Chapter 20



Programming the Task-Based Asynchronous Pattern

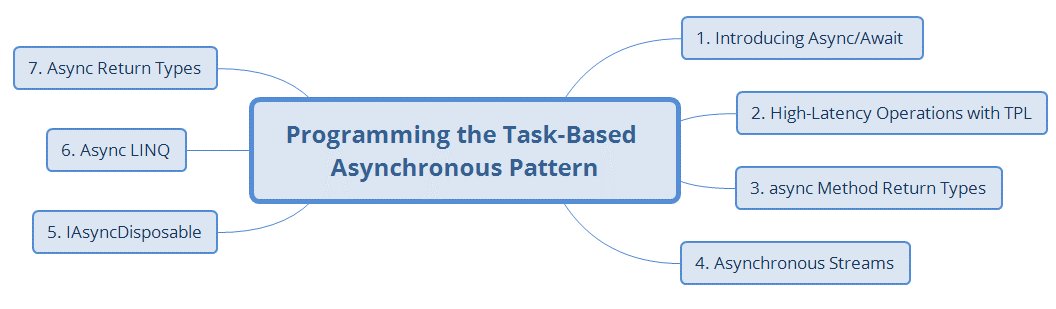
As we’ve seen in the last chapter, tasks provide an abstraction for the manipulation of asynchronous work than threads do. Tasks are automatically scheduled to the right number of threads, and large tasks can be composed by chaining together small tasks, just as large programs can be composed from multiple small methods.

However, there are some drawbacks to tasks. The principal difficulty with tasks is that they turn your program logic “inside out.” To illustrate this, we begin the chapter with a synchronous method that is blocked on an I/O-bound, high-latency operation—a web request. Next, we compare it to an asynchronous version prior to C# 5.0 and TAP. Lastly, we revise the same example by using C# 5.0 (and higher) and the async/await contextual keywords, demonstrating a significant simplification in authoring and reading of asynchronous code.

We finish the chapter with a look at asynchronous streams - a C# 8.0 introduced feature for defining and leveraging asynchronous iterators.

\*\*\*COMP: Insert 20mindmap

\*\*\*PRODUCTION: Please redraw to match mindmaps from other chapters.



Synchronously Invoking a High-Latency Operation

In Listing 20.1, the code uses a WebClient to download a web page and search for the number of times some text appears. If the operation fails, an exception is thrown.

Listing 20.1: A Synchronous Web Request

using System;

using System.IO;

using System.Net;

using System.Linq;

public class Program

{

public static void Main(string[] args)

{

if (args.Length == 0)

{

Console.WriteLine("ERROR: No findText argument specified.");

return;

}

string findText = args[0];

Console.WriteLine($"Searching for {findText}...");

string url = "http://www.IntelliTect.com";

if (args.Length > 1)

{

url = args[1];

// Ignore additional parameters

}

Console.Write(url);

Console.WriteLine("Searching...");

int textOccurrenceCount =

FindTextInWebUri(findText, url);

Console.WriteLine(textOccurrenceCount);

}

public static int FindTextInWebUri(

string findText, string url)

{

using WebClient webClient = new WebClient();

byte[] downloadData =

webClient.DownloadData(url);

return CountOccurencesInContent(

downloadData, findText);

}

private static int CountOccurencesInContent(byte[] downloadData, string findText)

{

int textOccurrenceCount = 0;

using MemoryStream stream = new MemoryStream(downloadData);

using StreamReader reader = new StreamReader(stream);

int findIndex = 0;

int length = 0;

do

{

char[] data = new char[reader.BaseStream.Length];

length = reader.Read(data);

for (int i = 0; i < length; i++)

{

if (findText[findIndex] == data[i])

{

findIndex++;

if (findIndex == findText.Length)

{

// Text was found

textOccurrenceCount++;

findIndex = 0;

}

}

else

{

findIndex = 0;

}

}

}

while (length != 0);

return textOccurrenceCount;

}

}

The logic in Listing 20.1 is relatively straightforward—using common C# idioms like try/catch blocks and return statements to describe the control flow. Given a WebClient, the code invokes the synchronous method DownloadData() to download webpage. Given the downloaded data, it then stores it in a MemoryStream and leverages a StreamReader's Read() method to retrieve a block of data and search it for the findText value.

The problem with this approach is, of course, that the calling thread is blocked until the I/O operation completes; this is wasting a thread that could be doing useful work while the asynchronous operation executes. For this reason, we cannot, for example, execute any other code, such as code that asynchronously indicates progress.

The Task-based Asynchronous Pattern with async and await

To address this problem, Listing 20.2's takes a similar approach but instead uses task-based asynchrony with the async/await feature.

Listing 20.2: An Asynchronous Web Request Using the Task-based Asynchronous Pattern

using System;

using System.IO;

using System.Net;

using System.Linq;

using System.Threading.Tasks;

public class Program

{

public static async Task Main(string[] args)

{

if (args.Length == 0)

{

Console.WriteLine( "ERROR: No findText argument specified.");

return;

}

string findText = args[0];

Console.WriteLine($"Searching for {findText}...");

string url = "http://www.IntelliTect.com";

if (args.Length > 1)

{

url = args[1];

// Ignore additional parameters

}

Console.Write(url);

Progress<DownloadProgressChangedEventArgs> progress =

new Progress<DownloadProgressChangedEventArgs>((value) =>

{

Console.Write(".");

}

);

// Using await later to elucidation.

Task<int> task =

FindTextInWebUriAsync(findText, url, progress);

Console.WriteLine( "Searching..." );

int textOccurrenceCount = await task;

Console.WriteLine(textOccurrenceCount);

}

public async static Task<int> FindTextInWebUriAsync(

string findText, string url,

IProgress<DownloadProgressChangedEventArgs>? progressCallback = null)

{

using WebClient webClient = new WebClient();

if (progressCallback is object)

{

webClient.DownloadProgressChanged += (sender, eventArgs) =>

{

progressCallback.Report(eventArgs);

};

}

byte[] downloadData =

await webClient.DownloadDataTaskAsync(url);

return await CountOccurrencesInContentAsync(

downloadData, findText);

}

private static async Task<int> CountOccurrencesInContentAsync(byte[] downloadData, string findText)

{

int textOccurrenceCount = 0;

using MemoryStream stream = new MemoryStream(downloadData);

using StreamReader reader = new StreamReader(stream);

int findIndex = 0;

int length = 0;

do

{

char[] data = new char[reader.BaseStream.Length];

length = await reader.ReadAsync(data);

for (int i = 0; i < length; i++)

{

if (findText[findIndex] == data[i])

{

findIndex++;

if (findIndex == findText.Length)

{

// Text was found

textOccurrenceCount++;

findIndex = 0;

}

}

else

{

findIndex = 0;

}

}

}

while (length != 0);

return textOccurrenceCount;

}

}

\*\*\*COMP: Insert “Begin 7.0” tab

Notice there are relatively small differences between Listing 20.1 and Listing 20.2, especially if you ignore the busy indicator which Listing 20.1 didn't support. First, we change the signature of the FindTextInWebUri method adding the new contextual async keyword, returning Task<int> (rather than just int) and inserting the “Async” suffix - a convention that indicates to callers that this method runs asynchronously. (We ignore the progressCallback parameter and the code that leverages it for the moment.) Any method decorated with the async keyword must return void, Task, Task<T>, ValueTask<T> (as of C# 7.0), or IAsyncEnumerable<T> (as of C# 8.0).[[1]](#footnote-2) In this case, since there is no data returned by the body of the method but we still want the capability of returning information about the asynchronous activity to the caller, FindTextInWebUri() returns Task. Finally, everywhere that we need to asynchronously wait for the result of an asynchronous method, we introduce the await operator.

In Main() the differences are similar. We invoke the new CountTextOccurencesOnPagesAsync() method with the await contextual keyword. We also modify Main()'s signature to return Task and include the keyword async.

\*\*\*COMP: Insert “End 7.0” tab

Fortunately, it is not too difficult to write a computer program that does these complex code transformations for you. The designers of the C# language realized this need would crop up, and they added such a capability to the C# 5.0 compiler. Starting with C# 5.0, you can rewrite the synchronous program given earlier into an asynchronous program much more easily using TAP; the C# compiler then does the tedious work of transforming your method into a series of task continuations.

Notice that nothing else changes between Listings 20.1 and 20.2. The asynchronous method versions seemingly still return the same data types as before—despite that each returns a Task<T>. This is not via some magical implicit cast, either. GetResponseAsync() is declared as follows:

public virtual Task<WebResponse> GetResponseAsync() { ... }

At the call site, we assign the return value to WebResponse:

WebResponse response = await webRequest.GetResponseAsync()

The async contextual keyword plays a critical role by signaling to the compiler that it should rewrite the expression into a state machine that represents all the control flow we saw in Listing 20.2 (and more).

To better explain the control flow, Table 19.2 shows each task in a separate column along with the execution that occurs on each task.

Table 19.2: Control Flow within Each Task

| Description | Main() Thread | FindTextInWebUriAsync() Task | Repeated ReadAsync() Task |
| --- | --- | --- | --- |
| Execution flows normally into Main and up through the first Console.Write(url) statement.  A call is made to FindTextInWebUriAsync (), so control flows into that method as it would normally | //...  Console.Write(url);  //...  // Using await later to elucidation.  Task<int> task =  FindTextInWebUriAsync(  findText,  url,  progress); |  |  |
| Instructions within FindTextInWebUriAsync() execute normally (still on the Main() thread). | int textOrruranceCount =  0;  using WebClient webClient =  new WebClient();  //... |  |  |
| The first await operator begins, generating a new Task on which a request to start downloading the URL content is started.  Assuming the download didn’t execute almost instantaneously, the control flow returns to Main() | byte[] downloadData =  await webClient.DownloadDataTaskAsync(  url);  Console.WriteLine(  "Searching..."); | await webClient.DownloadDataTaskAsync(  url);  //downloading begins...  // downloading ends |  |
| Once the DownloadDataTaskAsync() task completes, execution within the same task continues with the implicit assignment of the said task’s result (downloadData) to the variable downloadData, before proceeding to CountOccurrencesInContentAsync()to count the occurrences of findText.  Main() execution encounters await so execution instead waits for the asynchronous operation to complete | int textOccurrenceCount =  await task;    Console.WriteLine(  textOccurrenceCount); | int textOccurrenceCount =  await CountOccurrencesInContentAsync(  downloadData, findText); | await CountOccurrencesInContentAsync(  downloadData,  findText);  int textOrruranceCount = 0;  using MemoryStream stream =  new MemoryStream(  downloadData);  using StreamReader reader =  new StreamReader(stream);  // Additional code omitted  // for elucidation  //... |
| Once CountOccurrencesInContentAsync() completes it returns the count to FindTextInWebUriAsync() |  | return <result> |  |
| FindTextInWebUriAsync() returns the value to Main() where it is displayed in the console. | Console.WriteLine(  textOccurrenceCount); |  |  |

There are a couple of important misconceptions that the table helps to dismiss:

* **Misconception #1: A method decorated with the** async **keyword is automatically executed on a worker thread when called.** This is absolutely not true; the method is executed normally, on the calling thread, and if the implementation doesn’t await any incomplete awaitable tasks, it will complete synchronously on the same thread. It’s the method’s implementation that is responsible for starting any asynchronous work. Just using the async keyword does not change on which thread the method’s code executes. Also, there is nothing unusual about a call to an async method from the caller’s perspective; it is a method typed as returning a Task, it is called normally, and it returns an object of its return type normally. In Main(), for example, we FindTextInWebUriAsync() is assigned a task, just like it would for a non-async method. We later then await the task.
* **Misconception #2: The** await **keyword causes the current thread to block until the awaited task is completed.** That is also absolutely not true. If you want the current thread to block until the task completes, call the Wait() method, as we have already described. The await keyword evaluates the expression that follows it, which is usually of type Task or Task<T>, adds a continuation to the resultant task, and then immediately returns control to the caller. The creation of the task has started asynchronous work; the await keyword means that the developer wishes the caller of this method to continue executing its work on this thread while the asynchronous work is processed. At some point after that asynchronous work is complete, execution will resume at the point of control following the await expression.

In fact, the principal reasons why the async keyword exists in the first place are twofold. First, it makes it crystal clear to the reader of the code that the method that follows will be automatically rewritten by the compiler. Second, it informs the compiler that usages of the await contextual keyword in the method are to be treated as asynchronous control flow and not as an ordinary identifier.

\*\*\*COMP: Insert “Begin 7.0” tab

Starting with C# 7.1, it is possible to have an async Main method. As a result, Listing 20.3’s Main signature could be private static async Task Main(string[] args), and we could change the WriteWebRequestSizeAsync invocation to await WriteWebRequestSizeAsync(url). The disadvantage in this case would be that we could no longer have a timeout (task.Wait(100)).

\*\*\*COMP: Insert “End 7.0” tab

\*\*\*COMP: Insert “Begin 8.0” tab

Asynchronously Invoking a High-Latency Operation Using the TPL

To address this problem, Listing 20.3 takes a similar approach but instead uses task-based asynchrony with the TPL.

Listing 20.3: An Asynchronous Web Request

using System;

using System.IO;

using System.Net;

using System.Linq;

using System.Threading.Tasks;

using System.Runtime.ExceptionServices;

public class Program

{

public static void Main(string[] args)

{

string url = "http://www.IntelliTect.com";

if(args.Length > 0)

{

url = args[0];

}

Console.Write(url);

Task task = WriteWebRequestSizeAsync(url);

try

{

while(!task.Wait(100))

{

Console.Write(".");

}

}

catch(AggregateException exception)

{

exception = exception.Flatten();

try

{

exception.Handle(innerException =>

{

// Rethrowing rather than using

// if condition on the type

ExceptionDispatchInfo.Capture(

innerException)

.Throw();

return true;

});

}

catch(WebException)

{

// ...

}

catch(IOException )

{

// ...

}

catch(NotSupportedException )

{

// ...

}

}

}

private static Task WriteWebRequestSizeAsync(

string url)

{

StreamReader reader = null;

WebRequest webRequest =

WebRequest.Create(url);

Task task =

webRequest.GetResponseAsync()

.ContinueWith( antecedent =>

{

WebResponse response =

antecedent.Result;

reader =

new StreamReader(

response.GetResponseStream());

return reader.ReadToEndAsync();

})

.Unwrap()

.ContinueWith(antecedent =>

{

reader?.Dispose();

string text = antecedent.Result;

Console.WriteLine(

FormatBytes(text.Length));

});

return task;

}

static public string FormatBytes(long bytes)

{

string[] magnitudes =

new string[] { "GB", "MB", "KB", "Bytes" };

long max =

(long)Math.Pow(1024, magnitudes.Length);

return string.Format("{1:##.##} {0}",

magnitudes.FirstOrDefault(

magnitude =>

bytes > (max /= 1024)) ?? "0 Bytes",

(decimal)bytes / (decimal)max);

}

// ...

}

When Listing 20.3 executes, it prints periods to the console while the page is downloading. The result is that instead of simply printing four periods (....) to the console, Listing 20.3 is able to continuously print periods for as long as it takes to download the file, read it from the stream, and determine its size.

Unfortunately, this asynchrony comes at the cost of complexity. Interspersed throughout the code is TPL-related code that interrupts the flow. Rather than simply following the WebRequest.GetResponseAsync() call with steps to retrieve the StreamReader and call ReadToEndAsync(), the asynchronous version of the code requires ContinueWith() statements. The first ContinueWith() statement identifies what to execute after the WebRequest.GetResponseAsync(). Notice that the return statement in the first ContinueWith() expression returns StreamReader.ReadToEndAsync(), which returns another Task.

Without the Unwrap() call, therefore, the antecedent in the second ContinueWith() statement is a Task<Task<string>>, which alone indicates the complexity. As a result, it is necessary to call Result twice—once on the antecedent directly and a second time on the Task<string>.Result property antecedent.Result returned, with the latter blocking subsequent execution until the ReadToEnd() operation completes. To avoid the Task<Task<TResult>> structure, we preface the call to ContinueWith() with a call to Unwrap(), thereby shedding the outer Task and appropriately handling any errors or cancellation requests.

The complexity doesn’t stop with Tasks and ContinueWith(), however: The exception handling adds an entirely new dimension to the complexity. As mentioned earlier, the TPL generally throws an AggregateException exception because of the possibility that an asynchronous operation could encounter multiple exceptions. However, because we are calling the Result property from within ContinueWith() blocks, it is possible that inside the worker thread we might also throw an AggregateException.

As you learned earlier in the chapter, there are multiple ways to handle these exceptions:

1. We can add continuation tasks to all \*Async methods that return a task along with each ContinueWith() method call. However, doing so would prevent us from using the fluid API in which the ContinueWith() statements are chained together one after the other. Furthermore, this would force us to deeply embed error-handling logic into the control flow rather than simply relying on exception handling.

2. We can surround each delegate body with a try/catch block so that no exceptions go unhandled from the task. Unfortunately, this approach is less than ideal as well. First, some exceptions (like those triggered when calling antecedent.Result) will throw an AggregateException from which we will need to unwrap the InnerException(s) to handle them individually. Upon unwrapping them, we either rethrow them so as to catch a specific type or conditionally check for the type of the exception separately from any other catch blocks (even catch blocks for the same type). Second, each delegate body will require its own separate try/catch handler, even if some of the exception types between blocks are the same. Third, Main’s call to task.Wait() could still throw an exception because WebRequest.GetResponseAsync() could potentially throw an exception, and there is no way to surround it with a try/catch block. Therefore, there is no way to eliminate the try/catch block in Main that surrounds task.Wait().

3. We can ignore all exception handling from within WriteWebRequestSizeAsync() and instead rely solely on the try/catch block that surrounds Main’s task.Wait(). Given that we know the exception will be an AggregateException, we can have a catch for only that exception. Within the catch block, we can handle the exception by calling AggregateException.Handle() and throwing each exception using the Exception-Dispatch-Info object so as not to lose the original stack trace. These exceptions are then caught by the expected exception handlers and addressed accordingly. Notice, however, that before handling the Aggregate-Exception’s InnerExceptions, we first call AggregateException.Flatten(). This step addresses the issue of an AggregateException wrapping inner exceptions that are also of type AggregateException (and so on). By calling Flatten(), we ensure that all exceptions are moved to the first level and all contained AggregateExceptions are removed.

As shown in Listing 20.3, option 3 is probably the preferred approach because it keeps the exception handling outside the control flow for the most part. This doesn’t eliminate the error-handling complexity entirely; rather, it simply minimizes the occasions on which it is interspersed within the regular control flow.

Although the asynchronous version in Listing 20.2 has almost the same logical control flow as the synchronous version in Listing 20.1, both versions attempt to download a resource from a server, and if the download succeeds, the result is returned. (If the download fails, the exception’s type is interrogated to determine the right course of action.) However, clearly the asynchronous version of Listing 20.2 is significantly more difficult to read, understand, and change than the corresponding synchronous version in Listing 20.1. Unlike the synchronous version, which uses standard control flow statements, the asynchronous version is forced to create multiple lambda expressions to express the continuation logic in the form of delegates.

And this is a fairly simple example! Imagine what the asynchronous code would look like if, for example, the synchronous code contained a loop that retried the operation three times if it failed, if it tried to contact multiple different servers, if it took a collection of resources rather than a single one, or if all of these possible features occurred together. Adding those features to the synchronous version would be straightforward, but it is not at all clear how to do so in the asynchronous version. Rewriting synchronous methods into asynchronous methods by explicitly specifying the continuation of each task gets very complicated very quickly even if the synchronous continuations are what appear to be very simple control flows.

Introducing async Return of ValueTask<T>

We use asynchronous methods for long-running, high-latency operations. And (obviously), since Task/Task<T> is the return, we always need to obtain one of these objects to return. The alternative, to return null, would force callers to always check for null before accessing the Task—an unreasonable and frustrating API from a usability perspective. Generally, the cost to create a Task/Task<T> is insignificant in comparison to the long-running, high-latency operation.

What happens, though, if the operation can be short-circuited and a result returned immediately? Consider, for example, compressing a buffer. If the amount of data is significant, performing the operation asynchronously makes sense. If, however, data is 0-length, then the operation can return immediately, and obtaining a (cached or new instance of) Task/Task<T> is pointless because there is no need for a task when the operation completes immediately. Unfortunately, there was no alternative when async/await was introduced in C# 5.0. However, C# 7.0 added support for arbitrary types that meet certain criteria—namely, support for a GetAwaiter() method, as detailed in the Advanced Topic titled “Awaiting Non-Task<T> or Values.” For example, C# 7.0-related .NET frameworks include ValueTask<T>, a value type that scales down to support lightweight instantiation when a long-running operation can be short-circuited or that can be converted to a full Task otherwise. Listing 20.4 provides an example of file compression but escaping via ValueTask<T> if the compression can be short-circuited.

Listing 20.4: Returning ValueTask<T> from an async method

using System;

using System.IO;

using System.Net;

using System.Linq;

using System.Threading.Tasks;

public class Program

{

private static async ValueTask<byte[]> CompressAsync(byte[] buffer)

{

if (buffer.Length == 0)

{

return buffer;

}

using (MemoryStream memoryStream = new MemoryStream())

using (System.IO.Compression.GZipStream gZipStream =

new System.IO.Compression.GZipStream(

memoryStream, System.IO.Compression.CompressionMode.Compress))

{

await gZipStream.WriteAsync(buffer, 0, buffer.Length);

buffer = memoryStream.ToArray();

}

return buffer;

}

// ...

}

Notice that even though an asynchronous method, such as GZipStream.WriteAsync(), might return Task<T>, the await implementation still works within a ValueTask<T> returning method. In Listing 20.4, for example, changing the return from ValueTask<T> to Task<T> involves no other code changes.

ValueTask<T> raises the question of when to use it versus Task/Task<T>. If your operation doesn’t return a value, just use Task (there is no non-generic ValueTask<T> because it has no benefit). If your operation is likely to complete asynchronously, or if it’s not possible to cache tasks for common result values, Task<T> is also preferred. If, however, the operation is likely to complete synchronously and you can’t reasonably cache all common return values, ValueTask<T> might be appropriate. For example, there’s generally no benefit to returning ValueTask<bool> instead of Task<bool>, because you can easily cache a Task<bool> for both true and false values—and in fact, the async infrastructure does this automatically. In other words, when returning an asynchronous Task<bool> method that completes synchronously, a cached result Task<bool> will return regardless.

Asynchronous Streams

C# 8.0 introduced the ability to program **async streams**, essentially the ability to leverage the async pattern with iterators. As discussed in Chapter 15, collections in C# are all built on the IEnumerable<T> and IEnumerator<T>, the former with a single GetEnumerator<T>() function that returns an IEnumerator<T> over which you can iterate. And, when building an iterator with yield return, the method needs to return IEnumerable<T> or IEnumerator<T>. In contrast, an async method (prior to C# 8.0) must return Task, Task<T> or ValueTask<T>[[2]](#footnote-3). The conflict, therefore, is that you can't have both an async method and an iterator. When invoking an async method while iterating over a collection, for example, you can't yield the results to a calling function prior to the completion of all expected iterations.

To address these problems, the C# team added asynchronous streams (async stream) support in C# 8.0, which is specifically designed to enable asynchronous iteration and the building of asynchronous collections and enumerable type methods using yield return.

For example, imagine encrypting content with an async method, EncryptFilesAsync(), given a directory (defaulting to the current directory). **Listing 20.5** provides the code listing.

Listing 20.5: Async Streams

using System.IO;

using System.Linq;

using System.Threading.Tasks;

using System.Collections.Generic;

using System.Threading;

using System.Runtime.CompilerServices;

using AddisonWesley.Michaelis.EssentialCSharp.Shared;

public class Program

{

public static async void Main(params string[] args)

{

string directoryPath = Directory.GetCurrentDirectory();

const string searchPattern = "\*";

// ...

using Cryptographer cryptographer = new Cryptographer();

IEnumerable<string> files = Directory.EnumerateFiles(

directoryPath ?? Directory.GetCurrentDirectory(), searchPattern);

// Create a cancellation token source to cancel

// if the operation takes more than a minute.

using CancellationTokenSource cancellationTokenSource =

new CancellationTokenSource(1);

await foreach ((string fileName, string encryptedFileName)

in EncryptFilesAsync(files, cryptographer)

.Zip(files.ToAsyncEnumerable())

.WithCancellation(cancellationTokenSource.Token))

{

Console.WriteLine($"{fileName}=>{encryptedFileName}");

}

}

public static async IAsyncEnumerable<string> EncryptFilesAsync(

IEnumerable<string> files, Cryptographer cryptographer,

[EnumeratorCancellation] CancellationToken cancellationToken = default)

{

foreach (string fileName in files)

{

yield return await EncryptFileAsync(fileName, cryptographer);

cancellationToken.ThrowIfCancellationRequested();

}

}

private static async Task<string> EncryptFileAsync(

string fileName, Cryptographer cryptographer)

{

string encryptedFileName = $"{fileName}.encrypt";

await using FileStream outputFileStream =

new FileStream(encryptedFileName, FileMode.Create);

string data = await File.ReadAllTextAsync(fileName);

await cryptographer.EncryptAsync(data, outputFileStream);

return encryptedFileName;

}

}

The listing begins with Main() inside of which there is a C# 8.0 introduced async foreach statement iterating over an asynchronous method, EncryptFilesAsync() (we will address the WithCancellation() invocation shortly.)

The EncryptFilesAsync() method iterates over each of them specified files the with a foreach loop. Inside the foreach loop there are two async method invocations. The first is a call to File.ReadAllTextAsync(), which reads in all the content of the file. Once the content is available in memory, the code invokes the EncryptAsync() method to encrypt it before returning the encrypted file name via a yield return statement. Thus, the method provides an example of the need to provide an asynchronous iterator to the caller. The key to making this possible is the EncryptFilesAsync()'s decoration with async and its return of IAsyncEnumerable<T> (where T is a string in this case).

Given a method returning IAsyncEnumerable<T>, you can consume it using an await foreach statement as demonstrated in the Main method of **Listing 20.5**. - demonstrating both producing and consuming an async stream.

The signature for GetAsyncEnumerator() includes a CancellationToken parameter. Because await foreach generates the code that calls GetAsyncEnumerator(), the way to inject a cancellation token and provide cancellation is via the WithCancellation() extension method (as **Figure 19.4** shows, there’s no WithCancellation() method on IAsyncEnumerable<T> directly). To support cancellation in an async stream method, add an optional CancellationToken with an EnumeratorCancellationAttribute as demonstrated by the EncryptFilesAsunc method declaration:

static public async IAsyncEnumerable<string>

EncryptFilesAsync(

string directoryPath = null,

string searchPattern = "\*",

[EnumeratorCancellation] CancellationToken

cancellationToken = default)

{ ... }

In **Listing 20.5**, you provide an async stream method that returns the IAsyncEnumerable<T> interface. Like the non-async iterators, however, you can also implement the IAsyncEnumerable<T> interface with its GetAsyncEnumerator() method. Of course, any class implementing the interface can then be iterated over with an await foreach statement, as shown in **Listing 20.6**.

Listing 20.6: Async Streams

class AsyncEncryptionCollection : IAsyncEnumerable<string?>

{

public async IAsyncEnumerator<string> GetAsyncEnumerator(

CancellationToken cancellationToken = default)

{

// ...

}

static public async void Main()

{

AsyncEncryptionCollection collection =

new AsyncEncryptionCollection();

// ...

await foreach (string fileName in collection)

{

Console.WriteLine(fileName);

}

}

}

One point of caution: Remember that declaring an async method doesn’t automatically cause the execution to run in parallel. Just because the EncryptFilesAsync() method is asynchronous doesn’t mean that iterating over each file and invoking File.ReadAllTextAsync() and Cryptographer.EncryptAsync() will happen in parallel. For that, you need to leverage a Task invocation or something like a System.Threading.Tasks.Parallel.ForEach().

The IAsyncEnumerable<T> interfaces along with its partner, IAsyncEnumerator<T>, are C# 8.0 additions that match their synchronous equivalents as shown in **Figure 20.2**.

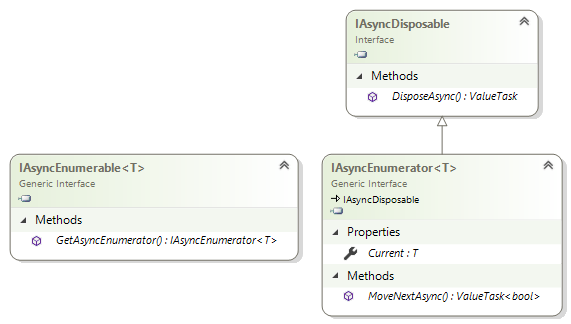


Figure 20.2: IAsycnEnumerable<T> and related interfaces

Note that both the IAsyncDisposable.DisposeAsync() and IAsyncEnumerator<T>.MoveNextAsync() methods are asynchronous versions of IEnumerators<T> equivalent methods. The Current property isn’t asynchronous. (Also, there’s no Reset() method in the asynchronous implementations.)

IAsyncDisposable and the await using declaration and statement

IAsyncDisposable, is the asynchronous equivalent of IDisposable so it can be invoked using C# 8.0’s new **await using statement** or **await using declaration**. In **Listing 20.5** we use the latter when declaring outputFileStream because, like IAsyncEnumerable<T>, FileStream also implements IAsyncDisposable. Like with the using declarative, you can't reassign a variable declared with an async using.

Not surprisingly, the await using statement follows the same syntax as a using statement:

await using FileStream outputFileStream =

new FileStream(encryptedFileName, FileMode.Create);

{ ... }

Both can be used any time the type implements IAsyncDisposable or simply has a DisposeAsync() method. The result is that the C# compiler injects a try/finally around the declaration and before the variable goes out of scope, and then invokes await DisposeAsync()within the finally block[[3]](#footnote-4). Thus, ensuring that all resources are cleaned up.

Note that IAsyncDisposable and IDisposable are not related to each other via inheritance, so their implementations are not dependent either; one can be implemented without the other.

Using LINQ with IAsyncEnumerable

In the await foreach statement of Listing 20.5 we invoke the LINQ AsyncEnumberable.Zip() method in order to pair the original file name with the encrypted file name.

await foreach (

(string fileName, string encryptedFileName) in

EncryptFilesAsync(files)

.Zip(files.ToAsyncEnumerable()))

{

Console.WriteLine($"{fileName}=>{encryptedFileName}");

}

AsyncEnumerable provides the LINQ functionality for IAsyncEnumerable<T> as you might expect. However, the library is not available in the BCL. Rather[[4]](#footnote-5) you need to add a reference to the System.Linq.Async NuGet package in order to access the async LINQ capabilities.

AsyncEnumerable is defined in the System.Linq (not a different unique namespace with async functionality). Not surprisingly it includes asynchronous versions of the standard LINQ operators such as Where(), Select() and the Zip() method used in the aforementioned listing. They are "asynchronous versions" because they are extension methods on IAsyncEnumerable rather than IEnumerable<T>. In addition, AsyncEnumerable includes a series of \*Async(), \*AwaitAsync() and \*AwaitWithCancellationAsync() methods. The Select\*() versions of each are shown in **Listing 20.6B**.

Listing 20.6B: Signatures of AsyncEnumerable Select\*()[[5]](#footnote-6) methods

public static IAsyncEnumerable<TResult> Select<TSource?, TResult?>(

this IAsyncEnumerable<TSource> source,

Func<TSource, TResult> selector);

public static IAsyncEnumerable<TResult> SelectAwait<TSource?, TResult?>(

this IAsyncEnumerable<TSource> source,

Func<TSource, ValueTask<TResult>>? selector);

public static

IAsyncEnumerable<TResult> SelectAwaitWithCancellation<TSource?, TResult?>(

this IAsyncEnumerable<TSource> source,

Func<TSource, CancellationToken, ValueTask<TResult>> selector);

The method names that match the Enumerable equivalents, Select() in this case, has a similar "instance" signature - the TResult and TSource are different. Both signatures with "Await" in the name take asynchronous signatures that have a selector that returns a ValueTask<T>. For example, you could invoke **Listing 20.5**'s EncryptFileAsync() from SelectAwait() as follows:

IAsyncEnumerable<string> items = files.ToAsyncEnumerable();

items = items.SelectAwait(

(text, id) => EncryptFileAsync(text));

The important thing to note is that EncyryptFileAsync() method returns a ValueTask<T>, which is what both \*Await() and \*AwaitWithCancellationAsync() require. The latter, of course, also allowing for a cancellation token to be specified.

Another async LINQ method worthy of mention is the ToAsyncEnumerable() method used in **Listing 20.5**. Since async LINQ methods work with IAsyncEnumerable<T> interfaces, ToAsyncEnumerable() takes care of converting IEnumerable<T> to IAsyncEnumerable<T>. Similarly there is a ToEnumerable() method that makes the opposite conversion. (Admittedly , using files.ToAsyncEnumerable() in the snippet is a contrived example for retrieving an IAsyncEnumerable<string>.)

The scalar versions of the async LINQ methods similarly match the IEnumerable<T> as well - with a \*Await(), \*AwaitAsync(), and \*AwaitWithCancellation() set of members. The key difference is that they all return a ValueTask<T>. The following snippet provides an example of using the AverageAsync() method:

double average = await AsyncEnumerable.Range(

0, 999).AverageAsync();

As such, we can use await to treat the return as a double rather than a ValueTask<double>.

\*\*\*COMP: Insert “End 8.0” tab

Returning void from an Asynchronous Method

We've already looked at examples of asynchronous methods that return Task, Task<T>, ValueTask<T>, and IAsyncEnumerable<T>. The last return option available for an async method is void—a method henceforth referred to as an async void **method**. However, async void methods should generally be avoided and might more accurately be considered a non-option. Unlike when returning a Task/Task<T>, when there is a return, it is indeterminate when a method completes executing, and if an exception occurs, returning void means there is no such container to report an exception. In the exception case, any exception that is thrown on an async void method likely ends up on the UI SynchronizationContext—effectively an unhandled exception (see the Advanced Topic titled “Dealing with Unhandled Exceptions on a Thread” in the previous chapter).

If async void methods should be generally avoided, why are they allowed in the first place? It’s because async void methods can be used to enable async event handlers. As discussed in Chapter 14, an event should be declared as an EventHandler<T> where EventHandler<T> has a signature of

void EventHandler<TEventArgs>(object sender, TEventArgs e)

Therefore, to fit the convention of an event matching the EventHandler<T> signature, an async event needs to return void. One might suggest changing the convention, but (as discussed in Chapter 14) there could be multiple subscribers and retrieving the return from multiple subscribers is nonintuitive and cumbersome. For this reason, the guideline is to avoid async void methods unless they are subscribers to an event handler—in which case they should not throw exceptions. Alternatively, you should provide a synchronization context to receive notifications of synchronization events such as the scheduling of work (e.g., Task.Run()) and, perhaps more important, unhandled exceptions. Listing 20.7 and the accompanying Output 20.1 provide an example of how to do this.

Listing 20.7: Catching an Exception from an async void Method

using System;

using System.Threading;

using System.Threading.Tasks;

public class AsyncSynchronizationContext : SynchronizationContext

{

public Exception? Exception { get; set; }

public ManualResetEventSlim ResetEvent { get;} = new ManualResetEventSlim();

public override void Send(SendOrPostCallback callback, object? state)

{

try

{

Console.WriteLine($@"Send notification invoked...(Thread ID: {

Thread.CurrentThread.ManagedThreadId})");

callback(state);

}

catch (Exception exception)

{

Exception = exception;

#if !WithOutUsingResetEvent

ResetEvent.Set();

#endif

}

}

public override void Post(SendOrPostCallback callback, object? state)

{

try

{

Console.WriteLine($@"Post notification invoked...(Thread ID: {

Thread.CurrentThread.ManagedThreadId})");

callback(state);

}

catch (Exception exception)

{

Exception = exception;

#if !WithOutUsingResetEvent

ResetEvent.Set();

#endif

}

}

}

public class Program

{

static bool EventTriggered { get; set; }

public const string ExpectedExceptionMessage = "Expected Exception";

public static async Task Main()

{

AsyncSynchronizationContext synchronizationContext =

new AsyncSynchronizationContext();

SynchronizationContext.SetSynchronizationContext(synchronizationContext);

try

{

await OnEvent(typeof(Program), EventArgs.Empty);

#if WithOutUsingResetEvent

Task.Delay(1000).Wait(); //

#else

synchronizationContext.ResetEvent.Wait();

#endif

if(synchronizationContext.Exception != null)

{

Console.WriteLine($@"Throwing expected exception....(Thread ID: {

Thread.CurrentThread.ManagedThreadId})");

System.Runtime.ExceptionServices.ExceptionDispatchInfo.Capture(

synchronizationContext.Exception).Throw();

}

}

catch(Exception exception)

{

Console.WriteLine($@"{exception} thrown as expected.(Thread ID: {

Thread.CurrentThread.ManagedThreadId})");

}

}

private static async void OnEvent(object sender, EventArgs eventArgs)

{

Console.WriteLine($@"Invoking Task.Run...(Thread ID: {

Thread.CurrentThread.ManagedThreadId})");

await Task.Run(()=>

{

Console.WriteLine($@"Running task... (Thread ID: {

Thread.CurrentThread.ManagedThreadId})");

throw new Exception(ExpectedExceptionMessage);

});

}

}

Output 20.1

Invoking Task.Run...(Thread ID: 8)

Running task... (Thread ID: 9)

Post notification invoked...(Thread ID: 8)

Post notification invoked...(Thread ID: 8)

Throwing expected exception....(Thread ID: 8)

System.Exception: Expected Exception

at AddisonWesley.Michaelis.EssentialCSharp.Chapter19.Listing19\_17.Program.Main() in

...Listing19.17.AsyncVoidReturn.cs:line 80 thrown as expected.(Thread ID: 8)

The code executes procedurally up until the await Task.Run() invocation within OnEvent() starts. Following its completion, control is passed to the Post() method within AsyncSynchronizationContext. After the execution and completion of the Post() invocation, the Console.WriteLine("throw Exception...") executes, and then an exception is thrown. This exception is captured by the AsyncSynchronizationContext.Post() method and passed back into Main().

In this example, we use a Task.Delay() call to ensure the program doesn’t end before the Task.Run() invocation but, as shown in a later chapter, a ManualResetEventSlim would be the preferred approach.

\*\*\*COMP: Insert “End 7.0” tab

Asynchronous Lambdas and Local Functions

Just as a lambda expression converted to a delegate can be used as a concise syntax for declaring a normal method, so C# 5.0 (and later) also allows lambdas containing await expressions to be converted to delegates. To do so, just precede the lambda expression with the async keyword. In Listing 20.8, we rewrite the GetResourceAsync() method from Listing 20.02 from an async method to an async lambda.

Listing 20.8: An Asynchronous Client-Server Interaction as a Lambda Expression

using System;

using System.IO;

using System.Net;

using System.Linq;

using System.Threading.Tasks;

public class Program

{

public static void Main(string[] args)

{

string url = "http://www.IntelliTect.com";

if(args.Length > 0)

{

url = args[0];

}

Console.Write(url);

Func<string, Task> writeWebRequestSizeAsync =

async (string webRequestUrl) =>

{

// Error handling ommitted for

// elucidation

WebRequest webRequest =

WebRequest.Create(url);

WebResponse response =

await webRequest.GetResponseAsync();

using(StreamReader reader =

new StreamReader(

response.GetResponseStream()))

{

string text =

(await reader.ReadToEndAsync());

Console.WriteLine(

FormatBytes(text.Length));

}

};

Task task = writeWebRequestSizeAsync(url);

while (!task.Wait(100))

{

Console.Write(".");

}

}

// ...

}

\*\*\*COMP: Insert “Begin 7.0”tab

Similarly, the same can be achieved in C# 7.0 or later with a local function. For example, in Listing 20.8, you could change the lambda expression header (everything up to and including the => operator) to:

async Task WriteWebRequestSizeAsync(string webRequestUrl)

leaving everything in the body, including the curly braces, unchanged.

Note that an async lambda expression has exactly the same restrictions as the named async method:

* An async lambda expression must be converted to a delegate whose return type is void, Task, Task<T>, or, as of C# 7.0, ValueTask<T>.
* The lambda is rewritten so that return statements become signals that the task returned by the lambda has completed with the given result.
* Execution within the lambda expression occurs synchronously until the first await on an incomplete awaitable is executed.
* All instructions following the await will execute as continuations on the return from the invoked asynchronous method (or, if the awaitable is already complete, will be simply executed synchronously rather than as continuations).
* An async lambda expression can be invoked with an await (not shown in Listing 20.8).

\*\*\*COMP: Insert “End 7.0” tab

Advanced Topic

Implementing a Custom Asynchronous Method

Implementing an asynchronous method by relying on other asynchronous methods (which, in turn, rely on more asynchronous methods) is relatively easy with the await keyword. However, at some point in the call hierarchy, it becomes necessary to write a “leaf” asynchronous Task-returning method. Consider, for example, an asynchronous method for running a command-line program with the eventual goal that the output could be accessed. Such a method would be declared as follows:

public static Task<Process> RunProcessAsync(string filename)

The simplest implementation would, of course, be to rely on Task.Run() again and call both the System.Diagnostics.Process’s Start() and WaitForExit() methods. However, creating an additional thread in the current process is unnecessary when the invoked process itself will have its own collection of one or more threads. To implement the RunProcessAsync() method and return to the caller’s synchronization context when the invoked process completes, we can rely on a TaskCompletionSource<T> object, as shown in Listing 20.9.

Listing 20.9: Implementing a Custom Asynchronous Method

using System.Diagnostics;

using System.Threading;

using System.Threading.Tasks;

class Program

{

public static Task<Process> RunProcessAsync(

string fileName,

string arguments = "",

CancellationToken cancellationToken =

default(CancellationToken))

{

TaskCompletionSource<Process> taskCS =

new TaskCompletionSource<Process>();

Process process = new Process()

{

StartInfo = new ProcessStartInfo(fileName)

{

UseShellExecute = false,

Arguments = arguments

},

EnableRaisingEvents = true

};

process.Exited += (sender, localEventArgs) =>

{

taskCS.SetResult(process);

};

cancellationToken

.ThrowIfCancellationRequested();

process.Start();

cancellationToken.Register(() =>

{

process.CloseMainWindow();

});

return taskCS.Task;

}

// ...

}

Ignore the highlighting for the moment and instead focus on the pattern of using an event for notification when the process completes. Since System.Diagnostics.Process includes a notification upon exit, we register for this notification and use it as a callback from which we can invoke TaskCompletionSource.SetResult(). The code in Listing 20.9 follows a fairly common pattern that you can use to create an asynchronous method without having to resort to Task.Run().

Another important characteristic that an async method might require is cancellation. TAP relies on the same methods for cancellation as the TPL does—namely, a System.Threading.CancellationToken. Listing 20.9 highlights the code necessary to support cancellation. In this example, we allow for canceling before the process ever starts, as well as an attempt to close the application’s main window (if there is one). A more aggressive approach would be to call Process.Kill(), but this method could potentially cause problems for the program that is executing.

Notice that we don’t register for the cancellation event until after the process is started. This avoids any race conditions that might occur if cancellation is triggered before the process actually begins.

One last feature to consider supporting is a progress update. Listing 20.10 is the full version of RunProcessAsync() with just such an update.

Listing 20.10: Implementing a Custom Asynchronous Method with Progress Support

using System;

using System.Diagnostics;

using System.Threading;

using System.Threading.Tasks;

class Program

{

static public Task<Process> RunProcessAsync(

string fileName,

string arguments = "",

CancellationToken cancellationToken =

default(CancellationToken),

IProgress<ProcessProgressEventArgs>? progress =

null,

object? objectState = null)

{

TaskCompletionSource<Process> taskCS =

new TaskCompletionSource<Process>();

Process process = new Process()

{

StartInfo = new ProcessStartInfo(fileName)

{

UseShellExecute = false,

Arguments = arguments,

RedirectStandardOutput =

progress != null

},

EnableRaisingEvents = true

};

process.Exited += (sender, localEventArgs) =>

{

taskCS.SetResult(process);

};

if(progress != null)

{

process.OutputDataReceived +=

(sender, localEventArgs) =>

{

progress.Report(

new ProcessProgressEventArgs(

localEventArgs.Data,

objectState));

};

}

if(cancellationToken.IsCancellationRequested)

{

cancellationToken

.ThrowIfCancellationRequested();

}

process.Start();

if(progress != null)

{

process.BeginOutputReadLine();

}

cancellationToken.Register(() =>

{

process.CloseMainWindow();

cancellationToken

.ThrowIfCancellationRequested();

});

return taskCS.Task;

}

// ...

}

class ProcessProgressEventArgs

{

// ...

}

\*\*\*COMP: End Advanced Topic after listing

Advanced Topic

Awaiting Non-Task<T> or Values

Generally, the expression that follows the await keyword is of either type Task or type Task<T>. In the examples of await shown so far in this chapter, the expressions that follow the keyword have all returned Task<T>. From a syntax perspective, an await operating on type Task is essentially the equivalent of an expression that returns void. In fact, because the compiler does not even know whether the task has a result, much less which type it is, such an expression is classified in the same way as a call to a void-returning method; that is, you can use it only in a statement context. Listing 20.11 shows some await expressions used as statement expressions.

Listing 20.11: An await Expression May Be a Statement Expression

async Task<int> DoStuffAsync()

{

await DoSomethingAsync();

await DoSomethingElseAsync();

return await GetAnIntegerAsync() + 1;

}

Here we presume that the first methods return a Task rather than a Task<T>. Since there is no result value associated with the first two tasks, awaiting them produces no value; thus the expression must appear as a statement. The third task is presumably of type Task<int>, and its value can be used in the computation of the value of the task returned by DoStuffAsync().

This Advanced Topic begins with the word Generally—a deliberate injection of incertitude. In fact, the exact rule regarding the return type that await requires is more generic than just Task or Task<T>. Rather, it requires that the type support a GetAwaiter. This method produces an object that has certain properties and methods needed by the compiler’s rewriting logic. This makes the system extensible by third parties.[[6]](#footnote-7) If you want to design your own non-Task-based asynchrony system that uses some other type to represent asynchronous work, however, you can do so and still use the await syntax.

Note, however, until C# 7.0’s introduction of ValueTask<T>, it was not possible to make async methods return something other than void, Task, or Task<T>, no matter which type is awaited inside the method.

Wrapping your head around precisely what is happening in an async method can be difficult, but it is far less difficult than trying to figure out what asynchronous code written with explicit continuations in lambdas is doing. The key points to remember are as follows:

* When control reaches an await keyword, the expression that follows it produces a task.[[7]](#footnote-8) Control then returns to the caller so that it can continue to do work while the task completes asynchronously.
* Sometime after the task completes, control resumes at the point following the await. If the awaited task produces a result, that result is then obtained. If it faulted, the exception is thrown.
* A return statement in an async method causes the task associated with the method invocation to become completed; if the return statement has a value, the value returned becomes the result of the task.

Task Schedulers and the Synchronization Context

On occasion, this chapter has mentioned the task scheduler and its role in determining how to assign work to threads efficiently. Programmatically, the task scheduler is an instance of the System.Threading.Tasks.TaskScheduler. This class, by default, uses the thread pool to schedule tasks appropriately, determining how to safely and efficiently execute them—when to reuse them, dispose them, or create additional ones.

It is possible to create your own task scheduler that makes different choices about how to schedule tasks by deriving a new type from the TaskScheduler class. You can obtain a TaskScheduler that will schedule a task to the current thread (or, more precisely, to the **synchronization context** associated with the current thread), rather than to a different worker thread, by using the static FromCurrentSynchronizationContext() method.[[8]](#footnote-9)

The synchronization context under which a task executes and, in turn, the continuation task(s) execute(s), is important because the awaiting task consults the synchronization context (assuming there is one) so that a task can execute efficiently and safely. Listing 2012 (along with Output 20.2) is similar to Listing 19.5 except that it also prints out the thread ID when it displays the message.

Listing 20.12: Calling Task.ContinueWith()

using System;

using System.Threading;

using System.Threading.Tasks;

public class Program

{

public static void Main()

{

DisplayStatus("Before");

Task taskA =

Task.Run(() =>

DisplayStatus("Starting..."))

.ContinueWith( antecedent =>

DisplayStatus("Continuing A..."));

Task taskB = taskA.ContinueWith( antecedent =>

DisplayStatus("Continuing B..."));

Task taskC = taskA.ContinueWith( antecedent =>

DisplayStatus("Continuing C..."));

Task.WaitAll(taskB, taskC);

DisplayStatus("Finished!");

}

private static void DisplayStatus(string message)

{

string text = string.Format(

$@"{ Thread.CurrentThread.ManagedThreadId

}: { message }");

Console.WriteLine(text);

}

}

Output 20.2

1: Before

3: Starting...

4: Continuing A...

3: Continuing C...

4: Continuing B...

1: Finished!

What is noteworthy about this output is that the thread ID changes sometimes and gets repeated at other times. In this kind of plain console application, the synchronization context (accessible from SynchronizationContext.Current) is null—the default synchronization context causes the thread pool to handle thread allocation instead. This explains why the thread ID changes between tasks: Sometimes the thread pool determines that it is more efficient to use a new thread, and sometimes it decides that the best course of action is to reuse an existing thread.

Fortunately, the synchronization context gets set automatically for types of applications where that is critical. For example, if the code creating tasks is running in a thread created by ASP.NET, the thread will have a synchronization context of type AspNetSynchronizationContext associated with it. In contrast, if your code is running in a thread created in a Windows UI application (Windows Presentation Foundation [WPF] or Windows Forms), the thread will have an instance of DispatcherSynchronizationContext associated with it. (For console applications, there is no synchronization context by default.) Since the TPL consults the synchronization context and the synchronization context varies depending on the circumstances of the execution, the TPL is able to schedule continuations executing in contexts that are both efficient and safe.

To modify the code so that the synchronization context is leveraged instead, you must (1) set the synchronization context and (2) use async/await so that the synchronization context is consulted.[[9]](#footnote-10)

It is possible to define custom synchronization contexts and to work with existing synchronization contexts to improve their performance in some specific scenarios. However, describing how to do so is beyond the scope of this text.

async/await with the Windows UI

One place where synchronization is especially important is in the context of UI. With the Windows UI, for example, a message pump processes messages such as mouse click and move events. Furthermore, the UI is single-threaded, so that interaction with any UI components (e.g., a text box) must always occur from the single UI thread. One of the key advantages of the async/await pattern is that it leverages the synchronization context to ensure that continuation work—work that appears after the await statement—will always execute on the same synchronization task that invoked the await statement. This approach is of significant value because it eliminates the need to explicitly switch back to the UI thread to update a control.

To better appreciate this benefit, consider the example of a UI event for a button click in WPF, as shown in Listing 20.13.

Listing 20.13: Synchronous High-Latency Invocation in WPF

using System;

private void PingButton\_Click(

object sender, RoutedEventArgs e)

{

StatusLabel.Content = "Pinging...";

UpdateLayout();

Ping ping = new Ping();

PingReply pingReply =

ping.Send("www.IntelliTect.com");

StatusLabel.Text = pingReply.Status.ToString();

}

Given that StatusLabel is a WPF System.Windows.Controls.TextBlock control and we have updated the Content property twice within the PingButton\_Click() event subscriber, it would be a reasonable assumption that first “Pinging…” would be displayed until Ping.Send() returned, and then the label would be updated with the status of the Send() reply. As those experienced with Windows UI frameworks well know, this is not, in fact, what happens. Rather, a message is posted to the Windows message pump to update the content with “Pinging…,” but because the UI thread is busy executing the PingButton\_Click() method, the Windows message pump is not processed. By the time the UI thread frees up to look at the Windows message pump, a second Text property update request has been queued and the only message that the user is able to observe is the final status.

To fix this problem using TAP, we change the code highlighted in Listing 20.14.

Listing 20.14: Synchronous High-Latency Invocation in WPF Using await

using System;

async private void PingButton\_Click(

object sender, RoutedEventArgs e)

{

StatusLabel.Content = "Pinging...";

UpdateLayout();

Ping ping = new Ping();

PingReply pingReply =

await ping.SendPingAsync("www.IntelliTect.com");

StatusLabel.Text = pingReply.Status.ToString();

}

This change offers two advantages. First, the asynchronous nature of the ping call frees up the caller thread to return to the Windows message pump caller’s synchronization context, and it processes the update to StatusLabel.Content so that “Pinging…” appears to the user. Second, when awaiting ping.SendTaskAsync() completes, it will always execute on the same synchronization context as the caller. Also, because the synchronization context is specifically appropriate for Windows UI, it is single-threaded, and therefore, the return will always be to the same thread—the UI thread. In other words, rather than immediately executing the continuation task, the TPL consults the synchronization context, which instead posts a message regarding the continuation work to the message pump. Next, because the UI thread monitors the message pump, upon picking up the continuation work message, it invokes the code following the await call. (As a result, the invocation of the continuation code is on the same thread as the caller that processed the message pump.)

There is a key code readability feature built into the TAP language pattern. Notice in Listing 20.14 that the call to return pingReply.Status appears to flow naturally after the await, providing a clear indication that it will execute immediately following the previous line. However, writing what really happens from scratch would be far less understandable for multiple reasons.

await Operators

There is no limitation on the number of times that await can be placed into a single method. In fact, such statements are not limited to appearing one after another. Rather, await statements can be placed into loops and processed consecutively one after the other, thereby following a natural control flow the way code appears. Consider the example in Listing 20.15.

Listing 20.15: Iterating over an Await Operation

async private void PingButton\_Click(

object sender, RoutedEventArgs e)

{

List<string> urls = new List<string>()

{

"www.habitat-spokane.org",

"www.partnersintl.org",

"www.iassist.org",

"www.fh.org",

"www.worldvision.org"

};

IPStatus status;

Func<string, Task<IPStatus>> func =

async (localUrl) =>

{

Ping ping = new Ping();

PingReply pingReply =

await ping.SendPingAsync(localUrl);

return pingReply.Status;

};

StatusLabel.Content = "Pinging…";

foreach(string url in urls)

{

status = await func(url);

StatusLabel.Text =

$@"{ url }: { status.ToString() } ({

Thread.CurrentThread.ManagedThreadId })";

}

}

Regardless of whether the await statements occur within an iteration or as separate entries, they will execute serially, one after the other and in the same order they were invoked from the calling thread. The underlying implementation is to string them together in the semantic equivalent of Task.ContinueWith() except that all of the code between the await operators will execute in the caller’s synchronization context.

Support for TAP from the UI is one of the key scenarios that led to TAP’s creation. A second scenario takes place on the server, when a request comes in from a client to query an entire table’s worth of data from the database. As querying the data could be time-consuming, a new thread should be created rather than consuming one from the limited number allocated to the thread pool. The problem with this approach is that the work to query from the database is executing entirely on another machine. There is no reason to block an entire thread given that the thread is generally not active anyway.

To summarize, TAP was created to address these key problems:

* There is a need to allow long-running activities to occur without blocking the UI thread.
* Creating a new thread (or Task) for non–CPU-intensive work is relatively expensive when you consider that all the thread is doing is waiting for the activity to complete.
* When the activity completes (either by using a new thread or via a callback), it is frequently necessary to make a thread synchronization context switch back to the original caller that initiated the activity.
* TAP provides a new pattern that works for both CPU-intensive and non–CPU-intensive asynchronous invocations—one that all .NET languages support explicitly.

\*\*\*COMP: Insert “Begin 7.0” tab

Parenthetically, in C# 5.0 and 6.0 there was a restriction that awaits couldn’t appear within exception handling catch or finally statements. However, this restriction has been removed starting with C# 7.0. This is a helpful improvement when you consider that you likely might want to log the exception from the outermost exception handler in the call stack and logging is a relatively expensive operation such that doing so with an asynchronous await is desirable.

\*\*\*COMP: Insert “End 7.0” tab

\*\*\*COMP: Insert “End 5.0” tab

Summary

In this chapter, we started by examining the basic parts of multithreaded programs: the Thread class, which represents an independent point of control in a program, and the ThreadPool, which encourages efficient allocation and scheduling of threads to multiple CPUs. However, these APIs are low-level entities that are difficult to work with directly. Starting with Version 4.0, the Microsoft .NET Framework provides the Parallel Extensions library, which includes the Task Parallel Library (TPL) and Parallel LINQ (PLINQ). Both provide new APIs for creating and scheduling units of work represented by Task objects, executing loops in parallel using Parallel.For() and Parallel.ForEach(), and automatically parallelizing LINQ queries with AsParallel().

We also discussed how C# 5.0 (and later) makes programming complex workflows with Task objects much easier by automatically rewriting your programs to manage the continuation “wiring” that composes larger tasks out of smaller tasks.

At the beginning of this chapter, we briefly glossed over some of the difficult problems that developers often face when writing multithreaded programs: atomicity problems, deadlocks, and other race conditions that introduce uncertainty and bad behavior into multithreaded programs. The standard way to avoid these problems is to carefully write code that uses locks to synchronize access to shared resources; this is the topic of the next chapter.

\*\*\*COMP: Insert “End 4.0” margin note

1. . Technically, you can also return any type that implements a GetAwaiter() method. See the Advanced Topic titled “Awaiting Non-Task<T> or Values” later in the chapter. [↑](#footnote-ref-2)
2. Or void [↑](#footnote-ref-3)
3. Using await inside a finally block was added in C# 6.0. [↑](#footnote-ref-4)
4. At least at the time of this writing - .NET Core 3.0 & 3.1 [↑](#footnote-ref-5)
5. Excluding SelectMany() category of methods [↑](#footnote-ref-6)
6. . This technique of allowing third-party extension by looking for a particular method by its signature is used in two other C# features: LINQ looks for methods like Select() and Where() by name to implement the select and where contextual keywords, and the foreach loop does not require that the collection implement IEnumerable, just that it have an appropriate GetEnumerator() method. [↑](#footnote-ref-7)
7. . Technically, it is an awaitable type, as described in the Advanced Topic titled “Awaiting Non-Task<T> Values.” [↑](#footnote-ref-8)
8. . For an example, see Listing C.8 in Multithreading Patterns Prior to C# 5.0, available at https://IntelliTect.com/EssentialCSharp. [↑](#footnote-ref-9)
9. . For a simple example of how to set the synchronization context of a thread and how to use a task scheduler to schedule a task to that thread, see Listing C.8 in Multithreading Patterns Prior to C# 5.0, available at https://IntelliTect.com/EssentialCSharp. [↑](#footnote-ref-10)