Parallel Programming with .NET

All about Async/Await, System.Threading.Tasks, System.Collections.Concurrent, System.Linq, and more…

Visual Studio

Await, SynchronizationContext, and Console Apps

★★★★★

January 20, 2012 by Stephen Toub - MSFT // 35 Comments

When I discuss the new async language features of C# and Visual Basic, one of the attributes I ascribe to the await keyword is that it "tries to bring you back to where you were." For example, if you use await on the UI thread of your WPF application, the code that comes after the await completes should run back on that same UI thread.

There are several mechanisms that are used by the async/await infrastructure under the covers to make this marshaling work: SynchronizationContext and TaskScheduler. While the transformation is much more complicated than what I'm about to show, logically you can think of the following code:

```
await FooAsync();
RestOfMethod();
```

as being similar in nature to this:

```
var t = FooAsync();
var currentContext = SynchronizationContext.Current;
t.ContinueWith(delegate
```

```
    {
        if (currentContext == null)
            RestOfMethod();
        else
            currentContext.Post(delegate { RestOfMethod(); }, null);
    }, TaskScheduler.Current);
```

In other words, before the async method yields to asynchronously wait for the Task 't', we capture the current SynchronizationContext. When the Task being awaited completes, a continuation will run the remainder of the asynchronous method. If the captured SynchronizationContext was null, then RestOfMethod() will be executed in the original TaskScheduler (which is often TaskScheduler.Default, meaning the ThreadPool). If, however, the captured context wasn't null, then the execution of RestOfMethod() will be posted to the captured context to run there.

Both SynchronizationContext and TaskScheduler are abstractions that represent a "scheduler", something that you give some work to, and it determines when and where to run that work. There are many different forms of schedulers. For example, the ThreadPool is a scheduler: you call ThreadPool.QueueUserWorkItem to supply a delegate to run, that delegate gets queued, and one of the ThreadPool's threads eventually picks up and runs that delegate. Your user interface also has a scheduler: the message pump. A dedicated thread sits in a loop, monitoring a queue of messages and processing each; that loop typically processes messages like mouse events or keyboard events or paint events, but in many frameworks you can also explicitly hand it work to do, e.g. the Control.BeginInvoke method in Windows Forms, or the Dispatcher.BeginInvoke method in WPF.

SynchronizationContext, then, is just an abstract class that can be used to represent such a scheduler. The base class exposes several virtual methods, but we'll focus on just one: Post. Post accepts a delegate, and the implementation of Post gets to decide when and where to run that delegate. The default implementation of SynchronizationContext.Post just turns around and passes it off to the ThreadPool via QueueUserWorkItem. But frameworks can derive their own context from SynchronizationContext and override the Post method to be more appropriate to the scheduler being represented. In the case of Windows Forms, for example, the WindowsFormsSynchronizationContext implements Post to pass the delegate off to Control.BeginInvoke. For DispatcherSynchronizationContext in WPF, it calls to Dispatcher.BeginInvoke. And so on.

That's how await "brings you back to where you were." It asks for the SynchronizationContext that's representing the current environment, and then when the await completes, the continuation is posted back to that context. It's up to the implementation of the captured context to run the delegate in the right place, e.g. in the case of a UI app, that means running the delegate on the UI thread. This explanation also helps to highlight what happens if the environment didn't set a

SynchronizationContext onto the current thread (and if there's not special TaskScheduler, as there isn't in this case). If the context comes back as null, then the continuation could run "anywhere". I put anywhere in quotes because obviously the continuation can't run "anywhere," but logically you can think of it like that... it'll either end up running on the same thread that completed the awaited task, or it'll end up running in the ThreadPool.

All of the UI application types you can create in Visual Studio will end up having a special SynchronizationContext published on the UI thread. Windows Forms, Windows Presentation Foundation, Metro style apps... they all have one. But there's one common kind of application that doesn't have a SynchronizationContext: console apps. When your console application's Main method is invoked, SynchronizationContext.Current will return null. That means that if you invoke an asynchronous method in your console app, unless you do something special, your asynchronous methods will not have thread affinity: the continuations within those asynchronous methods could end up running "anywhere."

As an example, consider this application:

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

class Program
{
  static void Main()
  {
    DemoAsync().Wait();
  }

  static async Task DemoAsync()
  {
    var d = new Dictionary<int, int>();
    for (int i = 0; i < 10000; i++)
    {
      int id = Thread.CurrentThread.ManagedThreadId;
      int count;
      d[id] = d.TryGetValue(id, out count) ? count+1 : 1;

      await Task.Yield();
    }
    foreach (var pair in d) Console.WriteLine(pair);
  }
}
```

Here I've created a dictionary that maps thread IDs to the number of times we encountered that particular thread. For thousands of iterations, I get the current thread's ID and increment the appropriate element of my histogram, then yield. The act of yielding will use a continuation to run the remainder of the method. Here's some representative output I see from executing this app:

```
[1, 1]
[3, 2687]
[4, 2399]
[5, 2397]
[6, 2516]
Press any key to continue . . .
```

We can see here that the execution of this code used 5 threads over the course of its run. Interestingly, one of the threads only had one hit. Can you guess which thread that was? It's the thread running the Main method of the console app. When we call DemoAsync, it runs synchronously until the first await the yields, so the first time we check the ManagedThreadId for the current thread, we're still on the thread that invoked DemoAsync. Once we hit the await, the method returns back to Main(), which then blocks waiting on the returned Task to complete. The continuations used by the remainder of the async method's execution would have been posted to SynchronizationContext.Current, except that it a console app, it's null (unless you explicitly override that with SynchronizationContext.SetSynchronizationContext). So the continuations just get scheduled to run on the ThreadPool. That's where the rest of those threads are coming from… they're all ThreadPool threads.

Is it a problem then that using async like this in a console app might end up running continuations on ThreadPool threads? I can't answer that, because the answer is entirely up to what kind of semantics you need in your application. For many applications, this will be perfectly reasonable behavior. Other applications, however, may require thread affinity, such that all of the continuations run on the same thread. For example, if you invoked multiple async methods concurrently, you might want all the continuations they use to be serialized, and an easy way to guarantee that is to ensure that only one thread is used for executing all of the continuations. If your application does demand such behavior, are you out of luck? Thankfully, the answer is 'no'. You can add such behavior yourself.

If you've made it this far in reading, hopefully the components of a solution here have started to become obvious. You effectively need a message pump, a scheduler, something that runs on the Main thread of your app processing a queue of work. And you need a SynchronizationContext (or a TaskScheduler if you prefer) that feeds the await continuations into that queue. With that framework in place, let's build a solution.

First, we need our SynchronizationContext. As described in the previous para-

graph, we'll need a queue to store the work to be done. The work provided to the Post method comes in the form of two objects: a SendOrPostCallback delegate, and an object state that is meant to be passed into that delegate when it's invoked. As such, we'll have our queue store a KeyValuePair<TKey,TValue> of these two objects. What kind of queue data structure should we use? We need something ideally suited to handle producer/consumer scenarios, as our asynchronous method will be "producing" these pairs of work, and our pumping loop will need to be "consuming" them from the queue and executing them. .NET 4 saw the introduction of the perfect type for the job: BlockingCollection<T>. BlockingCollection<T> is a data structure that encapsulates not only a queue, but also all of the synchronization necessary to coordinate between a producer adding elements to that queue and a consumer removing them, including blocking the consumer attempting a removal while the queue is empty.

With that, the pieces fall into place: a BlockingCollection<KeyValuePair<SendOrPostCallback,object>> instance; a Post method that adds to the queue; another method that sits in a consuming loop, removing each work item and processing it; and finally another method that lets the queue know that no more work will arrive, allowing the consuming loop to exit once the queue is empty.

```
private sealed class SingleThreadSynchronizationContext :
   SynchronizationContext
{
   private readonly
    BlockingCollection<KeyValuePair<SendOrPostCallback,object>>
     m_queue =
      new BlockingCollection<KeyValuePair<SendOrPostCallback,object>>();

   public override void Post(SendOrPostCallback d, object state)
   {
      m_queue.Add(
         new KeyValuePair<SendOrPostCallback,object>(d, state));
   }

   public void RunOnCurrentThread()
   {
      KeyValuePair<SendOrPostCallback, object> workItem;
      while(m_queue.TryTake(out workItem, Timeout.Infinite))
         workItem.Key(workItem.Value);
   }

   public void Complete() { m_queue.CompleteAdding(); }

   …
}
```

Believe it or not, we're already half done with our solution. We need to instantiate one of these contexts and set it as current onto the current thread, so that when we then invoke the asynchronous method, that method's awaits will see this context as Current. We need to alert the context to when there won't be any more work arriving, which we can do by using a continuation to call Complete on our context when the Task returned from the async method is compelted. We need to run the processing loop via the context's RunOnCurrentThread method. And we need to propagate any exceptions that may have occurred during the async method's processing. All in all, it's just a few lines:

```
public static void Run(Func<Task> func)
{
  var prevCtx = SynchronizationContext.Current;
  try
  {
    var syncCtx = new SingleThreadSynchronizationContext();
    SynchronizationContext.SetSynchronizationContext(syncCtx);

    var t = func();
    t.ContinueWith(
      delegate { syncCtx.Complete(); }, TaskScheduler.Default);

    syncCtx.RunOnCurrentThread();

    t.GetAwaiter().GetResult();
  }
  finally { SynchronizationContext.SetSynchronizationContext(prevCtx); }
}
```

That's it. With our solution now available, I can change the Main method of my demo console app from:

```
static void Main()
{
  DemoAsync().Wait();
}
```

to instead use our new AsyncPump.Run method:

```
static void Main()
{
  AsyncPump.Run(async delegate
  {
    await DemoAsync();
  });
}
```

When I then run my app again, this time I get the following output:

```
[1, 10000]
Press any key to continue . . .
```

As you can see, all of the continuations have run on just one thread, the main thread of my console app.

The AsyncPump sample class described in this post is available as an attachment to this post.

AsyncPump.cs

Recent Posts

Live Now on Developer Tools Blogs

Tags

Library Testing ThreadPool Tools Visual Studio Visual Studio 2010

Videos

Related Resources

Visual Studio Product Website

Visual Studio Developer Center

Archives

Tags .NET 4.5 Async Coordination Data Structures Parallel Extensions Task Parallel Library

Join the conversation

Add Comment

Jon Skeet

6 years ago

There's another kind of situation this will be particularly important in – unit testing. Just last week I was coming up against exactly this sort of issue – so between this and your next post, I should finally be able to make my "time machine" test helper work properly

Stephen Toub - MSFT

6 years ago

Thanks, Jon; I'm glad it's useful to you. And yes, this could definitely be applied in a unit testing scenario. One thing you'd want to be careful of, though (if it's relevant), is that this doesn't lead you into a false sense of security. For example, if you were testing a library function that could run in many different contexts, with this approach you'd only be doing your testing in a serialized context, one where potential races in your code-under-test would be less likely to show up. That might be reasonable, just something to keep in mind.

Michael K

6 years ago

Stephen, can you please elaborate on "Both SynchronizationContext and TaskScheduler are abstractions that represent a “scheduler”"?

What is the conceptual difference between them? Obviously someone thougth there should be two distinct "schedulers" like this involved.

Stephen Toub - MSFT

6 years ago

Hi Michael-

The primary difference is that SynchronizationContext is a general mechanism for working with delegates, whereas TaskScheduler is specific to and catered to Tasks (you can get a TaskScheduler that wraps a SynchronizationContext using TaskScheduler.FromCurrentSynchronizationContext). This is why awaiting Tasks takes both into account, first checking a SynchronizationContext (as the more general mechanism that most UI frameworks support), and then falling back to a TaskScheduler. Awaiting a different kind of object might choose to first use a SynchronizationContext, and then fall back to some other mechanism specific to that particular type.

John Michael Hauck

6 years ago

Very nicely illustrates the mechanism behind await. I should have read your article before posting this:

connect.microsoft.com/.../await-does-not-always-return-on-the-

-same-thread. Thanks again!

Stephen Toub - MSFT

6 years ago

John, I'm glad you've found the post helpful.

Andrew Arnott

6 years ago

Unfortunately it seems that BlockingCollection<T> throws an OperationCanceledException every time it completes (and thereby unblocks the worker thread that is dequeuing from it). This exception is thrown and caught within the framework so it only shows up as a first chance exception in the debugger. If the AsyncPump is called frequently, this can make the debugging experience painfully slow, and at the scale I'm seeing, will likely hurt runtime perf as well.

andreas_huber69@live.com

5 years ago

Shouldn't CreateCopy() be overridden as well? I'm asking because I have observed that this method is sometimes called, and it seems wrong to return a default-constructed SynchronizationContext, as the base class implementation does. In this case it seems "return this;" would be a good implementation?

Stephen Toub - MSFT

5 years ago

@andreas_huber69: Yes, technically CreateCopy should be overridden, but it should be overridden to return a new instance, not to return 'this'.

Sylvain

5 years ago

Hi Stephen.

Nice job!

A question remain though: if some code after await sends an ex-

ception, then it will propagate to workItem.Key(workItem.Value) and RunOnCurrentThread will exit...

I assume that it is normal to throw exceptions after await keywords, so there must be a way to catch the exception into the SynchronizationContext instance and tell that the Post(...) led to an exception...

Any idea?

Thanks.

Stephen Toub - MSFT

5 years ago

@Sylvain: I'm not understanding what the concern is. If the "async Task" method passed to Run incurs an exception, whether that exception comes before or after an await, the exception will be stored into the returned Task. Then that exception will propagate out of the call to t.GetAwaiter().GetResult(), by design.

Paul

4 years ago

Excellent solution, any idea how I would implement the AsyncPump solution when awaiting a Task in another Task?

E.g in the below the Task RunLoadTestAsync will run synchronously?

```
public async Task StartTestsAsync()

{

AsyncPump.Run(async () =>

{

await RunLoadTestAsync();

});

...

public async Task RunLoadTestAsync()
```

{

...

I cant make AsyncPump.Run an async method can I?

Stephen Toub - MSFT

4 years ago

@Paul: I'm not sure what you're trying to accomplish. Can you elaborate? Why wouldn't you just await the inner task rather than using this approach at all?

Dmytro Shevchenko

4 years ago

Why doesn't this approach work with HttpClient? Here's my code:

```
static async Task<string> GetPageSourceAsync()

{

  string result;

  Console.WriteLine(Thread.CurrentThread.ManagedThreadId);

  using (var client = new HttpClient())

  {

    result = await client.Get-
StringAsync("http://google.com&amp;quot;);

  }

  Console.WriteLine(Thread.CurrentThread.ManagedThreadId);

  return result;

}
```

When I run the above code using the AsyncPump.Run() method, I am still getting two different thread IDs in the console output.

@Dmytro Shevchenko: How are you invoking your method? I just tried it with:

```
public static void Main()
{
    AsyncPump.Run(() => GetPageSourceAsync());
}
```

and it works correctly for me.

Sylvain

4 years ago

Hi.

I wonder if the initial code could be more alike to:

```
var t = FooAsync();
t.ContinueWith(delegate
{
    RestOfMethod();
}, SynchronizationContext.Current==null ? TaskScheduler.Current :
TaskScheduler. FromCurrentSynchronizationContext());
```

I've run a test where I make an async call while both SynchronizationContext.Current and TaskScheduler.Current are set to custom objects, and the custom TaskScheduler gets no breakpoint (only the custom SynchronizationContext does).

Stephen Toub - MSFT

4 years ago

@Sylvian: My example was only approximate. If there's a custom SynchronizationContext, the continuation will run there. Otherwise, if there's a custom TaskScheduler, the continuation will run there. That's what my example highlights: "RestOfMethod()" will be invoked on TaskScheduler.Current if there wasn't a SynchronizationContext present (TaskScheduler.Current returns TaskScheduler.De-

fault if no custom scheduler is in use), and "RestOfMethod()" will be posted to the SynchronizationContext if there was one.

David

4 years ago

I was trying to use the AsyncPump in my VS2012 solution targetting .Net 4.0 together with the Microsoft.Bcl.Async libraries. The AsyncPump was not working (still using different threads) until I tried to add the override implementation of CreateCopy – like you suggested with "return new SingleThreadSynchronizationContext();". That made the console application hang forever. I then tried "return this;" like andreas_huber69 suggested and it worked. I also tried all three possibly ways (no override/calling base implementation, return new instance, return this) in a separate solution targetting .Net 4.5 and in any case the AsyncPump worked fine as it should. Do you have an idea why? Furthermore, why did you state to return a new instance instead of of "this" in the CreateCopy override? Thanks for the article and for sharing your knowledge.

Robin Caron

4 years ago

Please update the sample to implement CreateCopy or add an addendum to your blog post pointing out the need for an implementation. As it stands now at best others read all the comments and see that a CreateCopy implementation is needed or at worse spend multiple hours to understand why the current code does not works.

Note that as @Andreas and @David have pointed out without reworking the current code the only possible implementation of CreateCopy is "return this".

korggy

4 years ago

Any chance of publishing the AsyncPump as a NuGet package? I've used it in several projects and every time I need it I wind up coming back to this blog post. Would be nice to be able to simply pull it in from NuGet when I needed it. Ah well, maybe I'm just too lazy! ■

@korggy: Thanks for the suggestion. I don't currently have plans to release a NuGet package with this, but you or anyone else should feel free to do so.

Mr Kite

3 years ago

Is it somehow possible to use WaitAll/WhenAll with this class? Or would that require adding a new RunAll method?

It seems strange to me that there is no simple, built in option to just make everything run on one thread if that is what you desire, I thought part of the advantage of async was that you no longer had to worry about thread synchronization.

Jonathan

3 years ago

Thanks for your post. I did some tests to validate that a Task continuation is actually scheduled on the thread pool. My tests were not conclusive so I posted the following question on Stack-Overflow: stackoverflow.com/.../26942338

I received an interesting answer and I was wondering if you could you maybe comment on this? Thanks!

Andre

3 years ago

Hi,

I'm learning a lot from your posts, and I have a question:

What is the purpose of t.GetAwaiter().GetResult(); and what are the consequences if I remove this line?

I thought that t will be completed and continue with delegate { syncCtx.Complete(); }, TaskScheduler.Default);

Thank you.

Stephen Toub - MSFT

3 years ago

@Andre: The call to GetResult() will propagate any exceptions that may have occurred in the Task t.

Andre

3 years ago

I have a weird case; I'm using the AsyncPump in a GUI app but in a separate thread (created using Task.StartNew...)

The created Task will run for a very long time, many minutes.

The problem is that in extremely rare cases I have an exception thrown from the Post method:

"The BlockingCollection<T> has been marked as complete with regards to additions". But how is that possible?

The task completed, that's why the BlockingCollection is marked as complete. Who is posting to the BlockingCollection?

```
        updateTask = Task.Factory.StartNew(() =>
        {
            AsyncPump.Run(async () =>
            {
                while (!cancellationToken.IsCancellationRequested)
                {
                    while (!ActionQueue.Instance.IsEmpty)
                    {
                        Action action;
                        if (ActionQueue.Instance.TryDequeue(out ac-
tion))
                        {
                            action();
                        }
```

```
}

Update();

....

....

await Task.Delay(delta);

....
```

The ActionQueue is a ConcurrentQueue; I used it to communicate between the UI thread and the updateTask.

Thank you.

Daniel Marbach

3 years ago

Hi Stephen,

I'm using this to execute a Func<Task> inside an IEnlistment implementation which is added to Transaction.Current.EnlistVolatile(). So far it works smoothly. My question is can the calling thread (the one who creates the TransactionScope with AsyncFlow enabled and awaiting an IO operation inside the TxScope) still do other work while the IO operation is inside the IO completion port or thus this block the calling thread? Thanks for clarifying

Daniel

Stephen Toub - MSFT

3 years ago

@Daniel Marbach: As written this is blocking the thread calling AsyncPump.Run. You could certainly tweak it, though.

Stephen Toub - MSFT

3 years ago

@Andre: I'd suggest doing some tracing inside of the Post method to capture a stack trace and see where the call is coming from. It's likely that something inside of your Update(); call or your "..." is capturing the current SynchronizationContext and posting

back to it later on.

Daniel Marbach

3 years ago

Steven,

Could you elaborate how we could tweak it?

here is the code which is using it

github.com/.../TransactionExtensions.cs

github.com/.../SendResourceManager.cs

I don't know any other way to support TxScope with async inside the TxScope besides than implementing a custom transaction scope which fully leverages async. A bit painful the whole story.

So the summary is when I'm using AsyncPump the thread calling tx.Complete() is blocked until the internal IO operations is completed therefore I'm loosing the benefit of Async/Await there. Right?

Thanks for your help

Daniel

Andre

3 years ago

Hi Stephen,

The problem was that we have a fire and forget async void Method called inside the update.

Stephen Toub - MSFT

3 years ago

@Andre:

Glad you solved it.

@Daniel Marbach:

Just as one example, instead of taking a single Func<Task>, take

two Func<Tasks>s, invoke them both, and then instead of doing t.ContinueWith(...), do Task.WhenAll(t1, t2).ContinueWith(...). You could of course do the same with three tasks, or four task, or an IEnumerable<Tasks>. Etc.

Daniel Krikun

3 years ago

Excellent post!

Regarding the original console application code sample, won't it cause a data race on the e. g. dictionary object? That is, if the continuations are executed on the thread pool, does it mean they could potentially run in parallel and thus cause a data race etc.?

Thanks

Stephen Toub - MSFT

3 years ago

@Daniel Krikun: There's only one continuation at a time here. The dictionary will not be accessed concurrently.

Raphael

3 years ago

Wow, excellent article.

That was really appreciated.

Thanks.
