Synchronising an Asynchronous call in c#

Asked 15 years, 9 months ago Modified 15 years, 9 months ago Viewed 5k times

6

I've run into quite an awkward predicament in a project at work. We need to create users across 4 or 5 different services, and have set it up in such a way that if one fails, they all fail. They're encapsulated within a transaction scope block.

One of the services we need to add users to requires telnetting in, and fudging some data. There are other ways to do it (that cost money) but for now that's what we're stuck with. Adding one user takes approximately 3 minutes. We're going to work on getting that down significantly as can be imagined but that's not really the point. This call is asynchronous and kind of has to be to work correctly. The punchline is, there can only be a maximum of 10 connections to the service.

Our project is being set up to create users in a batch. So potentially 50 users created at a time. This presents a problem when only 10 connections can be made via telnet, and a processed user isn't likely to take very long other than the telnet service. I now need to synchronise this process so the rest can't continue until it has finished.

We're using callbacks and delegates with asynchronous calls to implement the functionality. What would be the best way of encapsulating the asynchronous portion, and not continuing until it's completed?

Should we set up a loop that only terminates when the call is finished? Is there something in the Threading library that could help? I've never worked with threads before so this will be a first for me. What tools are there to help with this problem?

EDIT:

If I use the BeginInvoke / EndInvoke pattern, will asynch calls within the first delegate honour the begin/end also?

Example:

```
public void dele1(string message) {
    Console.Write(message);
    delegate2 del2 = new delegate2;
    del2();
    Console.Write("End of Delegate 2");
}

public void dele2() {
    // Long Processing
    Console.Write("Delegate 2");
}
```

```
public delegate void delegate1(String message);
public delegate void delegate2();

delegate1 del1 = new delegate1(dele1);
del1("Delegate 1").BeginInvoke;
del1().EndInvoke;
Console.Write("End of Delegate 1");
```

// Expected Output (End Invoke waits until Delegate 2 is finished):

```
Delegate 1
End of Delegate 2
Delegate 2
End of Delegate 1
```

// Or (End Invoke only waits for delegate 1 to finish but not any internal delegate calls):

```
Delegate 1
End of Delegate 2
End of Delegate 1
Delegate 2
```

Will end invoke wait until the second delegate finishes processing also? Or will I need to use the invoke patterns on all delegate calls?

c# multithreading asynchronous delegates

Share

Improve this question

Follow

edited Mar 16, 2009 at 23:35

asked Mar 16, 2009 at 7:38

Josh Smeaton

48.7k ● 24 ● 135 ● 165

Not sure on your syntax for your edited example. Can you clarify more what is happening with del1? – strager Mar 16, 2009 at 23:04

6 Answers

Sorted by: Highest score (default)

7

You could indeed use monitors, semaphores, or you could even spin wait until your asynchronous method call(s) is done.

But you can get this for free too. If you call `EndInvoke()` on a delegate that was previously started with `BeginInvoke()`, you block until the asynchronous work is done.

Not sure if this helps because you have to be using this asynchronous pattern. If it does, you get Asynchronous execution (and conversion from Asynchronous back to synchronous calls) for free.

Check out [Calling Synchronous Methods Asynchronously on MSDN]() for more info on this pattern.

I hope this helps!

Share
Improve this answer
Follow

edited Mar 16, 2009 at 23:03
strager
89.9k ● 27 ● 138 ● 179

answered Mar 16, 2009 at 7:58
Aaron
829 ● 6 ● 12

That sounds like exactly what I need! Thanks mate. – Josh Smeaton Mar 16, 2009 at 10:30

2

It sounds to me like you want a queue... if there is a request in progress, add to the queue (with a `Monitor`); when a call completes, check the queue...

Share Improve this answer Follow

answered Mar 16, 2009 at 7:41
Marc Gravell
1.1m ● 273 ● 2.6k ● 3k

2

`IAsyncResult` has a property `AsyncWaitHandle` which gives you a wait handle (if one is available) which will be signalled when the operation completes.

Thus you can use `WaitHandle.WaitAll` (or `.WaitAny`) to perform non-spinning waits on many handles at once.

Share Improve this answer Follow

answered Mar 16, 2009 at 17:15
Richard
109k ● 21 ● 210 ● 272

0

You have a few options. Here are a couple of ideas -

First off, you could use a [ManualResetEvent]() to "block" the main thread until your async operations all complete. The idea is to just have the main thread call you function, then do event.WaitOne(), and the function is responsible for setting the event.

Alternatively, you could try to use something like a Semaphore. It is designed to help in these situations, since you can limit it to 10 simulataneous occurances without worrying about trying to handle it yourself. This would probably be my preferred approach.

Share Improve this answer Follow

answered Mar 16, 2009 at 7:43

Reed Copsey
564k ● 80 ● 1.2k ● 1.4k

0

There are two reasons for using threads:

- to take advantage of CPU resources, so speeding things up
- to write several simultaneous state machines as straightforward methods, which makes them more readable

The disadvantage of using threads is:

- it's really difficult

If the bottleneck is a 3 minute wait on a remote machine, it may be worth noting that multiple threads aren't going to get you any performance advantage. You could write the whole thiing single-threaded, maintaining a set of up to ten "state machine" objects and moving them through the stages of user creation, all from one thread, with little difference in the over all performance.

So what you're perhaps looking for is nice readable code layout, by making the user creation operation into a sequence of calls that read like a single method (which they would be if you ran them as a thread).

Another way to get that is via an iterator method, i.e. containing `yield return` statements, which also gives you the advantages of single-method readability. I must have linked to this three times in the last week! It's Jeffrey Richter's article about AsyncEnumerator.

Share Improve this answer Follow

answered Mar 16, 2009 at 8:04

Daniel Earwicker
117k ● 38 ● 208 ● 286

0

I'm a little unclear on how you're using TransactionScope (so I may be off base), but you can create DependentTransactions that can be doled out to the worker threads. Your top-level transaction will not commit until all DependentTransactions have successfully committed and vice-versa with rollback. I've found this as an easy way to do all or nothing across threads as long as the operation you're invoking encapsulates commit/rollback functionality (IEnlistmentNotification or the like).

Share Improve this answer Follow

answered Mar 16, 2009 at 23:55

jsw
1,782 ● 1 ● 14 ● 20

You're right, and if we weren't limited to the 10 max connections for the final process, this would be fine. Unfortunately, we are limited to 10 (actually we're allowed 2), so it's easier to do them all in sequence rather than keeping a queue. – Josh Smeaton Mar 17, 2009 at 2:48

Limited concurrent connections... always a fun requirement ;) – jsw Mar 17, 2009 at 7:31