**Task Parallel Library:**

**Reference URL:** <https://app.pluralsight.com/guides/async-programming-task-parallel-library>

**Why the Task Parallel Library Should Matter to You**

Asynchronous programming is a broad topic with many facets but its importance is hard to overstate. Even the simplest of applications often has functionality that, if not implemented asynchronously, is unusable or, at best, inefficient. For C# developers, a working knowledge of the async and await keywords is, therefore, essential. But the functionality provided by these keywords would not be possible without .NET's Task Parallel Library (TPL). For that reason, an understanding of the TPL is fundamental for anyone interested in professional asynchronous programming with C#.

**What Exactly Is the Task Parallel Library?**

The TPL is a set of software APIs in the System.Threading.Tasks namespace of .NET. It was originally introduced with version 4.0 of the .NET Framework. Previous versions of .NET had a number of other APIs enabling asynchronous operations but they were inconsistent, cumbersome to use, and did not have built-in support for commonly needed features such as cancellation and progress reporting. Furthermore, the TPL enables a level of control and coordination of asynchronous operations that is difficult to achieve if developers try to implement such features themselves.

**The Task: An Abstraction for All Things Asynchronous**

First, a quick note on terminology: while asynchronous programming and multithreaded programming are often mentioned in the same context, they are not the same thing. Asynchronous programming is a bit more general in that it has to do with latency (something on which your application has to wait, for one reason or another), whereas multithreaded programming is a way to achieve parallelization (one or more things that your application has to do at the same time). That said, the two topics are closely related; an application that performs work on multiple threads in parallel will often need to wait until such work is completed in order to take some action (e.g. update the user interface). So, this idea of waiting is the more general characteristic that is referenced by the term asynchronous, regardless of thread count.

What does all of this have to do with the TPL? Well, the TPL was introduced to address parallelization, hence the name Task Parallel Library, so many of its APIs deal with concepts that are specific to multithreaded programming. But, as we have learned, the requirements for multithreaded programming are very similar to that of asynchronous programming in general. The TPL took advantage of this fact and introduced a beautiful abstraction called a Task, that can be used for anything that the application needs to wait for. Need to perform some complex CPU-intensive operation on a separate thread? That is a task. Need to download something from a remote network? That is also a task. Local I/O operations such as saving files to disk can also be represented as tasks. You can even aggregate multiple disparate tasks (some involving threads and others not) and wait for them all as if they were a single task.

**The Task Parallel Library in Practice**

Let's consider an example to see the TPL's Task in action. Suppose you are writing a .NET Core console application that will process a remote image. Let's say you need to download an image from the Internet, apply a blur to that image, and save it to disk. Now, normally it's fine for console applications to be synchronous, but let's say that you want to have a real time dashboard that is constantly updating with milliseconds, e.g.

while (!done)

{

Console.CursorLeft = 0;

Console.Write(System.DateTime.Now.ToString("HH:mm:ss.fff"));

Thread.Sleep(50);

}

For such a dashboard to stay reliably up to date, you'll need I/O and image manipulation operations to happen asynchronously. Using the TPL, you can accomplish that by performing such operations in methods that return a Task:

static Task<byte[]> DownloadImage(string url) { ... }

static Task<byte[]> BlurImage(string imagePath) { ... }

static Task SaveImage(byte[] bytes, string imagePath) { ... }

Notice how Task can have a generic parameter T when you want to return something for a particular Task. In this example, for both such methods you want to return the byte array of the image downloaded or blurred. In the case of our SaveImage method, the image data is written to disk and there is nothing returned.

Now for the main part of our code, where we call said functions. Assume that we're working only with JPEG images.

bool done = false;

var url = "https://...jpg";

var fileName = Path.GetFileName(url);

DownloadImage(url).ContinueWith(task1 =>

{

var originalImageBytes = task1.Result;

var originalImagePath = Path.Combine(ImageResourcesPath, fileName);

SaveImage(originalImageBytes, originalImagePath).ContinueWith(task2 =>

{

BlurImage(originalImagePath).ContinueWith(task3 =>

{

var blurredImageBytes = task3.Result;

var blurredFileName = $"{Path.GetFileNameWithoutExtension(fileName)}\_blurred.jpg";

var blurredImagePath = Path.Combine(ImageResourcesPath, blurredFileName);

SaveImage(blurredImageBytes, blurredImagePath).ContinueWith(task4 =>

{

done = true;

});

});

});

});

while (!done) { /\* update the dashboard \*/ }

Console.WriteLine("Done!");

Notice that for each Task we are adding what's called a continuation using a function called ContinueWith. The continuation is a new task and is started automatically by the TPL when the antecedent (i.e. previous) task completes. So, we've defined a chain of actions up front, and the TPL monitors and coordinates when to invoke each action. Execution of the application continues through the task definitions quickly, proceeding to the dashboard's while loop at the bottom. Since we're performing all expensive and latent operations asynchronously with a task, each of those tasks can take as long as it needs without affecting the real time updates of our dashboard.

Does that mean that each Task runs on a separate thread? To truly know the answer to that question, we would need to look at the implementation of the DownloadImage, SaveImage and BlurImage methods. That said, the beauty of the Task abstraction means that, for the purpose of the calling code we've written here, we don't need to know.

**Adding a Continuation to a Set of Tasks**

We can take our example one step further by doing the same thing, but for multiple images. In that case, we would want to wait until all of the images are processed before exiting the application. One way to accomplish this would be to save a reference to each of the last tasks in the chain, namely the tasks that correspond to saving each blurred image. If we maintain a list of those tasks, when we get to the last image we can use Task.WhenAll to aggregate all of them into a single task, to which we can again add a continuation via ContinueWith:

var saveBlurImageTasks = new List<Task>();

foreach (var url in urls)

{

var fileName = Path.GetFileName(url);

DownloadImage(url).ContinueWith(task1 =>

{

var originalImageBytes = task1.Result;

var originalImagePath = Path.Combine(ImageResourcesPath, fileName);

SaveImage(originalImageBytes, originalImagePath).ContinueWith(task2 =>

{

BlurImage(originalImagePath).ContinueWith(task3 =>

{

var blurredImageBytes = task3.Result;

var blurredFileName = $"{Path.GetFileNameWithoutExtension(fileName)}\_blurred.jpg";

var blurredImagePath = Path.Combine(ImageResourcesPath, blurredFileName);

var saveBlurImageTask = SaveImage(blurredImageBytes, blurredImagePath);

saveBlurImageTasks.Add(saveBlurImageTask);

if (saveBlurImageTasks.Count == urls.Count)

{

Task.WhenAll(saveBlurImageTasks).ContinueWith(finalTask =>

{

done = true;

});

}

});

});

});

}

**Advanced Capabilities of the Task Parallel Library**

As you can see, the TPL consists primarily of the Task class and associated functions. So far we've only scratched the surface of what is possible with the TPL. There are a number of additional static methods in the Task class, some of which provide additional operations for sets of tasks. But, even for a single task, you can customize quite a few different aspects of its behavior. For example, if you would like to perform a continuation conditionally depending on if a task failed, was canceled, or completed successfully, you can do that by providing your selection of TaskContinuationOptions to the ContinueWith method. There are a number of optimizations that can be configured with that enum as well.

You can also control if, when, and how tasks correspond to threads. For example, you can create your own implementation of the TaskScheduler class and customize how tasks are queued onto threads. You can also specify if you want a continuation to run on the main application thread, even if the antecedent task ran on a thread from the thread pool.

Finally, as hinted earlier, the TPL enables consistent cancellation via what's called a CancellationToken throughout its APIs and progress reporting is possible using an interface called IProgress<T> that was introduced in version 4.5 of the .NET Framework.

**Exception Handling and Caveats**

The TPL has a very powerful set of APIs, but its extreme flexibility can have some drawbacks. As an example, let's look briefly at exception handling with the TPL. Tasks completely encapsulate their exceptions, meaning an exception that happens in a task's code does not interrupt execution of your application, so you can't just use try/catch from the caller. Instead, you must inspect the completed task status and other properties to see if it faulted and why. In complex applications with high degrees of parallelization (i.e. many threads running simultaneously), this exception encapsulation may be exactly what you want. If the task encountered any exceptions, an exception of type AggregateException will be set. You can iterate through the InnerExceptions of the aggregate and react accordingly.

if (task.Status == TaskStatus.Faulted && task.Exception != null)

{

foreach (var ex in task.Exception.InnerExceptions)

{

Console.WriteLine($"Exception: {ex}");

}

}

Developers getting started with the TPL are often confused when their application behaves unexpectedly without any indication of an exception, so be sure to keep that in mind. You will almost always want to add some sort of logging, at a minimum.

Another aspect of the TPL that is less than ideal is that, in order to get a task's result, you typically need to set a callback—a method that is called when the task completes. The continuation lambdas we set via ContinueWith above are examples of this. The callback is a tried and true pattern used in asynchronous programming but, as we saw in our example, it can be a bit hard to read since each callback is indented and marked with additional braces and parentheses. Fortunately for C# developers, the async and await keywords were created in part to alleviate that exact problem.

The Task Parallel Library has proven itself to be extremely important. Not only has it made asynchronous programming more consistent, reliable and flexible for C# developers, it has also provided the foundation for a revolutionary approach to asynchronous programming at the language level, namely C#'s async and await keywords. The [next guide in this series](https://app.pluralsight.com/guides/csharp-async-await-keywords-getting-started) will explore how async and await built on the Task Parallel Library's success to make asynchronous programming even better.