Async Programming Questions

Friday, May 10, 2024 1:51 PM

Synchronous	Asynchronous
Each statement of code is executed one after the other, in sequence.	Tasks are executed in parallel with the main thread of execution.
When a statement is executed, the program waits for it to complete before moving on to the next statement.	Asynchronous code is designed to allow the main thread of execution to continue executing while the asynchronous tasks run in the background. (This can be achieved when you run task(s) without await command, like fire and forget).
Synchronous code is easy to read and understand, but it can lead to long wait times when dealing with time-consuming tasks.	Asynchronous code is usually more complex than synchronous code, but it can provide better performance and responsiveness.
	Asynchronous code is useful when dealing with time-consuming tasks, such as network or disk I/O, because it allows the main thread of execution to continue running while waiting for the task to complete. Ex : Imagine kicking off multiple N/W tasks (without await) and later wait for all tasks.

Here, the main thread has to wait until it gets back data.
 	
byte[] data = new WebClient().DownloadData("http://example.com/file.txt");
Console.WriteLine("Downloaded {0} bytes.", data.Length);
	Here the main thread creates a task and also waits for the task to finish (since we are using await).
	
	async Task DownloadFileAsync(string url)
	{
	  using (HttpClient client = new HttpClient())
	  {
	  byte[] data = **await** client.GetByteArrayAsync(url);
	  Console.WriteLine("Downloaded {0} bytes.", data.Length);
	  }
	}
	
	In both cases, the result is same as we are waiting for N/W call to finish.

	Public static Task<string> DoAsyncResult(stringitem)
	{
	  Task.Delay(1000);
	  Return Task.FromResult(item);
	}
	
	publicstaticasyncTask<IEnumerable<string>> LoopAsyncResult(IEnumerable<string> thingsToLoop)
	{
	  List<Task<string>> listOfTasks = newList<Task<string>>();
	  foreach(var thing in thingsToLoop)
	  listOfTasks.Add(DoAsyncResult(thing));
	  Return await Task.WhenAll<string>(listOfTasks);
	}
	
	Here, all the tasks are being kicked off without awaiting. These tasks will run in **parallel**.
	If the same thing is done using synchronous approach, we would have to run the task in sequence.

Thread vs Task

"Thread" and "Task" are two different ways to work with concurrent execution

Thread	Task
Threads are a **low-level construct** for concurrent programming. They are part of the System.Threading namespace and are provided by the **operating system**.	Tasks are a **higher-level abstraction** introduced in **.NET** to simplify asynchronous and parallel programming. They are part of the Task Parallel Library (TPL)
Creating and managing threads can be **resource-intensive**, as they require their own memory allocation and kernel resources.	Tasks are more **lightweight** than threads because they are built on top of thread pool threads. They are more **efficient in terms of resource usage**.
Threads require manual management of thread creation, synchronization, and termination. This can make code more complex and error-prone.	The TPL manages tasks automatically, making it easier to create, schedule, and manage parallel and asynchronous operations.
Threads can be **blocking**. When a thread performs a time-consuming operation, it can block the entire application until the operation is complete.	Tasks are generally **non-blocking**, meaning they allow you to perform other work while waiting for an operation to complete, making your application more responsive.