Asynchronous Programming in
C# and Visual Basic

Mads Torgersen, Microsoft

October 2010

Contents

[Why asynchronous? 2](#_Toc275330899)

[How bad is it? 3](#_Toc275330900)

[A new approach 7](#_Toc275330901)

[Tasks 8](#_Toc275330902)

[The Task-based asynchronous pattern 9](#_Toc275330903)

[Async methods and awaiting 9](#_Toc275330904)

[Event handlers and void returning async methods 11](#_Toc275330905)

[Asynchronous lambda expressions 12](#_Toc275330906)

[Contexts 13](#_Toc275330907)

[Beyond asynchronous: easing into parallelism 13](#_Toc275330908)

When your user interface is unresponsive or your server doesn’t scale, chances are you need your code to be more asynchronous. With today’s .NET Framework and language features, though, that is easier said than done.

The Microsoft Visual Studio Async CTP proposes a new language feature in C# and VB, and a new framework pattern to go with it, that will make asynchronous programming similar to – and about as straightforward as –synchronous programming.

This document describes the limitations of the current callback based programming model for asynchrony, and describes the sweeping new opportunities offered by the framework and language features proposed in the Async CTP. It is intended as an overview, and further details can be found in specifications and detailed documents also included in the Async CTP.

|  |
| --- |
| ***Async CTP:*** APIs and language features are described herein as they are intended to be. For technical and scheduling reasons the Async CTP differs in certain ways, which are called out in the text. |

# Why asynchronous?

For decades programming with remote resources has presented a conundrum. As the level of abstraction in “local” programming has been steadily rising, there has been a push for transparency of remote operations – they should look just like local ones, so that a developer doesn’t need to grapple with conceptual overhead, architectural impedance mismatch and leaky abstractions.

The problem is that remote operations *are* different from local ones. They have orders of magnitude more latency even at the best of times, may fail in new ways or simply never come back, depend on a variety of external factors beyond the developer’s control or even perception, etc. So while they *can* be represented like “just method calls,” it is not desirable to do so because the developer is left without handles to manage the special conditions arising from their remoteness – managing cancellation and timeouts, preserving threading resources during blocking waits, predicting and handling threats to responsiveness, etc.

On .NET we have not ignored this challenge. In fact we have not just one but *several* patterns for how to do *asynchronous programming*; that is, dealing with I/O and similar high latency operations without blocking threads. Most often there is both a synchronous (i.e. blocking transparently) and an asynchronous (i.e. latency-explicit) way of doing things. The problem is that these current patterns are very disruptive to program structure, leading to exceedingly complex and error prone code or (more commonly) developers giving up and using the blocking approach, taking a responsiveness and performance hit instead.

The goal should be to bring the asynchronous development experience as close to the synchronous paradigm as possible, without letting go of the ability to handle the asynchrony-specific situations. Asynchrony should be explicit and non-transparent, but in a very lightweight and non-disruptive manner. Composability, abstraction and control structures should all work as simply and intuitively as with synchronous code.

This is the goal of the features of the Async CTP.

# How bad is it?

The problem is best understood in the common scenario of a UI that has just one thread to run all its user interface code on, but applies equally in, for example, server scenarios where thread resources may be a scaling bottleneck and having thousands of threads spend most of their time doing nothing is a bad strategy.

A client app that doesn’t react to mouse events or update the display for user-recognizable periods of time is likely the result of code holding on to the single UI thread for far too long. Maybe it is waiting for network IO or maybe it is performing an intensive computation. Meanwhile, other events just can’t get processing time, and the user-perceived world grinds to a halt. What’s a more frustrating user experience than losing all contact with an app that is “busy” standing still, staring down a pipe for a response that may be seconds away?

Easy to say, hard to fix. For years the recommended approach to these issues has been *asynchrony*: don’t wait for that response. Return as soon as you issue the request, letting other events take place in the meantime, but have the eventual response *call you back* when it arrives so that you can process the result as a separate event. This is a great approach: your UI thread is never blocked waiting, but is instead blazing through small, nimble events that easily interleave and never have to wait long for their turn.

The problem: Asynchronous code totally blows up your control flow. The *call you back* part needs a *callback* – a delegate describing what comes after. But what if you wanted to “wait” inside a while loop? An if statement? A try block or using block? How do you then describe “what comes after”?

Look at this simple example:

|  |
| --- |
| public int SumPageSizes(IList<Uri> uris) {    int total = 0;    foreach (var uri in uris) {        statusText.Text = string.Format("Found {0} bytes ...", total);        var data = new WebClient().DownloadData(uri);        total += data.Length;    }    statusText.Text = string.Format("Found {0} bytes total", total);    return total;} |
| Public Function SumPageSizes(uris As IList(Of Uri)) As Integer    Dim total As Integer = 0    For Each uri In uris        statusText.Text = String.Format("Found {0} bytes ...", total)        Dim data = New WebClient().DownloadData(uri)        total += data.Length    Next    statusText.Text = String.Format("Found {0} bytes total", total)    Return totalEnd Function |

The method downloads a number of URI’s, totaling their sizes and updating a status text along the way.

Clearly this method doesn’t belong on the UI thread because it may take a very long time to complete, while holding up the UI completely. Just as clearly it *does* belong on the UI thread because it repeatedly updates the UI. What to do?

We can put it on a background thread, making it repeatedly “post” back to the UI thread to do the UI updates. That seems wasteful in this case, since a thread will be occupied spending most of its time just waiting for downloads, but sometimes it is really the only thing you can do. In this case, however, WebClient offers an asynchronous version of DownloadData – DownloadDataAsync – which returns promptly, and then fires an event – DownloadDataCompleted – when it is done. This allows us to write an *asynchronous* version of our method that splits it up into little callbacks and runs the next one on the UI thread whenever the download initiated by the previous one completes. Here’s a first attempt:

|  |
| --- |
| public void SumPageSizesAsync(IList<Uri> uris) {    SumPageSizesAsyncHelper(uris.GetEnumerator(), 0);}private void SumPageSizesAsyncHelper(IEnumerator<Uri> enumerator, int total) {    if (enumerator.MoveNext()) {        statusText.Text = string.Format("Found {0} bytes ...", total);        var client = new WebClient();        client.DownloadDataCompleted += (sender, e) => {            SumPageSizesAsyncHelper(enumerator, total + e.Result.Length);        };        client.DownloadDataAsync(enumerator.Current);    }    else {        statusText.Text = string.Format("Found {0} bytes total", total);        enumerator.Dispose();    }} |
| Public Sub SumPageSizesAsync(uris As IList(Of Uri))    SumPageSizesAsyncHelper(uris.GetEnumerator(), 0)End SubPrivate Sub SumPageSizesAsyncHelper(        enumerator As IEnumerator(Of Uri), total As Integer)    If enumerator.MoveNext() Then        statusText.Text = String.Format("Found {0} bytes ...", total)        Dim client = New WebClient()        AddHandler client.DownloadDataCompleted,            Sub(sender, e)                SumPageSizesAsyncHelper(enumerator, total + e.Result.Length)            End Sub        client.DownloadDataAsync(enumerator.Current)    Else        statusText.Text = String.Format("Found {0} bytes total", total)        enumerator.Dispose()    End IfEnd Sub |

Already this is bad. We have to break up the neat foreach loop and manually get an enumerator. Each call to the private helper method hooks up an event handler for the completion of its download, that will eventually call the private helper again for the next element – if any. The code looks recursive instead of iterative. Still you may squint and be able to discern the intent of this code. But we are not nearly done yet.

The original code returned the total as well as display it. Our new asynchronous version returns to its caller way before the total has even been computed. How do we get a result back to our caller? The answer is: our caller must provide a callback to *us* – which we can then invoke with the total when ready. The caller must in turn have its code restructured so that it consumes the total in a callback instead of as a return value.

And what about exceptions? The original code said nothing about exceptions; they were just silently propagated to the caller. In the async case, though, exceptions will arise *after* we returned to the caller. We must extend the callback from the caller to also tell it about exceptions, and we have to explicitly propagate those, wherever they may arise.

Together, these requirements will further clutter the code:

|  |
| --- |
| public void SumPageSizesAsync(IList<Uri> uris, Action<int, Exception> callback) {    SumPageSizesAsyncHelper(uris.GetEnumerator(), 0, callback);}private void SumPageSizesAsyncHelper(IEnumerator<Uri> enumerator, int total,     Action<int, Exception> callback) {    try {        if (enumerator.MoveNext()) {            statusText.Text = string.Format("Found {0} bytes ...", total);            var client = new WebClient();            client.DownloadDataCompleted += (sender, e) => {                if (e.Error != null)  { enumerator.Dispose(); callback(0, e.Error); }                else SumPageSizesAsyncHelper(                    enumerator, total + e.Result.Length, callback);            };            client.DownloadDataAsync(enumerator.Current);        }        else {            statusText.Text = string.Format("Found {0} bytes total", total);            enumerator.Dispose();            callback(total, null);        }    }    catch (Exception ex) { enumerator.Dispose();        callback(0, ex);    }} |
| Public Sub SumPageSizesAsync(        uris As IList(Of Uri),  callback As Action(Of Integer, Exception))    SumPageSizesAsyncHelper(uris.GetEnumerator(), 0, callback)End SubPrivate Sub SumPageSizesAsyncHelper(        enumerator As IEnumerator(Of Uri),        total As Integer,        callback As Action(Of Integer, Exception))    Try        If enumerator.MoveNext() Then            statusText.Text = String.Format("Found {0} bytes ...", total)            Dim client = New WebClient()            AddHandler client.DownloadDataCompleted,                Sub(sender, e)                    If e.Error IsNot Nothing Then enumerator.Dispose()                        callback(0, e.Error)                    Else                        SumPageSizesAsyncHelper(                            enumerator, total + e.Result.Length, callback)                    End If                End Sub            client.DownloadDataAsync(enumerator.Current)        Else            statusText.Text = String.Format("Found {0} bytes total", total)            enumerator.Dispose()            callback(total, Nothing)        End If    Catch ex As Exception enumerator.Dispose()        callback(0, ex)    End TryEnd Sub |

Is this code now correct? Did we expand the foreach statement correctly, and propagate all exceptions? In fact when you look at it can you tell what it does?

Unlikely. And this corresponds to a synchronous method with just *one* blocking call to replace with an asynchronous one (DownloadData), and *one* layer of control structure around it (the foreach loop). Imagine trying to compose more asynchronous calls, or having more complex control structure! And we haven’t even started on the *callers* of SumPageSizesAsync!

The real problem here is that we can no longer describe the logical flow of our method using the control flow constructs of the language. Instead of describing the flow, our program becomes about describing the *wiring up* of the flow. “Where to go next” becomes a matter not of the execution of a loop or conditional or try-block, but of which callback you installed.

Hence, making your code asynchronous with today’s tools is extremely ungrateful: After a lot of hard work the result is unappealing, hard to read and likely full of bugs.

# A new approach

With the proposed new features, the asynchronous version of the code will instead look like this:

|  |
| --- |
| public async Task<int> SumPageSizesAsync(IList<Uri> uris) {    int total = 0;    foreach (var uri in uris) {        statusText.Text = string.Format("Found {0} bytes ...", total);        var data = await new WebClient().DownloadDataAsync(uri);        total += data.Length;    }    statusText.Text = string.Format("Found {0} bytes total", total);    return total;} |
| Public Async Function SumPageSizesAsync(uris As IList(Of Uri)) As Task(Of Integer)    Dim total As Integer = 0    For Each uri In uris        statusText.Text = String.Format("Found {0} bytes ...", total)        Dim data = Await New WebClient().DownloadDataAsync(uri)        total += data.Length    Next    statusText.Text = String.Format("Found {0} bytes total", total)    Return totalEnd Function |

Notice first how similar it is to the synchronous code. The highlighted parts are added, the rest stays the same. The control flow is completely unaltered, and there are no callbacks in sight. That doesn’t mean that there *are* no callbacks, but the compiler takes care of creating and signing them up, as we shall see.

The method is asynchronous by virtue of returning a Task<int> instead of an int. The Task and Task<T> types are in the framework today, and are good at representing ongoing work. The caller of SumPageSizesAsync can later use the returned Task<int> to inquire whether the work is complete, wait for the int result synchronously or sign up callbacks to get it when it is ready. So instead of taking a callback as a parameter, an asynchronous method now returns a *representation* of the ongoing work.

The asynchronous method has no extra parameters. By convention and to distinguish it from its synchronous counterpart (if we keep that in our code) we append “Async” to the method name.

The method is also marked as async. This means that the method body is compiled specially, allowing parts of it to be turned into callbacks, and automatically creating the Task<int> that is returned.

Here’s how that works: Inside of the method body we call another asynchronous method, DownloadDataAsync. This quickly returns a Task<byte[]> that will eventually complete when the downloaded data is available. However, we don’t want to do anything else until we have that data, so we immediately await the task, something that is only allowed inside of async methods.

|  |
| --- |
| ***Async CTP:*** The Task-based version of DownloadData is called DownloadData**Task**Async to avoid conflict with an existing method on WebClient. |

At first glance the await keyword *looks* like it blocks the thread until the task is complete and the data is available, but it doesn’t. Instead it signs up *the rest of the method* as a callback on the task, and immediately returns. When the awaited task eventually completes, it will invoke that callback and thus resume the execution of the method right where it left off!

By the time execution reaches the return statement, we have already been suspended and resumed several times by await’ing in the foreach loop, and have returned to the original caller long ago. We returned them not a result, because we didn’t have one yet (we were still in the process of computing the total) but a Task<int> that *they* could await if and when they wanted to. The effect of the return statement is to *complete* that task, so that whoever is looking at it – e.g. by await’ing it – can now get the result.

In the following we will go into details with the different components of this approach.

# Tasks

The Task and Task<TResult> types are already in the .NET Framework 4. A Task represents an ongoing activity, which *may* be CPU intensive work running on a separate thread, but may also represent an I/O operation, for example an outstanding response for an internet request. Task represents an activity without a result, and Task<TResult>, which derives from Task, represents an activity with a result of type TResult.

Tasks are often used to represent CPU-bound work running on a separate thread. Saying

|  |
| --- |
| Task<double> task = Task.Run(() => LongRunningComputation(x, y, z)); |
| Dim task As Task(Of Double) = Task.Run(Function() LongRunningComputation(x, y, z)) |

is an easy way to schedule the work in the lambda on a thread pool thread, immediately returning a task that will complete when the work is done.

|  |
| --- |
| ***Async CTP:*** Since the CTP installs on top of .NET 4, it is not possible for the CTP to add new members to existing types. While instance members are mimicked by extension methods, there is no way to add static members such as Task.Run. Instead these are offered on a “temporary” type called TaskEx. |

Manually creating a Task that does not run on a separate thread is also easy:

|  |
| --- |
| var tcs = new TaskCompletionSource<double>();return tcs.Task;…tcs.TrySetResult(result); |
| Dim tcs = New TaskCompletionSource(Of Double)()Return tcs.Task…tcs.TrySetResult(result) |

Once you have a TaskCompletionSource you can immediately hand out its associated Task, and complete it later when the work is done and the result is ready. The tasks generated by the async language feature are of the latter sort – they don’t occupy a thread of their own.

Tasks can represent exceptional as well as successful completion. If a Task ends up in a faulted state, the relevant Exception will be available, and awaiting the Task will result in its exception being propagated.

The easiest way to consume a Task is to await it in an async method, but there are also plenty of more “manual” ways to consume it – synchronously or asynchronously. These are outside the scope of this document (you can learn more about tasks in the MSDN documentation at <http://msdn.microsoft.com/en-us/library/dd537609.aspx>).

# The Task-based asynchronous pattern

Part of this proposal is a shift to a new model for what asynchronous methods should look like – the *Task-based Asynchronous Pattern* (TAP). Current patterns offer one method for launching the operation and another method or event for obtaining the result. By returning a Task or Task<T> instead, as in the example above, only one method is needed, and all further interaction with the future result is done through the Task.

There are a couple of other conventions around the TAP, including how to handle cancellation and progress. These are further described in the *Task-based Asynchronous Pattern* document included in the Async CTP.

# Async methods and awaiting

It is important to understand that async methods like SumPageSizesAsync do *not* run on their own thread. In fact if you write an async method without any await’s in it, it will be completely synchronous:

|  |
| --- |
| public async Task<int> TenToSevenAsync() {    Thread.Sleep(10000);    return 7;} |
| Public Async Function TenToSevenAsync() As Task(Of Integer)    Thread.Sleep(10000)    Return 7End Function |

If you call this method you will be blocked for ten seconds and then get an already completed Task<int> back with a result of 7. This is probably not what you expect, and async methods without await’s do yield a warning because they are almost never what you want.

Only when an async method gets to the first await will it return to its original caller. Even then, in fact, it won’t return until it await’s a task that is not yet complete. This means that you should write your async methods so that they do not do a lot of work, or perform blocking calls, before their first await – or indeed between await’s. Async methods are for doing things that don’t need a lot of thread time. If you want to get intensive work or unavoidable blocking calls off your thread – e.g. off the UI thread – you should explicitly put them on the thread pool using Task.Run.

Instead the right way to write the above method uses await:

|  |
| --- |
| public async Task<int> TenToSevenAsync() {    await Task.Delay(10000);    return 7;} |
| Public Async Function TenToSevenAsync() As Task(Of Integer)    Await Task.Delay(10000)    Return 7End Function |

Task.Delay is essentially the asynchronous version of Thread.Sleep: it returns a Task which completes after the specified amount of time. Given that, let us look in detail at how this very simple async method runs.

|  |
| --- |
| ***Async CTP:*** The Task.Delay method is instead offered on the TaskEx type. |

When the async method is first invoked, a Task<int> is immediately created. The method then executes up to the first (and in this case only) await. This means evaluating the “operand” to await, i.e. the call to Task.Delay, which quickly returns us an as-yet uncompleted Task.

Now we await that Task. This involves a number of steps, which are undertaken in an intricate collaboration between the compiler-generated code and the Task type:

1. We check to see if the Task is already completed. If so, there’s nothing to wait for: we can just keep going. In this example, however, it is reasonable to assume that it is not yet completed.
2. We capture the *context* we are running in. To a loose approximation this means a representation of *where* we are running. (Typically this is represented either as a SynchronizationContext or as a TaskScheduler, but you probably didn’t want to know that). Let us say that we were called on the UI thread: the context then represents that fact.
3. We now sign up a callback to the Task, saying “when you complete, please do ‘this’.” We’ll see in a bit what ‘this’ actually does.
4. Finally we return to the caller. Since this is the first await, we are returning to the original caller of TenToSevenAsync, and we make sure to return them the Task<int> that we started out by creating.
5. About ten seconds later the Task we got from Task.Delay completes. It checks to see if it had any callbacks registered in the meantime. Indeed it has ours, which it calls.
6. This probably happens on some kind of system thread that has to do with timers – a thread we don’t care about and definitely don’t want to be resumed on. However, all the callback does here is to ask the *context* that we captured before to resume the method execution *there*. So we quickly get off of the “random” thread that resumed us.
7. The context schedules the resumption of the method according to its specific scheduling semantics. For the UI context this means queuing up the work in its message queue, from where it soon gets pumped onto the UI thread to be executed there.
8. Back on the UI thread, the compiler-generated resumption code – often known as the *continuation* – knows how to jump into the middle of the execution of our method, right where we left off ten seconds before. In fact the compiler turned the async method into a state machine object just so that it would be able to do this. (This transformation is similar to what is done to implement iterators in C#.)
9. The last action of the await expression is to consume the await’ed task’s result. In this case the task doesn’t produce a result, but we still check to see if the task was faulted – i.e. ended up completing with an exception. If that is the case, the exception is thrown again from the await expression.

We are now back in our method’s code following the await expression, and next up is the return statement, which also has special meaning in an async method. While it *does* return (in our case to the message pump that put us on the UI thread), it doesn’t return to our *original* caller: we already did that previously! Instead the return statement completes the already-returned Task<int> with a final result: the value seven.

The execution of the method is now finally complete. If the calling code signed up on the Task<int> in the meantime – e.g. by await’ing it – these callbacks in turn will now get invoked, and the cycle continues.

The execution of await looks – and is – somewhat complex. While the observable result is simple (you “wait” for something without occupying a thread), there is quite a bit of work going on under the hood to make it work. It is worth remembering that async methods are an alternative to having long-blocking calls. In comparison to the cost that such blocking incurs, the overhead of the bookkeeping involved in executing async methods is typically insignificant.

Further details about the syntax and semantics of the async language feature can be found in the C# and Visual Basic Language Specifications included in the Async CTP.

# Event handlers and void returning async methods

Async methods can be built from other async methods using await. But where does the asynchrony end? What kind of top level synchronous methods call asynchronous ones to start with?

In a client application the typical answer is that asynchronous operations are set off by events. The user clicks a button, and an asynchronous activity is set in motion that keeps trickling little chunks of work onto the UI thread in between awaits, until it is finally done. The event itself does not care when the asynchronous operation is done – in fact no-one does. It is what we call “fire and forget.”

To accommodate this pattern, async methods can be explicitly written to be fire and forget – by returning void instead of Task or Task<TResult>. This lets the method have a signature that allows it to be signed up directly as an event handler. When a void async method runs, no Task is produced and returned, and it is therefore not possible for the caller to track its completion.

|  |
| --- |
| private async void sumButton\_Click(object sender, RoutedEventArgs e) {    sumButton.IsEnabled = false;    await SumPageSizesAsync(GetUrls()));    sumButton.IsEnabled = true;} |
| Private Async Sub sumButton\_Click(sender As Object, e As RoutedEventArgs)    sumButton.IsEnabled = False    Await SumPageSizesAsync(GetUrls())    sumButton.IsEnabled = TrueEnd Sub |

Often the UI needs to change for the duration of an operation caused by a user event. In this case for instance, the button clicked is temporarily disabled. The very natural and symmetric pattern of just enclosing the core operation in the code to change the UI and change it back is possible now only because of await.

# Asynchronous lambda expressions

Not only methods but also lambda expressions can be async. Among other things this allows event handlers to be anonymous:

|  |
| --- |
| sumButton.Click += async (sender, e) => {    sumButton.IsEnabled = false;    await SumPageSizesAsync(GetUrls());    sumButton.IsEnabled = true;}; |
| AddHandler sumButton.Click, Async Function(sender, e)                                sumButton.IsEnabled = False                                Await SumPageSizesAsync(GetUrls())                                sumButton.IsEnabled = True                            End Function |

Another useful application of asynchronous lambdas is for running asynchronous work on a background thread. Task.Run has overloads that expect delegates of type Func<Task> and Func<Task<TResult>> so that you can easily express this.

|  |
| --- |
| ***Async CTP:*** Because of limitations in the overload resolution implementations the Run overloads expecting asynchronous lambdas would not be correctly selected. They are instead offered with a different name as TaskEx.RunEx. |

# Contexts

As described above, asynchronous methods depend on their context for the exact semantics of their resumption after an await. A UI context is usually single threaded, which means that only one asynchronous operation can be executing on it at any given time. This is very useful in a stateful environment such as a UI, because less care needs to be taken to prevent against races between different operations simultaneously manipulating the UI state. No locks are needed (nor would they be useful!); operations just need to make sure that they finish their changes in one go, before they relinquish the thread.

With asynchronous operations, relinquishing the thread occurs not just at the end but also at every await. This means that an asynchronous UI operation needs to be prepared for the global state to look different after each await. This is a good thing: remember that we are explicitly putting the await’s there *precisely* so that other operations can get in and have an effect!

Another context that is fundamentally single threaded is the one in which ASP.NET requests are handled. While multiple requests can be processed concurrently, each request is executed sequentially even in the presence of asynchronous operations. Again this simplifies the interaction between the different operations collaborating to respond to the request, by obviating the need for locking to protect against races on the state of the request.

Other contexts are not single threaded. Most notably the thread pool context is multithreaded – a shared queue will feed jobs to whichever thread pool thread happens to free up next. This goes also for the method resumptions – the *continuations* – that get scheduled to it once their awaited task completes. This means that an async method that gets called on a thread pool thread may resume on different thread pool threads after each await. This is usually not a problem, but means that it should not rely on thread local state to remain accessible across await’s. In particular, locks are not to be trusted, and the compilers explicitly disallow await expressions in lock statements (SyncLock in Visual Basic).

Another aspect of the context is that void returning async methods will let the context know when they start and when they complete. Most contexts ignore this, but ASP.NET will use that to keep a count of ongoing asynchronous operations. That way it can hold back on moving on to the next phase in the pipeline until all top-level asynchronous operations –the void ones – are completed.

It is possible to create your own context by deriving from SynchronizationContext or TaskScheduler but it is not recommended except for very advanced scenarios where you really need to control your own scheduling.

# Beyond asynchronous: easing into parallelism

The asynchronous framework and language features make it easy to take the step from synchronous to asynchronous sequential code. Moreover, this first step often enables further gradual changes to make your code more parallel.

Let’s return to our original SumPageSizesAsync example once more. While it is extremely useful to have the downloads be async so that our UI becomes responsive, we are still sequentially downloading one page at a time. Surely it is time to go “parallel” – not in the CPU-parallel multi-threaded sense, but by getting all those downloads started at once and using the parallelism of the web to “work” on them all simultaneously.

The Task type offers easy help here: Given a collection of tasks, you can call Task.WhenAll to obtain a task that completes when all of the constituent tasks have completed. Ignoring UI updates for the moment we can write our method like this:

|  |
| --- |
| public async Task<int> SumPageSizesAsync(IList<Uri> uris) {    var tasks = from uri in uris select new WebClient().DownloadDataAsync(uri);    var data = await Task.WhenAll(tasks);    return await Task.Run(() => data.Sum(s => s.Length));} |
| Public Async Function SumPageSizesAsync(uris As IList(Of Uri)) As Task(Of Integer)    Dim tasks = From uri In uris Select New WebClient().DownloadDataAsync(uri)    Dim data = Await Task.WhenAll(tasks)    Return Await TaskEx.Run(Function() data.Sum(Function(s) s.Length))End Function |

First we use a LINQ query to get us a list of started tasks. Then we call WhenAll on the tasks and immediately await the resulting collection of strings. Finally we compute the result on a background thread (to keep the UI responsive in cases it takes a long time), await it and return it.

|  |
| --- |
| ***Async CTP:*** The WhenAll and WhenAny methods are instead offered on the TaskEx type. |

WhenAll and its cousin WhenAny (which completes when *any* of the input tasks is complete) are useful to orchestrate batches of similar jobs that occur in parallel. Sometimes the opportunity for parallelism is a little more intertwined with the program flow.

Consider this example, where we download movies based on a list of titles, and show them on the screen one at a time. We don’t want to eagerly download all the movies on the list up front – presumably they are big, and we also want to start playing the first one as soon as we can. But we can prevent most or all delay between the showing of the movies by downloading the next movie while the previous one is playing.

|  |
| --- |
| async void ShowMoviesAsync(string[] titles) {    Task playing = null;    foreach (var title in titles) {        var movie = await DownloadMovieAsync(title);        if (playing != null) await playing;         playing = PlayMovieAsync(movie);    }} |
| Async Sub ShowMoviesAsync(titles As String())    Dim playing As Task = Nothing    For Each title In titles        Dim movie = Await DownloadMovieAsync(title)        If playing IsNot Nothing Then Await playing        playing = PlayMovieAsync(movie)    NextEnd Sub |

Note that this code is “sloppy” in the sense that the last Task in playing never gets await’ed. Throwing away a Task unobserved is usually not a problem, but it does mean that any exception that occurs in that Task will never be handled. If your Tasks can throw exceptions that should not go unhandled, then you should make sure to await or otherwise observe them.