Exercise 12 Threads and Apartments

In this exercise you will investigate the issues involved in synchronising method calls across apartment boundaries.

Step 1 - Getting Started

Open the workspace ***Threads and Apartments*** in the following folder

```
COM Programming\Exercises\Threads and Apartments\
```

You will find three projects, as follows:

ThreadServer contains code for two in process COM classes (***Single*** and ***Multi***), one of which wants to live in an STA and one of which wants to live in the MTA. These are implemented in the classes ***CSingle*** and ***CMulti*** respectively. Both classes support the same IQAPrint interface, with its single method for printing out a string (very slowly).

STAThreadTest contains the code for a test harness that will instantiate a single instance of the ***Single*** COM class, and will promptly fire multiple threads through its PrintString() method simultaneously.

Unfortunately, the code that has been written in these first two projects is not correct, and will require some modifications on your part to make the applications work correctly.

MTAThreadTest contains the code for a test harness that will instantiate a single instance of the ***Multi*** COM class, and will promptly fire multiple threads through its PrintString() method simultaneously. This test harness is completely finished and requires no modification.

Step 2 – Modifying the Multi COM Class to be Thread Safe

Build and run the ***MTAThreadTest*** program. The output is garbled, and we don't want that! So let's fix it.

Using Class View, locate the function ***PrintString()*** on the ***CMulti*** class in the ***ThreadServer*** project. Remember that because this COM class has indicated that it wants to live in an MTA, multiple threads can execute this code simultaneously, which is why we get garbled output.

What we need to do is use the ***Lock()*** and ***Unlock()*** methods that are provided in one of ***Multi***'s base classes, ***CComObjectRootEx***, to provide a simple synchronisation mechanism.

Inside the ***PrintString()*** function, add a call to ***Lock()*** before the loop that prints all the characters and a call to ***Unlock()*** after you have printed the CR/LF pair.

Build and test with the ***MTATestServer*** and you should no longer get garbled output.

Always remember – when you're writing a COM object and you specify that it can live in the MTA, it is up to YOU to ensure that the code is synchronised correctly.

Step 3 – Using Marshalling APIs to Pass References between Threads

Build and run the project ***STAThreadTest***. Oh no, it looks like we're getting more garbled output.

We have to address a different problem here. In this case, the COM object that is being used is the ***Single*** object, which will be loaded in an STA. This is coded correctly for an STA object, in that it is relying on COM itself to provide synchronisation (***Lock()*** and ***Unlock()*** are therefore not required or used).

However, the client program currently violates the laws of COM, and needs fixing. It is passing raw interface pointers between threads, whereas it should be passing apartment neutral stream references.

So let’s fix the problem.

Open the file ***STAThreadTest.cpp*** from the ***STAThreadTest*** project, and locate the following code in the ***main()*** function

```
for( int i = 0; i < 4; ++i )
{
        InterlockedIncrement( &threadCount );

        //
        // TODO:    Passing a raw interface pointer here
        //          isn't robust. Change the code so that
        //          we pass a reference to an apartment
        //          neutral stream.
        //
        _beginthread( ThreadFunc, 0, p );
}
```

INSIDE the for() block, declare a local variable, called ***pStream***, of type ***IStream **** and set it to 0.

Then, using the function ***CoMarshalInterThreadInterfaceInStream()***, marshal the reference ***p*** into ***pStream***. You should now pass ***pStream*** through to the ***_beginthread()*** instead of ***p***.

Now we have to alter ***ThreadFunc()*** to deal with the new type of reference that it is dealing with.

Locate the function ***ThreadFunc()*** and the section of code as follows:

```
//
// TODO:    This code is unsafe because we are taking
//          a raw interface and using it from another
//          thread. Replace this line with a more
//          robust mechanism.
//
spQAP = (IQAPrint *) arg;
```

Use ***CoGetInterfaceAndReleaseStream()*** to convert the ***IStream*** pointer held in ***arg*** back into an ***IQAPrint*** reference. Remember, you can always use the address-of operator (&) on a smart pointer as the ***void**** placeholder argument in calls like ***QueryInterface()*** and ***CoGetInterfaceAndReleaseStream()***.

With these modifications in place, build and test. Voila, no more garbled output.

This exercise shows how important it is to remember the golden rule – an object reference can only be used from the thread that created it, otherwise chaos will ensue!

Step 4 – Proxies? What proxy?

Did you remember that a proxy/stub DLL was in place? If you examine the Custom Build steps on the ThreadServer project, you will see that it was building and registering the proxy/stub DLL behind the scenes.

Try unregistering the proxy/stub DLL (run regsvr32 /U threadserverps.dll) and see what happens when you try to run the STAThreadTest – it fails badly. Re-register the DLL, and its all fine again.

That other golden rule with apartments – to cross an apartment (or any other) boundary requires a proxy/stub DLL.