Luis Serra

StaThreadSyncronizer

Switching work into STA threads

You might be wondering why I am coding this in the first place. STA is a threading model that is used by COM. A long time ago, before the age of .NET Remoting and WCF (sounds like hundred years ago), developers coded COM classes that could only run on Single Apartment Threads (STA). Why? Because the COM runtime would handle thread marshaling for the developer, and always made sure that your COM class would execute on the same thread. This way, the COM developer did not need to worry about multi-threading, Mutexes, Semaphores, Events, and all the other multi-threading toys out there. Just for the record, COM also provided a Multi Threading Apartment model (for the brave ones). With MTA, the developer had to worry about multi-threading issues, but had more control. There is a lot of documentation out there about MTA and STA, all you have to do is Google "*STA MTA*" and read all the history about it. Thank God, I don't need to code COM anymore. However, there is a lot of COM out there, and a lot of business logic is coded into COM classes that can only execute on an STA thread. To be able to call these classes from any thread in .NET, I decided to code a custom STA thread synchronization context. After all the sweat and work put into it, it felt right to show you it.

How do we switch between two threads?

The first question is how would we manage marshaling between two running threads. This problem is typically solved by implementing some sort of a common communication block that both threads can read and write from. An ideal communication object between two threads is a queue. A queue provides us the ability to send work from one thread to another based on the invocation order. This is the typical Consumer / Provider model, where one thread plays the role of a consumer (reading from the queue), and another thread plays the role of a provider (writing items to the queue). To simplify things, let's see how this might work:

- **Thread 1:** Sends a message to a common queue.

- **Thread 2:** Listens for incoming messages from the common queue. In our case, Thread 2 will be an STA thread.

A closer look at the blocking queue

I wanted to have a queue to queue up work items from thread X to my STA thread. I also wanted my thread to dequeue items only when there are items in the queue. If there are no items, I want the `Dequeue` method to wait until something pops into the queue. Typically, this is called a "*Blocking Queue*". (See code)

- Because this queue is used by multiple threads, notice that I am blocking access to the queue using the `lock` statement.
- Normally, I block the `Dequeue` method until there is an item in the queue. The way this works is by having a semaphore that represents all the items in the queue as resources. When the semaphore is created for the first time, the queue is empty, and therefore there are no resources available, so calling `Dequeue` will block (notice there is zero at the first argument indicating no available resources, and a large number for the second argument representing the size of the queue).

```
private Semaphore mSemaphore = new Semaphore(0, int.MaxValue);
```

- Notice that when I dequeue an item, I block on an array of `WaitHandle`s (`WaitHandle.WaitAny(mWaitHandles);`). This code means "*Wait until there is a message, or until the read thread is marked to stop running.*"
- I have not shown the actual thread yet, I will show it next. However, the STA thread will be the reading thread, spending most of its time waiting for a message on the queue or processing a message from the queue.
- Notice that when a message is enqueued into the queue, it releases the semaphore, indicating that a resource is available; this will cause the `Dequeue` method to unblock.
- The semaphore has a max limit of `Int.Max`; we should never reach anything close to this limit as long as the thread is dequeue-ing more of less as fast as it is enqueue-ing.

The SendOrPostCallbackItem class

Notice that the blocking queue class is generic, this was done in case I decide to re-use it in another application (and you are free to use it for your needs as well). So, what are we planning to put into this queue? Considering this queue is responsible to marshal code from one thread to another, the ideal item to queue is a delegate. Still, we need a little more than a delegate, and not just a simple delegate, but a `SendOrPostCallback` delegate.

- `SendOrPostCallbackItem` contains the delegate we wish to execute on the STA thread.
- The `Send` and `Post` are really helper methods, they are both responsible for launching the code, and they are both designed to be called from the STA thread. However, because the `Send` is required to block, and to report exceptions back to the calling thread (non-STA thread), I use a `ManualResentEvent` when the execution is complete. I also keep track of the exception; if there is one, it will be thrown on the non-STA thread (producer thread).
- `Post` is simple. It just calls the method, no need to notify when it is done, and there is no need to track the exception either.

Overall, this class is responsible for two main tasks. Storing the delegate to execute and executing it in two possible modes: `Send` and `Post`. `Send` requires additional tracking (such as the exception and notification of complication). `Post` just executes the method without doing anything else. Normally, if `Post` is executed on the STA thread, any exceptions reported by the delegate will cause the thread to end.

The STA thread and all its glory

Finally, we can show and explain the meat and potatoes of this sync context. Now that we have a queue, and we know what we are planning to push into it, let's look at the STA thread (the thread responsible for marshaling code).

One of the most important parts of this class is in the constructor, so let's take a look at it.

```
internal StaThread(IQueueReader<sendorpostcallbackitem> reader)
```

```
{
    mQueueConsumer = reader;
    mStaThread = new Thread(Run);
    mStaThread.Name = "STA Worker Thread";
    mStaThread.SetApartmentState(ApartmentState.STA);
}
```

- This class takes an interface of type `IQueueReader`, this is really our blocking queue. The reason I decided to put an interface here is because this thread is a reading thread and should not have access to writing methods.
- The thread is being setup as an STA thread. Giving the thread a name helps when debugging using the thread output window.
- Notice, the thread is not started yet. A method called `Start` will start the thread, and this will happen within our `StaSynchronizationContext` class, which I will show soon.

Let's take a look at the `Run` method. The `Run` method represents our STA thread. Its main job is to dequeue items from our blocking queue and execute them. Executing any work items on the `Run` method means executing them on the STA thread. Therefore, it doesn't really matter which thread has placed them in the queue, what's important is that items are read within the STA thread, and executed in the STA thread. If you think about this, this is, in fact, thread marshalling in action.

```
private void Run()
{
    while (true)
    {
        bool stop = mStopEvent.WaitOne(0);
        if (stop)
        {
            break;
        }

        SendOrPostCallbackItem workItem = mQueueConsumer.Dequeue();
        if (workItem != null)
            workItem.Execute();
    }
}
```

I tried to keep the `Run` method as simple as possible, but let's stress out a few points.

- The STA thread is running all the time, so I have made a `while(true)` loop. Normally, I am not a fan of this type of loop, but I wanted the reader of the code to understand that this thread is not supposed to go down unless the context class is disposed. A `while(true)` sends this type of a message.
- `mStopEvent` is a `ManualResetEvent`, it is signaled when the STA thread is marked to stop running. When the `Stop()` method is called, the `mStopEvent` is set, causing the main loop to exit. The `Stop` method also releases any waiting dequeue operation by marking the queue to stop processing messages.
- `mQueueConsumer.Dequeue()` is responsible for reading work items from the queue. This method will block until a work item is in the queue.
- When a work-item is dequeued, the work-item is executed. `Execute()`, if you remember, will execute the code in the delegate associated with the work item. It is during this `Execute` method that the code is marshaled on the STA thread.

Creating the STA synchronization context class

We have almost all the pieces we need to have our STA Sync Context running. We got a work item that contains our delegate to execute on the STA thread. We got a nice little blocking queue to handle communication between the STA thread and any other thread. We even have our little STA `Run` method always looking at our queue, pumping messages out of it, and running any work items that are fetched. The only thing we are missing now is the actual Synchronization Context class itself.

This is really the class that uses all the other classes I have shown before. I have named it the `StaSynchronizationContext` because it is responsible to marshal code into an STA thread, allowing the caller to execute COM APIs that must be on an STA thread. Let's look a the Send API which is responsible for sending work on the STA thread. Notice, this class inherits from `SynchronizationContext`, but overrides the default `Send` and `Post` methods.

```
public void Send(SendOrPostCallback d, object state, int milisecondsTimeout)
```

```
        {
            // to avoid deadlock!
            if (Thread.CurrentThread.ManagedThreadId ==
mStaThread.ManagedThreadId)
            {
                d(state);
                return;
            }

            // create an item for execution
            SendOrPostCallbackItem item = new SendOrPostCallbackItem(d, state,
ExecutionType.Send);
            // queue the item
            mQueue.Enqueue(item);
            // wait for the item execution to end
            if (!item.ExecutionCompleteWaitHandle.WaitOne(miliseconds Timeout))
                mQueue.Dequeue(item);

            // if there was an exception, throw it on the caller thread, not the
            // sta thread.
            if (item.ExecutedWithException)
                throw item.Exception;
        }
```

Notice that the send operation is a blocking operation, this means we block until the operation on the STA thread is complete. Remember, we have placed a `ManualReset` event on the `SendOrPostCallbackItem` class, so we know when the execution is done. We are also trapping and caching any exceptions within `SendOrPostCallbackItem` so we can throw them on the calling thread and not on the STA thread. The `Post`, on the other hand, is not a waiting call, so all we do is queue the item and we are not waiting for the delegate execution to finish.

That's it, we now have a `SynchronizationContext` that will marshal code between any thread into a single STA thread.