Sync Threads Automatically

Simplify the management of .NET component concurrency by using automatic thread synchronization to prevent deadlocks.

by Juval Löwy

Technology Toolbox

- ✓ VB.NET
- ✓ C#
- ☐ SQL Server 2000
- ☐ ASP.NET
- ☐ XML
- ☐ VB6
- ✓ Other: .NET Framework

Go Online!

Use these Locator+ codes at **www.visualstudiomagazine.com** to go directly to these related resources.

Download
VS0209BB Download the code for this article, which includes a sample component that uses a synchronization domain and a test client, both provided in C# and VB.NET.

Discuss
VS0209BB_D Discuss this article in the .NET forum.

Read More
VS0209BB_T Read this article online. It includes inline code in both C# and VB.NET.

VS0202BB_T Black Belt, "Demystify .NET App Domains and Contexts," by Juval Löwy

N020213CS "Make Synchronization Automatic" by Juval Löwy

The introduction of multithreading into your VB6 application opens up a Pandora's box of synchronization and concurrency management issues. You have to worry about threads deadlocking while contesting for the same resources. You must synchronize access to objects by concurrent multiple threads. And you have to handle method re-entrance.

Your first step in tackling such issues is to build thread-safe components by equipping them with mechanisms that prevent multiple threads from accessing them and corrupting the state of the objects. This helps, but it still doesn't prevent deadlocks, which occur when thread T1, which owns thread-safe resource R1, tries to access thread-safe resource R2 just as R2's owner, thread T2, tries to access R1 (see Figure 1). Multithreading defects are notoriously hard to isolate, reproduce, and eliminate. They often involve rare race conditions, and fixing one problem often introduces another. Before .NET, it was nontrivial to write robust, high-performance, multithreaded code. You needed a great deal of skill and discipline to succeed.

Enter .NET, which aims at simplifying component concurrency management. By default, all .NET components execute in a multithreaded environment that allows concurrent access by multiple threads. I'll show you how to use .NET's automatic synchronization, which lets you decorate your component with an attribute and have .NET manage concurrent access to the object.

Automatic synchronization revolves around intercepting calls coming into a component's context (see the Read More section in the Go Online box for more information on app domains and contexts). Components must be context-bound to take advantage of .NET automatic synchronization. This means you need to constrain them to execute always in the same context. These components must derive from the ContextBoundObject class directly or indirectly, and they must use the Synchronization attribute, defined in the System.Runtime.Remoting.Contexts namespace:

```
//C#
using System.Runtime.Remoting.Contexts;
[Synchronization]
public class MyClass : ContextBoundObject
{
   public MyClass(){}
   public void DoSomething(){}
   //other methods and data members
}
```

The Synchronization attribute, combined with the ContextBoundObject base class, tells .NET to place the object in a context and associate it with a lock. When a client on thread T1 tries to access the object by calling a method on it (or accessing a public member variable), the client actually interacts with a proxy. .NET intercepts the client access and tries to acquire the lock associated with the object. If the lock isn't owned by another thread currently, .NET acquires the lock and accesses the object on thread T1. .NET

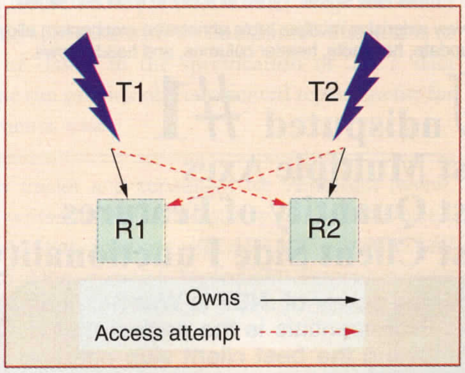

Figure 1 Multithreading Can Be Deadly. A deadlock can occur when two or more threads each owns a resource and is waiting for a resource owned by another thread. Deadlocks are notoriously hard to resolve and often appear unpredictably.

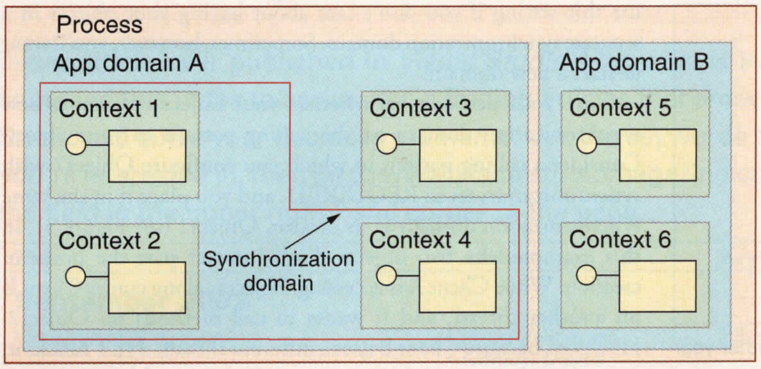

Figure 2 Synchronize Domains, Contexts, and App Domains. The .NET Framework's synchronization domains can save you from error-prone manual thread synchronization. A synchronization domain is independent of context, but is limited to a single app domain. A context belongs to one synchronization domain at most.

releases the lock and returns control to the client when the call returns from the method. However, if another thread T2 were accessing the object, then T1 would be blocked until T2 releases the lock. In fact, all other threads are placed in a queue while the object is accessed by one thread, so they get to access the object in order one at a time.

Avoid Deadlocks With Shared Locks

.NET could have allocated one lock per context-bound object, but that would have been inefficient. Objects can often share a lock and execute on the same thread—if by design the components are all meant to participate in the same activity on behalf of a client. In such situations, allocating one lock per object would waste resources and processing time, forcing .NET to do additional locks and unlocks on every object access. Moreover, sharing locks among objects would reduce the likelihood of deadlocks. If two objects interact with each other and each has its own lock, two different clients on different threads can use these objects. The objects would then deadlock when they try to access each other. If the objects were to share a lock, only one client thread would be allowed to access them.

In .NET, a set of context-bound objects sharing a lock belongs to a *synchronization domain.* Each domain has one lock; multiple threads can't make concurrent calls within the same synchronization domain. When a thread accesses one object in a synchronization domain, that thread can access the other objects in the domain. In fact, the synchronization domain locks all its objects from access by other threads, even though the current thread in the domain accesses only one object at a time.

Synchronization domains are context-independent and can include objects from multiple contexts. However, a synchronization domain is limited to a single app domain: Objects from different app domains can't share a synchronization domain lock. A context can belong to no more than one synchronization domain at a time, if any. If a context belongs to a synchronization domain, then all the objects in that context belong to that synchronization domain (see Figure 2).

You need to decide how to associate a component with a synchronization domain lock. You can choose whether the object needs a lock at all, whether it can share a lock with other objects, or whether it requires a new lock. The SynchronizationAttribute class provides a number of overloaded constructors, all accepting a constant integer flag. Possible values for the flag are NOT_SUPPORTED, SUPPORTED, REQUIRED, and REQUIRES_NEW. You use these constants to *allocate* an object to a synchronization domain relative to its creating client:

```
//C#
[Synchronization(SynchronizationAttribute.REQUIRES_NEW)]
public class MyClass : ContextBoundObject
{}
```

The default constructor of the SynchronizationAttribute class uses REQUIRED. .NET gives you three options. First, you can place an object in its creator's synchronization domain, in which case the object shares a lock with its creator. Second, you can place an object in a new synchronization domain, where the object has its own lock and starts a new synchronization domain. Finally, you can choose to not place the object in a synchronization domain, in which case you get concurrent access and no lock.

Pick a Sync Domain

.NET determines an object's synchronization domain at creation time, based on the synchronization domain of its creator and the constant value you choose for the Synchronization attribute (see Table 1). That's because .NET uses a heuristic, which assumes

Synchronization constant flag	Does the object's creator have a synchronization domain?	Synchronization domain determined by .NET
NOT SUPPORTED	—	The object will never be part of a synchronization domain, regardless of whether its creator has a synchronization domain.
SUPPORTED	Yes	.NET places the object in its creator's synchronization domain.
SUPPORTED	No	The newly created object doesn't have a synchronization domain.
REQUIRED	Yes	.NET puts the object in its creator's synchronization domain.
REQUIRED	No	.NET creates a new synchronization domain for the object.
REQUIRES NEW	—	.NET creates a new synchronization domain for the object, regardless of whether its creator has a synchronization domain.

Table 1 Determine Your Object's Synchronization Domain. .NET determines an object's synchronization domain at creation time, based on the synchronization domain of its creator and the constant value you provide for the Synchronization attribute. REQUIRED is the default for the Synchronization Attribute class.

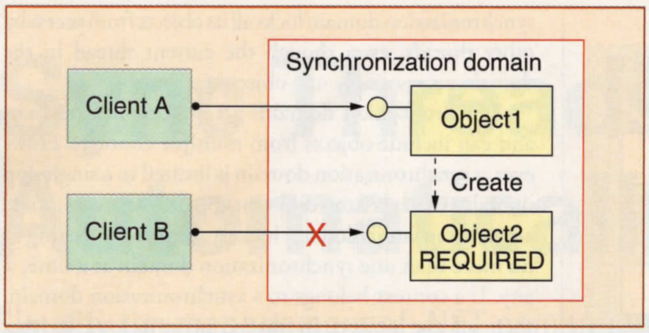

Figure 3 Share a Lock With the Creator. Determine whether your object requires a new synchronization domain based on the calling pattern to your object. Consider this calling pattern, in which you configure Object2 with synchronization set to REQUIRED, and you place it in the same synchronization domain as its creator, Object1. Sharing the lock with the creator when the two objects don't interact causes Client B to wait until the first call is completed, even though it could access the object safely.

that calling patterns, interactions, and synchronization needs between objects usually resemble the relationship between an object and its creator.

The various Synchronization attribute construction values give you a variety of options. An object with synchronization set to NOT_SUPPORTED never participates in a synchronization domain. The object must provide its own synchronization mechanism. Use this setting only if you expect concurrent access, and you want to provide your own synchronization mechanisms. But why do that? Context-bound objects should leverage .NET's component services support.

An object with synchronization set to SUPPORTED shares its creator's synchronization domain if it has one, and has no synchronization of its own if the creator doesn't have one. Use SUPPORTED for the rare case when the component itself has no need for synchronization, but downstream objects it creates do require it. Components with synchronization SUPPORTED can propagate the synchronization domain of their creating client to downstream objects, which then share one synchronization domain instead of having separate ones. This reduces the likelihood of deadlocks.

You most often set object synchronization to REQUIRED—that's why it's the SynchronizationAttribute class default. Always use this setting if you don't care about having your objects in a separate synchronization domain. Sometimes, however, you'll want to start a new domain.

Base your decision on whether your object requires a new synchronization domain on the calling pattern to your object. Consider a calling pattern in which you configure Object2 with synchronization set to REQUIRED, and you place it in the same synchronization domain as its creator, Object1 (see Figure 3). In this example, the two objects don't interact past the point of creation. While Client A is accessing Object1, along comes Client B on another thread, and it wants to call methods on Object2. However, because Client B uses a different thread, .NET blocks it, even though it could have accessed Object2 safely, because it doesn't violate the synchronization requirement for the creating object, Object1.

Class Factories Need New Domains

On the other hand, if you were to configure Object2 to require its own synchronization domain by using REQUIRES_NEW, the object could process calls from other clients at the same time as Object1 (see Figure 4). Using class factories to create objects provides a classic example of needing to configure components to require a new synchronization domain.

Class factories usually require thread safety because they service multiple clients. Once a factory creates an object, though, it hands the object back to a client and has nothing more to do with it. You need to configure the objects to require a new synchronization domain because you don't want all the objects created by a class factory to share the same synchronization domain.

However, calls from the creator object (Object1) to Object2 will now potentially block and will be more expensive because the calls must cross context boundaries and pay the overhead of trying to acquire the lock. You can synchronize context-bound objects most easily using .NET synchronization domains. These provide a modern synchronization technique that formally eliminates many synchronization problems and the consequent need to code around them, then test the handcrafted solution.

Synchronization domains provide a substantial productivity gain, but you do need to consider four limitations. First, you can use synchronization domains only with context-bound objects. All other .NET types require manual synchronization objects. Second, you could have performance issues when you access context-bound objects using proxies and interceptors in an intense calling pattern. Third, synchronization domains don't protect static class members and static methods. Those require manual synchronization objects. Finally, synchronization domains are not throughput-oriented. An incoming thread locks a set of objects even if it interacts with only one. This precludes other threads from accessing these objects, and theoretically could degrade application throughput.

For balance, you must rely on synchronization domains and other advanced component services in any decent-sized application—or whenever productivity and quality are top priorities. VSM

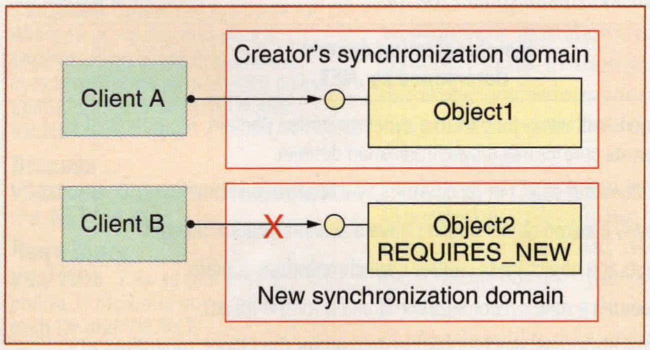

Figure 4 Start a New Synchronization Domain. In this calling pattern, having a synchronization domain separate from the created object enables the object to service its clients more efficiently. Using class factories to create objects provides a classic example of needing to configure components to require a new synchronization domain.

Juval Löwy is a software architect and principal of IDesign, a consulting and training company focused on .NET. This article derives from his upcoming book on programming .NET components (O'Reilly). Juval speaks at development conferences and chairs the .NET California Bay Area User Group's program committee. Contact him at www.idesign.net.