terms in all of Windows. They aren't that bad, although it doesn't help that the nomenclature keeps changing.

A threading model is a set of rules describing the interaction of threading and COM that an object or a client Follows. My previous article, "Give ActiveX[TM]-based Web Pages a Boost with the Apartment Threading Model," (MSJ, February 1997), explained the apartment threading model, also known as the single-threaded apartment model, one set of rules that can be followed. This article picks up where the February 1997 article left off, so if you are new to the issues of threading under COM, I strongly suggest that you begin by reading that one. It is available on the April Microsoft Developer Network CD-ROM, or you can order back issues of MSJ via their Web site (http://www.microsoft.com/msj).

In the apartment model, the main rule was that a client app could call an object's methods only from the thread on which the object was created. This meant that different objects from the same server could be called from different threads, but that each object would be called from only one thread. This in turn meant that objects needed to serialize their access to their servers' global variables and functions, but not to their own instance data. Under the apartment model, objects are relatively easy to write, but clients are potentially

tricky. If an apartment threaded client needs to call an object's method from a different thread, it can do so, but it has to jump through some hoops and negotiate with the operating system first.

This article describes the free threading model, also known as the multithreaded apartment model, the other set of rules that an object or server may choose to follow. In the free threading model, a client app may call any object method or COM function from any thread at any time. It is up to the object to serialize access to all of its methods to whatever extent it requires to keep incoming calls from conflicting. It provides the maximum in performance and flexibility. The cost is that the objects themselves get harder to write compared to apartment model objects, although free threaded client apps are easier than apartment threaded client apps.

Where does using the free threaded model make you a profit over the apartment model? Any place that an object needs to be accessed from more than one thread. Suppose you have a client app connected via DCOM to an object on a remote machine. When the remote client calls a method on that object, the server receives that call on an existing thread from a pool that exists for just such a purpose. This receiving thread makes the call locally to the actual object. If the object supports the free threading model, then the call into the object can be made directly from the receiving thread. If the object supports the apartment threading model, then the call will have to be transferred to the thread on which the object was originally created and the result transferred back into the receiving thread before being returned to the client. You incur extra marshaling overhead that you don't need. You also run the risk of the object's thread being busy servicing other objects or-- worse--blocked while waiting on some synchronization event. Finally, you are unable to take advantage of the case where the server has multiple CPU chips, which is becoming more and more common.

For another example, suppose that you have a single object that represents a connection to a database, or perhaps to a piece of hardware. The sample programs supplied with this article simulate the latter case. Suppose you have four or five threads that need to access that object. In the apartment threading model, the object could reside on, at most, one of those threads. The other threads would have to set up interthread marshaling proxies to access the object from their own threads. You would incur the penalties of the code needed to set up the marshaling initially and the marshaling overhead every time you made a call. If both object and client support the free threading model, then the multiple threads could all access the connection object directly, without any marshaling. The object itself would have to serialize its own calls to the extent they require, and you would incur this overhead, but the object programmer's intimate knowledge of its behavior allows him to optimize this case. Frequently, many of the calls can be made reentrant and won't need serialization at all. The ones you do have to serialize can frequently be done much more cheaply than COM's generic marshaler.

Free Threaded Client Apps

Free threaded client apps are easier to write than apartment clients. An apartment threaded client has to follow some restrictive rules about which threads are allowed to access which objects. All a free threaded container needs to do is make one call to the API function CoInitializeEx, passing a value of COINIT_MULTITHREADED as its second parameter as shown in Figure 1. Unlike the apartment model, each individual thread does not need to call CoInitializeEx; one call takes care of all threads in the process. This initializes COM in the client app's address space and tells it that this app supports the free threading model. Once it does this, the client app may then call any COM function or object method from any thread at anytime.

When the client app creates an object, COM detects via registry entries any object that cannot handle being called in this manner. For these objects, COM transparently sets up marshaling to serialize access to their methods in the way that the object is expecting. Your client app does not have to think about the threading model used by any of the objects that it creates. Your app tells COM which rules it is following and then follows them. COM checks the rules that the object says it follows and transparently intercedes between client and object to the extent, if any, needed to make them work together properly.

There are two gotchas here, both related to the fact that the free threading model is a recent arrival. First, you must define the constant_WIN32_DCOM in your project. In the samples supplied with this article, I have done it in the project settings to make sure it's applied everywhere. The system header files that come with Visual C++[R] 5.0 omit the DCOM-related functions, such as CoInitializeEx, unless this flag is defined.

The second gotcha is that the free threading model is present in Windows NT[R] 4.0 and later, but it is not supported in earlier versions of Windows NT, or in Windows[R] 95 unless the DCOM extensions have been installed. The DCOM extensions are installed automatically with Internet Explorer 4.0 and are also available separately from the Microsoft Web site. A client app that depends on the free threading model must check at startup time whether the operating system on which it is running supports this model. Fortunately, this is easy to do; just check for the presence of the function CoInitializeEx in the operating system DLL ole32.dll. As shown in Figure 1, simply call the API functions GetModuleHandle and GetProcAddress. If CoInitializeEx is

present, then the free threading model is supported on the user's current operating system; otherwise your app needs to back out more gracefully than this simple example.

Sample Programs

Before I dive into the sample programs, I strongly suggest that you download the code and work along with me as you read the article. The text in this article refers to the sample application in the file freethread.zip, which is available from the MSJWeb site (http://www.microsoft.com/ msj). When you unpack this, you will find a directory, \freethread, containing a client app in the subdirectory \client, an in-proc server in the subdirectory \inprocsv, and a local server in the directory \localsv.

Before running the client app, you must register the servers. To do this, move the \freethread directory to the root directory of your C: drive and double-click on the registration file C:\freethread\freethread.reg. If you do not want the sample directory in this location, you can place it anywhere you want, but you will then have to edit the file freethread.reg to change all of its LocalServer32 and InprocServer32 entries so they point to the locations where you actually place these files.

An in-proc server that supports the free threading model identifies itself to COM by adding the named value ThreadingModel to its InProcServer32 registry key and setting its value to Free, as shown in Figure 2. The supplied registration file makes entries for four separate class IDs, each identifying itself as supporting one of the allowed threading models (none, apartment, free, or both). The InProcServer32 entry of each of these points to the same server DLL. Obviously, the code executed for each class ID is the same for each case, but the threading model entry will cause COM to treat them differently with regard to threading. The sample client can then use objects created under the different class IDs to demonstrate the differences in COM's treatment of the threading models.

The in-proc server consists of a control called data6ips.dll. This is a control in the most general sense of the word: an in-proc server that supports only the IDataObj ect interface.

Imagine that I am the manufacturer of a radio receiver computer board that listens to the National Bureau of Standards time signal, radio station WWWV. (It's the perfect gift for the total geek on your Christmas list.) I need to provide a software mechanism for apps to access the current cesium clock time received by the board. Rather than a straight DLL, I have chosen to provide it via a COM in-proc server. The actual time is provided via the method IDataObject::GetData in a private format called TimeDataObject, which consists of a SYSTEMTIME structure transferred via an HGLOBAL. More important for this demonstration is the fact that this object supports another format called ThreadId, which simply returns the thread ID on which the call is received by the object. This allows me to compare the ID of the thread on which a call was made with that of the thread on which the object receives the call. The code for this method is shown in Figure 3.

The local server consists of an app named data61osv.exe. It provides the same objects to a client that the in-proc server does. It is provided to demonstrate the concepts and special concerns that apply only to EXE servers and not to DLLs.

In order to demonstrate multithreading concepts, these sample servers do not create a separate object for each client that requests one. Rather, they create a new object to give to the first requesting client and provide all subsequent clients with a reference to the existing object. This might not be the best way to write a real-world app; I do it here for demonstration purposes. However, some pieces of COM exist expressly for this purpose, such as the running object table. It is not a forbidden architecture.

The client app is a multithreaded MDI app named data6cl.exe. The main frame window is handled by the app's main thread. It contains three MDI child windows, each of which is handled by its own separate thread. The exact operation of this app is described in the following section.

Free Threaded In-Proc Servers

To begin the demonstration of the free threading model, run the sample client app data6cl.exe. It will look like Figure 4: a main MDI frame window, two child windows tiled horizontally and labeled Free, and one minimized window labeled Apartment. The latter is present to demonstrate a point in a later section of this article; ignore it for now. Each window is created and controlled by a separate thread. The thread ID of the thread controlling the window is shown in that window's title bar. In Figure 4, the main window is controlled by thread 0x67, which is the app's main thread, the top child window by thread 0x41, and the bottom child window by thread 0xb6. When the client app first started up, the main thread called CoInitializeEx to initialize COM using the free threading model, as shown in Figure 1.

Click either of the tiled child windows, then choose Data Object--Create Apartment-Threaded In-Proc from the main menu. The WM_COMMAND message is received by the main window on the main thread, which contains code that posts it to the active child window. The message is thus processed by the window procedure of the child window on the thread that owns the child window. When you make this menu selection,

the active child window calls CoCreateInstance to create an object of the specified class, storing its pointer in a global variable called pFreeObj where all threads can access it. It then invalidates all the child windows, forcing them to repaint themselves.

When it receives the WM_PAINT message, each free child window calls IDataObject::GetData on pFreeObj, as shown in Figure 5. The child window is making this call from its own thread, whose ID is in its title bar. The child windows request the data in the format cfrhreadId. As shown in Figure 3, this causes the object to report the ID of the thread on which it receives the call. The child windows display this ID for you to read, as shown in Figure 6. The transfer time shown on your screen will be zero at this point; it will not change until you perform the time test (which I'll get to shortly).

In this example, both free threaded windows report the same thread ID, which is different from the ID of the thread that owns the window and from which the call was originally made. COM detected from the server's registry entries that the object supports only the apartment threading model, while COM knows from the container's initial call to CoInitializeEx that the container is using the free threading model. COM therefore creates a new thread, creates the object on that thread, and returns from the creation function not a direct pointer to the object, but rather a pointer to a marshaling proxy that marshals all calls onto the object's thread. In this manner, the apartment threaded object sees only the threading behavior that it has told COM it knows how to handle. This is similar to the case in my previous article where an apartment container created a nonthreaded control. COM is interceding between client and object to reconcile the differences between the threading models. Neither client nor object knows or especially cares about the threading model used by the other.

It works seamlessly, but there's a price. Pick Data ObjectIGetData Time Test from the menu. This performs a timing test by repeatedly calling the method IDataObject::GetData, asking it to transfer a specified amount of data (see Figure 3). The elapsed time is shown on the thread's child window. The numbers in Figure 6 were obtained on a Pentium Pro 200 CPU, transferring 10KB on each of 10,000 repetitions. The total elapsed time was 1,533 milliseconds, or about 0.15 milliseconds per call. This overhead results as the call is marshaled from the thread on which it is made to the thread on which the apartment object lives and the results are marshaled back.

Suppose I don't want my customers to incur this overhead; I'm afraid they'll buy someone else's product. I want to write an object that provides a direct connection in the free threaded model without any of this marshaling overhead. Choose Data Object | Release( ) from the menu, then Data Object ICreate Free-Threaded In-Proc. This calls CoCreateInstance, specifying the class ID of the object that has identified itself as supporting this model by placing the Free key under Threading Model in the InProcServer32 key in the registry, as shown in Figure 2. The client app will then appear similar to Figure 7. Note that each free threaded window now reports that the object is receiving calls on a different thread ID, which is the same thread ID as the one that owns the window and from which the call was made. Even though the call to CoCreateInstance was made from one thread and not from the other, the pointer returned from CoCreateInstance is to a direct connection to the object. Each thread can now call its methods directly. Perform the time test again and you'll find that the transfer time has gone down to 70 milliseconds, speeding up by a factor of about 22.

All right, that's cool, but what about the case where the two threads might call the time test simultaneously? You can try this yourself by bumping up the number of repetitions in the timing test to a million or more so that you can do it manually. What about now? In an apartment threaded object, COM would automatically serialize calls to each object's methods, so I didn't have to worry. That isn't happening here. COM detected that both client and object are using the free threading model, so it got out of the way. This means the object has to serialize its own methods to whatever extent the methods need. It is up to the programmer to determine what that extent is and to write the code that makes it happen.

Consider my object's implementation of IUnknown, shown in Figure 8. In the apartment model I used InterlockedIncrement and InterlockedDecrement in the object's constructor and destructor because these functions changed my server's global object count and I had to serialize access to that in the apartment threading model. Now I need to serialize access to my object's own internal reference count, otherwise one thread could get swapped out in the middle of AddRef or Release and another thread could get swapped in and call the same method, resulting in an incorrect reference count. I use the functions InterlockedIncrement and InterlockedDecrement here to make the operation thread-safe. So even though I had to write this code, I've still made a performance profit because my serialization code has much lower overhead than the interthread marshaling that COM would have provided had I written this control to use the apartment model.

My QueryInterface method, also shown in Figure 8, doesn't need to change at all. It is completely reentrant; everything is done on the stack. This illustrates another place where the free threading model makes you a performance profit-- your objects serialize only the calls that need it. Since it had no way of telling which calls needed serialization, the apartment model serialized all calls to all interfaces on its objects, whether they needed it or not. Another example is the method IDataObject::GetData, shown in Figure 3. This method

accesses two global variables, cfTimeData and cfThreadId. These are UINTs that denote private Clipboard formats. Their values are set in DllMain when the server's DLL is first loaded, and are never changed. I don't have to serialize access to them, and it doesn't matter if they are read by multiple threads simultaneously. The rest of the work of this method is done with stack parameters, which are thread-safe. So this method does not require any type of serialization either--I've made a profit with my intimate knowledge of this specific app.

OK, you've dodged all that, but when do you have to do some actual serialization? The answer for this example is in the methods IDataObject::DAdvise and DUnadvise, shown in Figure 9. These methods are used to establish a callback circuit so the object can advise the client when its data changes. The client provides an object that supports the IAdviseSink interface, and the object calls the method IAdviseSink::OnDataChange to notify the client when the data changes. The actual operation of this callback architecture is discussed later in the article, in the section on servers that support both threading models.

As written, my object only supports one connection at a time. A request for a second connection will fail until the first one is released. Since these methods access internal member variables, I need to write code to make sure that only one thread uses them at a time. I do this with a critical section that I set up as a member variable of my object's class. Each method calls EnterCriticalSection at its beginning and LeaveCriticalSection at its end. If one thread gets swapped out in the middle of one of these methods and another thread gets swapped in and calls one of them, the second thread will block until the first one finishes.

You may wonder why I am working directly with the IAdviseSink interface instead of following the standard practice of delegating these operations to a data advise holder, an object created by the API function CreateDataAdviseHolder. I don't do that in this example because IDataAdviseHolder, even though it is a standard interface, does not seem to have a marshaler in my operating system (Windows NT 4.0 Service Pack 2). This means that it can be used in a free threaded object, but not in one that supports the apartment model or both models. I could write and install a marshaler for it, which would take under an hour, but I thought it was better to show how to run with the OS as shipped.

Free Threaded Local Servers

A free threaded object provided by an EXE server must serialize access to its methods and data in the same manner as an object provided by a DLL server. However, there are important differences between the two types of servers in the way they tell COM which threading model they support, and also in their handling of class factories.

An EXE server does not tell COM which threading model it supports by making registry entries, as does a DLL server. Instead, it signals this to COM as does a client app, by calling CoInitializeEx, passing the COINIT_MULTITHREADED flag to support the free threading model. When the server does this, it is saying to COM that any objects it provides to COM, including class factories, may be called from any thread; that the objects have already been internally serialized to whatever extent they require.

In the sample code provided with this article, I use the same local server app to demonstrate both the free and apartment threading models. I have registered two different class IDs, one for each model. In the LocalServer32 entry of the apartment model server class ID, I have appended the string -Apt to the server's command line. When the server's WinMain function starts running, the first thing it does is check its command line for this flag. If present, the server initializes itself to use the apartment threading model; otherwise it initializes itself to use the free threading model. I did this for the convenience of my not having to write two server apps. It is not a standard part of COM, as is the -Embedding command-line switch that tells a server app that it was launched by COM. Although it's nonstandard, you can certainly use this technique if you want to have a switch-hitting local server. This is shown in Figure 10. When I ran the previous time test on the local server using each threading model, the free threaded server took about 3,700 milliseconds while the apartment threaded server took 4,500 milliseconds.

COM reaches into a DLL server via the function DllGetClassObject and pulls out a class factory whenever it needs one. A free threaded DLL server handles this situation no differently than in the apartment threaded case; it needed to be thread-safe there as well. In an EXE, it is up to your code to register class factories when it first starts up, maintain a global object count, and revoke the class factories and shut itself down when the object count reaches zero. This causes a multithreaded EXE server to have timing and concurrency problems that a DLL server doesn't. Fortunately, COM now contains auxiliary functions that make it easy to handle these situations, once you know where they are.

The first timing problem comes when the free threaded EXE server registers its class factory. A single-threaded server could call CoRegisterClassObject at whatever point in its initialization it felt like doing so. The server didn't have to worry about receiving incoming calls to IClassFactory::CreateInstance until it checked its message loop, since that was necessary for receiving the call. A free threaded server doesn't have this luxury. At any instant after it registers its class factory, it might receive calls from external threads to

create objects. An object might get created and try to run before the server app completed its own initialization. The object wouldn't have the necessary infrastructure to do its work--for example, the object might need to talk to a window on your server, but the window might not have been created. You could make registering the class factory your last piece of initialization, but that would needlessly tie your hands. And what about the fairly common case of an EXE server registering more than one class factory? At best, you would have to do a complicated timing dance to make sure everything happened in the right order. There has to be a better way.

This is an easy problem to solve. You create and register the class factory as before, but this time you add the flag REGCLS_SUSPENDED to the fourth parameter of CoRegisterClassObject. This registers the class factory with COM, but leaves it in a suspended state in which the class factory will not accept incoming calls. When your server has finished its initialization, it tells COM that its class factories are open for business by calling the new API function CoResumeClassObjects. This will allow all of your registered class factories to accept incoming calls. This code is shown in Figure 10.

The next problem comes when your local server is ready to shut down. Again, the problem is one of thread-safe timing. When the last object on an EXE server is destroyed, it usually starts the sequence of events that leads to the server shutdown, calling CoRevokeClassObject and posting itself a WM_QUIT message. In the single-threaded case, you never had to worry about a call to CreateInstance coming in during this shutdown because you wouldn't service the message loop until you had revoked the class factory. With the addition of multiple threads, this strategy doesn't work any more. What happens if a call comes into the class factory on another thread just after decrementing the server's global object count to zero and starting the revoke and shutdown code on one thread, but before the call to CoRevokeClassObject is made? It would create a new object on a server that was in the process of shutting itself down. At best, you'd have to write code for aborting the shutdown; at worst, you'd croak.

COM has added two new API functions for solving this problem, CoAddRefServerProcess and CoReleaseServerProcess. You call the former in the constructor of every object that you create (other than the class factory itself), and also when your IClassFactory::LockServer method is called with a value of TRUE. It increments a global perprocess reference count maintained by COM. You call the latter in the destructor of every object you create, and also when IClassFactory::LockServer is called with the value of FALSE. This decrements COM's global reference count on your process. The important addition is that, when this reference count reaches zero, COM immediately suspends all activity on all of your server app's registered class factories by internally calling the API function CoSuspendClass0bjects. When CoReleaseServerProcess returns zero, you know that the object was the last one in your process and your server can safely initiate the shutdown--your class factories are already suspended, and no other app has a current pointer to any of your objects. Any client app attempting to create an object of the class manufactured by your server will cause COM to launch another instance of it. These functions are thread-safe; there is no window of vulnerability. This code is shown in Figure 11.

Apartment Threads in Free Containers

What happens if there is an apartment threaded object that your free threaded client app just can't live without? Your client can certainly use the apartment threaded object and incur the interthread marshaling penalty, but suppose that's too restrictive--you need the last microsecond of performance from the object, but you can't get its vendor to rewrite it for the free threading model. Is there a way around this?

Yes, there is. COM deals with threading issues on a perthread basis. You can create apartment threads within a free threaded container (or vice versa, for that matter). All a thread needs to do is call CoInitializeEx when it starts up, passing the flag COINIT_APARTMENTTHREADED. COM will know that this thread uses the apartment model for all of its COM operations, even if the other threads in the app all use the free threading model. This thread must follow the rules of an apartment threaded container as discussed in my previous article, which include having a message loop, servicing it frequently, and calling its objects only from within the thread that created them.

The sample app shows how this is done. Go to the client app and restore the minimized window that contains the word Apartment in its title bar. This window is controlled by a thread within the client app that has initialized COM with the apartment threading model, as described in the previous paragraph. As with the other windows, the thread ID of the thread controlling the apartment window is displayed in its title bar. The code for this thread is shown in Figure 12.

Click on the Apartment window, then use the app's main menu to create an apartment threaded object. You will find that the thread ID reported by the object is the same as that of the thread controlling the window. This was the case when a free threaded window created a free threaded object. Now, release the apartment control from the apartment window and create a free threaded object in that window. The thread IDs are now different. COM set up marshaling between client thread and object because it knew that they used different threading models.

Both Servers-Disciplined Callbacks

As you saw in the previous section and in my apartment threading article, it isn't always a good idea for a control to use the free threading model. A free threaded control actually loses performance when used by an apartment container because COM insists on marshaling all calls into and out of the control. Why does COM do this when the control has stated that it can be called from any thread at any time, and that it has already serialized its methods to whatever extent they require?

The reason has to do with callbacks from the control to its container. It is a relatively rare case when a control is purely an object and its container purely a client. It's much more common for each side to call methods on interfaces provided by the other. For example, a container calls methods on a control by calling the IDispatch::Invoke function on an interface provided by the control. When a control fires an event to its container, it is calling IDispatch::Invoke on an interface provided by its container. Each side is the server of one interface and the client of another. While a free threaded control has indicated its willingness to accept method calls from any thread, an apartment container is expecting its callback methods to be called only from within the thread that created them. By marking itself as Free, the control is indicating that it doesn't have the discipline to do this, saying in effect, "You can call me from any thread at any time, but I reserve the right to call you back from any thread at any time." Since the apartment threaded container can't handle that, COM creates the free control in a separate multithreaded apartment and marshals all calls into and out of it.

Is there a way in which a control can avoid any marshaling and provide direct connections to all clients? Yes, by marking itself as supporting both threading models by setting the ThreadingModel registry value to Both. A control that supports this model is saying that it will accept calls to any method on any object from any thread at any time, and also that the control will ensure that any call-backs it might make into its container are made only on the thread on which the control received the callback object. The control is promising to be both robust and civilized. In this case, when the container creates a control, it will always be given a direct connection to the control, whether the container uses the apartment or the free threading model.

Supporting both threading models has a reputation for being difficult. Like most such reputations in COM, it's way overrated. If you look through your registry, you will find that servers such as the VBScript engine, class ID {B54F3741-5B07-11CF-A4B0-00AA004A55E8}, support both threading models, although you'll also find that they are currently in the minority. This is because most controls have been built with MFC, which supports only the apartment threading model, and not because supporting both threading models is especially difficult, as I'll demonstrate. It's just that support for both threading models wasn't prefabricated, and free threaded container apps that would care about it were (and still are) pretty rare, so hardly any vendors thought the effort was cost-effective. When you see how relatively easy it is and learn that the Active Template Library (ATL) does, in fact, provide prefabricated support for both threading models, I hope you will consider supporting them in the future.

Suppose you are writing a control and you want to support both threading models. What do you have to do? Essentially, you have to go through the motions of setting up marshaling so that every callback object is called on the thread from which it came, whether the callback object needs it or not. COM will detect and ignore the ones that don't need it, so this doesn't cost you much. Let me show you what I mean.

Run the client app, select a free thread window, and create a free threaded object. Then pick Data Object I DAdvise from the main menu. This calls the object's method IDataObject::DAdvise, passing it a pointer to an IAdviseSink interface supplied by the client app. Whenever the data changes, hardwired to once per second in this simple example, the object will call the method IAdviseSink::OnDataChange (see Figure 13). In the sample app, the object accomplishes this by creating a new thread and having it sleep for a second, then making the callback when it wakes up. The thread in this example is hardwired to always send data in the format cfThreadId, thereby reporting to the client the ID of the thread from which the callback was made. The client then calls GetCurrentThreadId on its own, gets the ID of the thread on which the callback was received, and displays them both to the user, as shown in Figures14 and l5. Note that a free threaded client receives callbacks on the same thread from which the object made the call, while an apartment thread always receives callbacks on the thread on which the object was created.

Go back to the code in Figure 9, and look at the code that the comment block says to ignore. The client is passing the object a pointer, the aforementioned IAdviseSink interface. I want to create a new thread to handle the callbacks, to make the actual call to IAdviseSink::OnDataChange. If my object was to simply implement the free threading model, I could pass this pointer directly to my thread and have my thread make the call directly, without further fuss. However, since my object wants to support both threading models, I need to consider the case where the client uses the apartment threading model and, therefore, cannot accept a callback from any thread other than the one on which the IAdviseSink pointer is originally passed.

In the apartment threading model, you can make a call to an object from a different thread if you set up marshaling. That's what I have to do here. The descriptively named API function CoMarshalInterThreadInterfaceInStream takes the information necessary to set up marshaling for this

interface and writes it into a stream. I then pass that stream to the new thread. In the new thread's procedure, as shown in Figure 16, I take the stream and call CoGetInterfaceAndReleaseStream. This reads the marshaling information from the stream and returns an interface pointer that is legal for the new thread to call. If the original callback interface came from an apartment thread, which COM knows, the interface pointer is not a direct connection, but rather a proxy. When my new thread calls a method on this interface, the call will be marshaled back to the original thread, made from where the apartment container is expecting it, and the result marshaled back to the caller.

What if the client uses the free threading model? Do I incur all the marshaling overhead every time I make the callback, even though it isn't necessary? Fortunately, no. My object doesn't know or care about the threading model used by the client. COM, however, knows the threading models used by both parties. If the client that supplied the callback object uses the free threading model, COM detects that. The final result of making the marshaling function calls is not a marshaling proxy, but rather a direct connection. You make the calls; COM figures out if you can have a direct connection or a marshaling proxy and gives you back the right one. So you see, it isn't really hard to support both threading models, just a little tedious. You have to make two relatively simple function calls, one on each side of the thread. COM figures out the rest, and the right thing happens. Cool.

Support in MFC and ATL

The MFC does not currently support the free threading model, and I have heard no plans or even rumors that it might one day do so. This isn't especially surprising since such support would be foreign to the MFC gestalt. The MFC is thread-safe at a class level; in other words, if one thread accesses a CCmdTarget and another thread accesses a different CCmdTarget, the two threads won't conflict over any global data. However, MFC is not thread-safe at an individual object level. If two threads try to access the same CCmdTarget, they could conceivably step on each other's toes. The architects of MFC decided that providing object-level serialization would slow down everyone while benefiting only a few, and so have left that as an exercise for the student. This implementation corresponds well with the apartment threading model which MFC does support, but making it support the free threading model would take such a large and broad-based effort, and incur such performance penalties, that I don't expect it will ever happen.

Does that mean that I'm back to square one, needing to write all of my code straight through the API if I want to support the free threading model? Fortunately not, or I'd be cutting and pasting till doomsday. ATL 2.1, which shipped with Visual C++ 5.0, contains support for all threading models. When you generate a new ATL project, you are asked for your choice (see Figure 17). ATL has already serialized itself internally, such as in the code shown in Figure 18. This serialization applies only to the ATL code itself, not to any code that you might add. For example, ira call to a single ATL object's method comes in on two different threads, the ATL code won't have any problem, but it is up to you to serialize your own code to whatever extent it requires.

Miscellaneous Gotchas

Here, in no particular order, are miscellaneous tips from the trenches of writing free threaded 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.

An object should not keep a critical section locked from one method call to another. Suppose a thread calls ISomeInterface::StartSomething, which enters a critical section, but doesn't leave it before the method returns. The object plans on leaving the critical section when the thread calls ISomeInterface::FinishSomething. If the thread dies before it can do so, or simply forgets, then every thread that calls ISomeInterface::StartSomething will block and hang forever. To be a good component, an object should not depend on its client calling its methods in any particular order. Don't make your objects do this without a powerful reason and, if you do, document it in very large letters.

Conversely, a client app can never be sure when an object method call is going to block and, if so, when the block is going to clear. This also applies to apartment model calls. You are used to thinking of your method calls as synchronous, and they are in the sense that they don't return until they complete. However, in a multithreaded environment, your client never knows how the object is organized internally. Some methods are serialized and some aren't; those that are may block at some times and not at others. Because you can't really know when or whether this will happen, it's a good idea to try to avoid making calls to objects from within critical sections. You might want to consider making your object calls from auxiliary threads so that if an object blocks for a significant period of time, at least your main UI thread won't be hung.

Conclusion

The free threading model offers the utmost in speed and flexibility when using COM in a multithreaded environment. This mode] requires a COM programmer to serialize access to all of its methods to whatever extent they require, which often doesn't take much effort (sometimes none). If you are writing an in-proc

object, you should consider writing it to support direct connections in both the apartment and free threading models, which isn't very hard either. The MFC does not provide prefabricated support to the free threading model, but ATL does.

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]

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 program

Document msjn000020011007dt810001i

