Understanding COM Apartments, Part I

Rating: ☆☆☆☆☆

Jeff Prosise (view profile)
April 13, 2001

Source: CodeGuru.com - http://www.codeguru.com/cpp/com-tech/activex/apts/print.php/c5529/

Let me begin my inaugural column for CodeGuru by stating that I'm on a crusade-a crusade to stamp out bugs related to COM concurrency. COM features a concurrency mechanism that's capable of intercepting and serializing concurrent method calls to objects that were designed to process only one method call at a time. That mechanism centers around the notion of abstract boundaries called apartments. When I troubleshoot COM systems that don't work, probably 40% of the bugs that I find result from a lack of understanding of apartments. This deficiency of knowledge shouldn't be surprising, because apartments are at once one of the most complex areas of COM and also the least well documented. Microsoft's intentions were good, but when they introduced apartments to COM in Windows NT 3.51, they laid a mine field for unwary developers. Play by the rules and you can avoid stepping on mines. But it's hard to obey the rules when you don't know what the rules are.

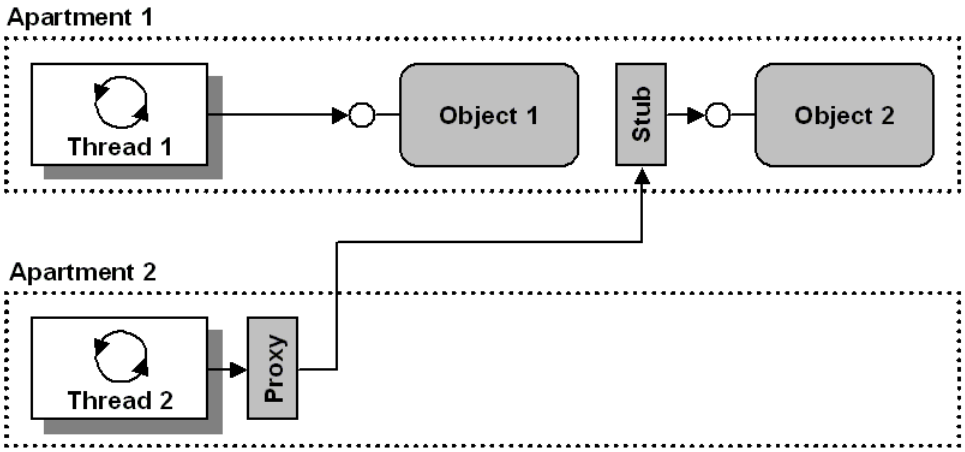
This article is the first in a two-part series that describes what apartments are, why they exist, and how to avoid the problems that they introduce. In Part 1, I'll describe COM's apartment-based concurrency mechanism. In Part 2, I'll provide a set of rules that you can follow to avoid some of the nastiest and most insidious bugs that afflict COM programmers.

Apartment Basics

An *apartment* is a concurrency boundary; it's an imaginary box drawn around objects and client threads that separates COM clients and COM objects that have incompatible threading characteristics. The primary reason that apartments exist is to enable COM to serialize method calls to objects that aren't thread-safe. If you don't tell COM that an object is thread-safe, COM won't allow more than one call at a time to reach the object. Tell COM that the object is thread-safe, however, and it will happily allow the object to field concurrent method calls on multiple threads.
(continued)

Every thread that uses COM, and every object that those threads create, is assigned to an apartment. Apartments never span process boundaries, so if an object and its client reside in two different processes, then they reside in different apartments, too. When a client creates an in-proc object, COM must decide whether to place the object in its creator's apartment or in another apartment in the client process. If COM assigns the object and the thread that created it to the same apartment, then the client has direct, unimpeded access to the object. But if COM places the object in another apartment, calls to the object from the thread that created it are marshaled.

Figure 1 depicts the relationship between threads and objects that share an apartment, and threads and objects that are assigned to different apartments. Calls from thread 1 travel directly to the object that it created. Calls from thread 2 go through a proxy and a stub. COM creates the proxy/stub pair when it marshals the interface pointer to thread 2's apartment. As a rule, an interface pointer must be marshaled when it's passed across apartment boundaries. This means that when custom interfaces are involved, the same proxy/stub DLL (or type library if you prefer typelib marshaling) you use to provide marshaling support for cross-process and cross-machine method calls is needed even for in-proc objects if those objects will be communicating with clients in other apartments.



[Click here for larger image]

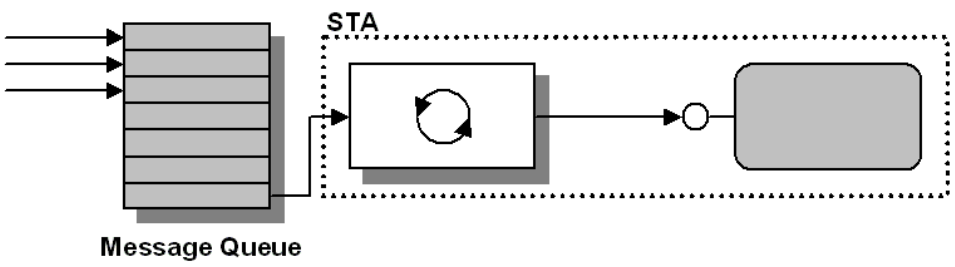
Figure 1: Calls to objects in other apartments are marshaled, even if the object and its caller belong to the same process.

Windows NT 4.0 supports two different types of apartments; Windows 2000 supports three. The three types of apartments are:

- Single-threaded apartments, or STAs (Windows NT 4.0 and Windows 2000)
- Multithreaded apartments, or MTAs (Windows NT 4.0 and Windows 2000)
- Neutral-threaded apartments, or NTAs (Windows 2000 only)

Single-threaded apartments are limited to one thread each, but can host an unlimited number of objects. Additionally, COM places no limit on the number of STAs in a given process. The very first STA created in a process is referred to as the process's main STA. What's important about STAs is that every call destined for an object in an STA is transferred to the STA's thread before being delivered. Since all of the object's calls execute on the same thread, it's impossible for an STA-based object to execute more than one call at a time. COM uses STAs to serialize incoming calls to non-thread-safe objects. If you don't explicitly tell COM that an object is thread-safe, it will place instances of that object in an STA so the object won't suffer concurrent thread accesses.

One of the more interesting aspects of an STA's operation is how COM transfers calls destined for an STA-based object to the STA's thread. When it creates an STA, COM creates a hidden window to go with it. The window is accompanied by a window procedure that knows how to handle private messages representing method calls. When a method call destined for an STA comes out of COM's RPC channel, COM posts a message representing that call to the STA's window. When the thread in the STA retrieves the message, it dispatches it to the hidden window, and the window's window procedure delivers the call to the stub. The stub, in turn, executes the call to the object. Because a thread retrieves, dispatches, and processes just one message at a time, putting an object in an STA enacts a crude (but effective) call serialization mechanism. As shown in Figure 2, if n method calls are placed to an STA-based object at exactly the same time, the calls are queued and delivered to the object one at a time.



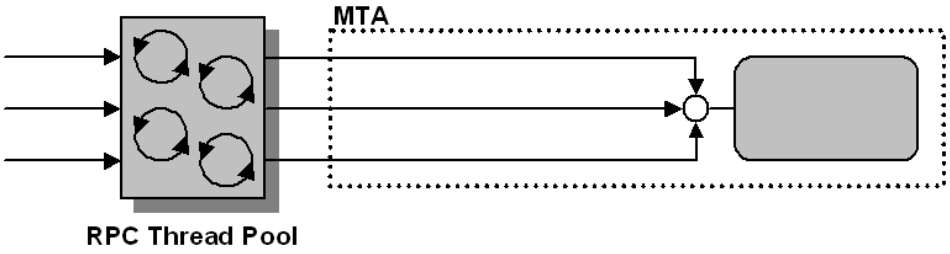
[Click here for larger image]

Figure 2: Calls entering an STA are converted into messages and posted to a message queue. Messages are retrieved from the message queue and converted back into method calls one at a time by the thread running in the STA.

Something equally important happens when a call leaves an STA. COM can't simply allow the thread to block inside the RPC channel, because a callback would induce deadlock. (Imagine what would happen if an STA thread called an object in another apartment, and that object, in turn, called an object in the calling thread's apartment. If the thread were blocked, the call would never return because the one and only thread that can process the callback is waiting in the RPC channel for the original call to return.) Therefore, when a call leaves an STA, COM blocks the calling thread is such a way that the thread can be awakened to process callbacks. To enable such callbacks to occur, COM tracks the causality of each and every method call so it can recognize when an STA thread that's waiting in the RPC channel for a call to return should be released to process another incoming call. By default, a call that arrives at the entrance to an STA blocks if the STA's thread is currently waiting for an outbound call to return and the inbound and outbound calls are not part of the same causality chain. You can change this default behavior by writing a message filter, but that's a topic for another day.

Multithreaded apartments are different animals altogether. COM limits each process to one MTA, but it places no limit on the number of threads or objects in an MTA. If you look inside a process, you might find several STAs containing one thread each, but you'll never see more than on MTA. However, that one MTA, if it exists, can host any number of threads.

How do MTAs differ from STAs? Besides the fact that each process is limited to one MTA and that a given MTA can host any number of threads, an MTA has no hidden window and no message queue. Calls inbound to an object in an MTA are transferred to threads randomly selected from an RPC thread pool and are not serialized (see Figure 3). This means that objects placed in an MTA better be thread-safe, because in the absence of an external synchronization mechanism guaranteeing that an MTA-based object will only receive one call at a time, that object is likely to see calls execute concurrently on different RPC threads.



[Click here for larger image]

Figure 3: Calls entering an MTA are transferred to RPC threads but are not serialized.

When a call leaves an MTA, COM does nothing special. The calling thread is simply allowed to block inside the RPC channel, and if a callback occurs, no deadlock will occur because the callback will be transferred to another RPC thread.

Windows 2000 introduced a third apartment type: the neutral-threaded apartment, or NTA. COM limits processes to a maximum of one NTA each. Threads are never assigned to the NTA; the NTA hosts objects only. What's important about the NTA is the fact that calls to NTA-based objects do not incur thread switches as they enter the NTA. In other words, when a call emanates from an STA or MTA to an NTA in the same process, the calling thread temporarily leaves the apartment it's in and executes code directly in the NTA. Contrast this to STA- and MTA-based objects, which always incur thread switches when called from other apartments. This thread switching accounts for the bulk of the overhead incurred when a call is marshaled between apartments. Eliminating these thread switches improves performance. Therefore, you can think of the NTA as an optimization that allows interapartment method calls to execute more efficiently. In addition, Windows 2000 supports an external synchronization mechanism based on activities that lets you specify separately whether calls to NTA-based objects should be serialized. Activity-based call serialization is more efficient than message-based serialization and can be enacted (or not enacted) on an object-by-object basis.

How Threads are Assigned Apartments

One of the cardinal rules of COM programming is that every thread that uses COM in any way must first initialize COM by calling either CoInitialize or CoInitializeEx. When a thread calls either of these functions, it is placed in an apartment. What type of apartment it's placed in depends on which function the thread called and how it called it. If a thread calls CoInitialize, COM creates a new STA and places the thread inside it:

```
CoInitialize (NULL); // STA
```

If the thread calls CoInitializeEx and passes in the parameter COINIT_APARTMENTTHREADED, it, too, is placed in an STA:

```
CoInitializeEx (NULL, COINIT_APARTMENTTHREADED); // STA
```

Calling CoInitializeEx with a COINIT_MULTITHREADED parameter places the thread inside the process's one and only MTA:

```
CoInitializeEx (NULL, COINIT_MULTITHREADED); // MTA
```

To a very large extent, a process's apartment configuration is driven by how the threads in that process call CoInitialize[Ex]. There are instances in which COM will create a new apartment outside of a call to CoInitialize[Ex], but for now we won't muddy the water by considering such circumstances.

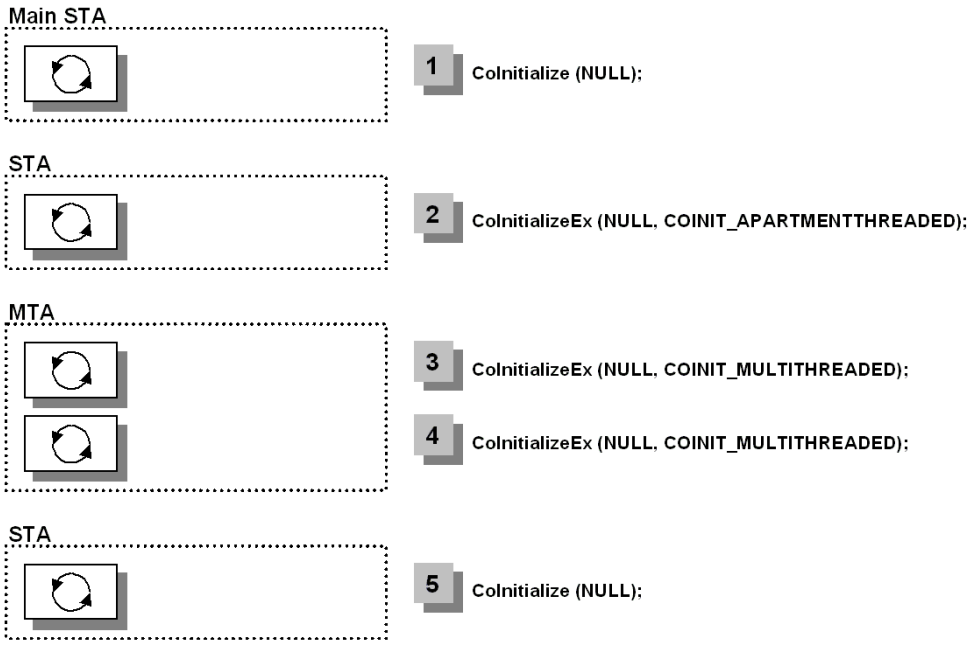
For the sake of example, suppose that a new process is begun and that a thread in that process (thread 1) calls CoInitialize:

```
CoInitialize (NULL); // Thread 1
```

Furthermore, suppose that thread 1 starts threads 2, 3, 4, and 5, and that these threads initialize COM with the following statements:

```
CoInitializeEx (NULL, COINIT_APARTMENTTHREADED); // Thread 2
CoInitializeEx (NULL, COINIT_MULTITHREADED);     // Thread 3
CoInitializeEx (NULL, COINIT_MULTITHREADED);     // Thread 4
CoInitialize (NULL);                             // Thread 5
```

Figure 4 shows the resulting apartment configuration. Threads 1, 2, and 5 are assigned to STAs because of how they called CoInitialize and CoInitializeEx. They're placed in separate STAs because STAs are limited to one thread each. Threads 3 and 4, on the other hand, go in the process's MTA. Remember, COM never creates more than one MTA in a given process, but it's willing to place any number of threads in that MTA.



Click here for larger image

Figure 4: A process with five threads distributed among three STAs and one MTA.

If you're a nuts and bolts person, you might be curious to know more about the physical nature of an apartment-that is, how COM represents apartments internally. Whenever it creates a new apartment, COM allocates an apartment object on the heap and initializes it with important information such as the apartment ID and apartment type. When it assigns a thread to an apartment, COM records the address of the corresponding apartment object in thread-local storage (TLS). Thus, if COM is executing on a thread and it wants to know which, if any, apartment the thread belongs to, all it has to do is reach into thread-local storage and look for the address of an apartment object.

How In-Proc Objects are Assigned Apartments

Now that we know how threads are assigned to apartments, we should consider the other half of the equation-that is, how objects are assigned apartments. The algorithm that COM uses to decide which apartment to create an object in differs depending on whether the object is an in-proc object or out-of-proc object. The in-proc case tends to be the most interesting, because only in-proc objects can be created in their creator's apartment. We'll discuss the in-proc case first, and then double back to discuss apartment considerations for out-of-proc objects.

COM determines which apartment an in-proc object will be created in by reading the object's ThreadingModel value from the registry. ThreadingModel is a named value assigned to the InprocServer32 key that identifies the object's DLL. The following registry entries, shown here in REGEDIT format, identify an object whose CLSID is 99999999-0000-0000-0000-111111111111, whose DLL is MyServer.dll, and whose ThreadingModel is Apartment:

```
[HKEY_CLASSES_ROOT\CLSID\{99999999-0000-0000-0000-111111111111}]
@="My Object"
[HKEY_CLASSES_ROOT\CLSID\{99999999-0000-0000-0000-111111111111}
\InprocServer32]
@="C:\\COM Servers\\MyServer.dll"
"ThreadingModel"="Apartment"
```

Apartment is one of four threading models supported by Windows NT 4.0, and one of five supported by Windows 2000. The five threading models-and the operating systems in which they're supported-are:

ThreadingModel	Apartment Type	NT 4.0	Windows 2000
None	Main STA	X	X
Apartment	Any STA	X	X
Free	MTA	X	X
Both	STA or MTA	X	X
Neutral	NTA		X

The column labeled "Apartment Type" indicates how COM treats an object with the designated ThreadingModel value. For example, COM restricts an object that has no ThreadingModel value ("ThreadingModel=None") to the process's main STA. ThreadingModel=Apartment allows the object to be created in any STA (not just the main STA), while ThreadingModel=Free restricts the object to the MTA and ThreadingModel=Neutral restricts it to the NTA. Only ThreadingModel=Both offers COM any real choice in the matter by giving it permission to create an object in either an STA or MTA.

COM tries its best to place in-proc objects in the same apartments as the threads that create them. For example, if an STA thread creates an object marked ThreadingModel=Apartment, then COM will create the object in the creating thread's STA. If an MTA thread creates a ThreadingModel=Free object, COM will place the object in the MTA alongside the creating thread. Sometimes, however, COM can't put an object in its creator's apartment. If an STA thread, for example, creates an object marked ThreadingModel=Free, then the object will be created in the process's MTA and the creating thread will access the object through a proxy and stub. Similarly, if an MTA thread creates a ThreadingModel=None or ThreadingModel=Apartment object, calls from that thread will be marshaled from the MTA to the object's STA. The following table documents what happens when a thread in either an STA or MTA creates an object marked with any valid ThreadingModel value (or no ThreadingModel value):

	None	Apartment	Free	Both	Neutral
STA	Main STA	Creator's STA	MTA	Creator's STA	NTA
MTA	Main STA	STA	MTA	MTA	NTA

Why does ThreadingModel=None restrict an object to a process's main STA? Because only then can COM ensure that multiple instances of an object that knows nothing about thread safety can execute safely. Suppose that two ThreadingModel=None objects are created from the same DLL. If the objects access any global variables in that DLL (and they almost certainly will), COM must execute all calls to both objects on the same thread, or else the objects might attempt to read or write the same global variable at once. Restricting object instances to the main STA is COM's way of getting the objects on the same thread.

Although it might not be obvious at first, the threading model that you choose has important implications for the code that you write. For example, an object marked ThreadingModel=Free or ThreadingModel=Both should be completely thread-safe since calls to MTA-based objects aren't serialized. Even a ThreadingModel=Apartment object should be partially thread-safe, because ThreadingModel=Apartment doesn't prevent multiple objects created from the same DLL from colliding over shared data. We'll explore this subject in my next column.

How Out-of-Proc Objects are Assigned Apartments

Out-of-process objects don't have ThreadingModel values because COM uses a completely different algorithm to assign out-of-proc objects to apartments. To make a long story short, COM places an out-of-proc object in the same apartment as the thread in the server process that creates the object. Most out-of-proc (EXE) COM servers begin by calling either CoInitialize or CoInitializeEx to place their primary thread in an STA. They then create class objects for the object types that they're capable of creating and register them with CoRegisterClassObject. When an activation request reaches a server that's initialized this way, the request is processed in the process's STA. As a result, objects created in the server process are placed in the process's STA, too.

You can move out-of-proc objects to the MTA by placing the thread that registers the objects' class objects in the MTA. Incoming activation requests will then arrive on RPC threads that execute in the server process's MTA. Objects created in response to these activation requests will reside in the MTA as well.

The upshot is that in most cases involving EXE COM servers, the apartment that hosts the thread that calls CoRegisterClassObject is also the apartment that hosts the objects that the server creates. Exceptions do exist; EXE COM servers written with ATL's CComAutoThreadModule and CComClassFactoryAutoThread classes, which create multiple STAs in the server process and divide objects evenly among those STAs, are one example. These, however, account for a tiny fraction of the EXE COM servers that exist today, and can very much be considered the exception rather than the rule.

Coming Up Next

So what does it all mean? Much of the detail presented in this article may seem too arcane to have any practical value. The reality, however, is that understanding COM apartments is absolutely essential if you want to avoid some of the most common-and potentially most dangerous-pitfalls that afflict COM programmers. You'll see what I mean in my next column.