# From Book: Concurrency in C# Cookbook: Asynchronous, Parallel, and Multithreaded Programming

1. Concurrency an overview (Important terms):

* Concurrency:
  + Concurrency: Doing more than one thing at a time.
* Multithreading:
  + Multithreading: A form of concurrency that uses multiple threads of execution.
  + Multithreading means apply concurrency in form of multi threads.
  + In fact, direct use of the low-level threading types has almost no purpose in a modern 1 application; higher-level abstractions are more powerful and more efficient than old-school multithreading.
  + As soon as you type new Thread() or BackgroundWorker, it’s over; your project already has legacy code.
  + But don’t get the idea that multithreading is dead! Multithreading lives on in the thread pool, a useful place to queue work that automatically adjusts itself according to demand. In turn, the thread pool enables another important form of concurrency: parallel processing.
* Parallel Processing:
  + Parallel Processing: Doing lots of work by dividing it up among multiple threads that run concurrently.
  + Parallel processing (or parallel programming) uses multithreading to maximize the use of multiple processors. Modern CPUs have multiple cores, and if there’s a lot of work to do, then it makes no sense to just make one core do all the work while the others sit idle. Parallel processing will split up the work among multiple threads, which can each run independently on a different core.
* Parallel processing is one type of multithreading, and multithreading is one type of concurrency. There’s another type of concurrency that is important in modern applications but is not (currently) familiar to many developers: asynchronous programming.
* Asynchronous Programming:
  + Asynchronous Programming: A form of concurrency that uses futures or callbacks to avoid unnecessary threads.
  + A future (or promise) is a type that represents some operation that will complete in the future.
  + The modern future types in .NET are Task and Task. Older asynchronous APIs use callbacks or events instead of futures.
  + Asynchronous programming is centered around the idea of an asynchronous operation.
  + asynchronous operation: some operation that is started that will complete some time later. While the operation is in progress, it does not block the original thread; the thread that starts the operation is free to do other work. When the operation completes, it notifies its future or invokes its completion callback event to let the application know the operation is finished.
  + Asynchronous programming is a powerful form of concurrency, but until recently, it required extremely complex code. The async and await support in VS2012 make asynchronous programming almost as easy as synchronous (nonconcurrent) programming.
* Reactive Programming:
  + Another form of concurrency is reactive programming. Asynchronous programming implies that the application will start an operation that will complete once at a later time. Reactive programming is closely related to asynchronous programming, but is built on asynchronous events instead of asynchronous operations. Asynchronous events may not have an actual “start,” may happen at any time, and may be raised multiple times. One example is user input.
  + Reactive Programming A declarative style of programming where the application reacts to events.
  + If you consider an application to be a massive state machine, the application’s behavior can be described as reacting to a series of events by updating its state at each event. This is not as abstract or theoretical as it sounds; modern frameworks make this approach quite useful in real-world applications. Reactive programming is not necessarily con‐ current, but it is closely related to concurrency, so we’ll be covering the basics.
* Usually, a mixture of techniques is used in a concurrent program. Most applications at least use multithreading (via the thread pool) and asynchronous programming. Feel free to mix and match all the various forms of concurrency, using the appropriate tool for each part of the application.

1. **Introduction to Asynchronous Programming:**

**2-1-** [**https://docs.microsoft.com/en-us/dotnet/csharp/async**](https://docs.microsoft.com/en-us/dotnet/csharp/async) **conclusion:**

If you have any I/O-bound needs (such as requesting data from a network or accessing a database), you'll want to utilize asynchronous programming. You could also have CPU-bound code, such as performing an expensive calculation, which is also a good scenario for writing async code.

C# has a language-level asynchronous programming model which allows for easily writing asynchronous code without having to juggle callbacks or conform to a library which supports asynchrony. It follows what is known as the [Task-based Asynchronous Pattern (TAP)](https://msdn.microsoft.com/library/hh873175.aspx).

## **2-1-1- Basic Overview of the Asynchronous Model:**

The core of async programming are the Task and Task<T> objects, which model asynchronous operations. They are supported by the asyncand await keywords. The model is fairly simple in most cases:

**For I/O-bound code**, you await an operation which returns a Task or Task<T> inside of an async method.

**For CPU-bound code**, you await an operation which is started on a background thread with the Task.Run method.

The await keyword is where the magic happens. It yields control to the caller of the method that performed await, and it ultimately allows a UI to be responsive or a service to be elastic.

There are other ways to approach async code than async and await, but here we will focus on the language-level constructs from this point forward.

## **2-1-2- Example for I/O-Bound Vs. CPU Bound:**



**2-2- Async programming benefits for the GUI programs and Server-Side programs:**

* GUI: An asynchronous program can remain responsive to user input while it’s working.
* Server-Side: The second benefit is for server-side programs: asynchronous programming enables scalability. A server application can scale somewhat just by using the thread pool, but an asynchronous server application can usually scale an order of magnitude better than that.

**2-3- asp.net (IIS) and async and await:**

* In normal IIS use thread pool to handle the concurrent requests.
* Async / await in asp.net is not about concurrency, it's about blocking or not blocking threads.
* If you use async / await you release the thread while you're waiting for an operation. If this operation is CPU-bound, there's no benefit (it'll be even slightly slower because of the context switching)
* If the operation is IO-bound though (network, disk, ...) it means that IIS can handle more concurrent requests, since you're not blocking any threads that are doing nothing.
* For the same sessions, the concurrent requests are queued because the ASP.NET (MVC not sure about Web Forms)mark session object as thread-safe using some locking techniques which make the concurrent requests queued and not run concurrently so we can decrease the locking period to enhance performance or implement a lockless sessions.

**2-4- Understand the .Net CLR thread pool:**

**2-4-1- Intro:**

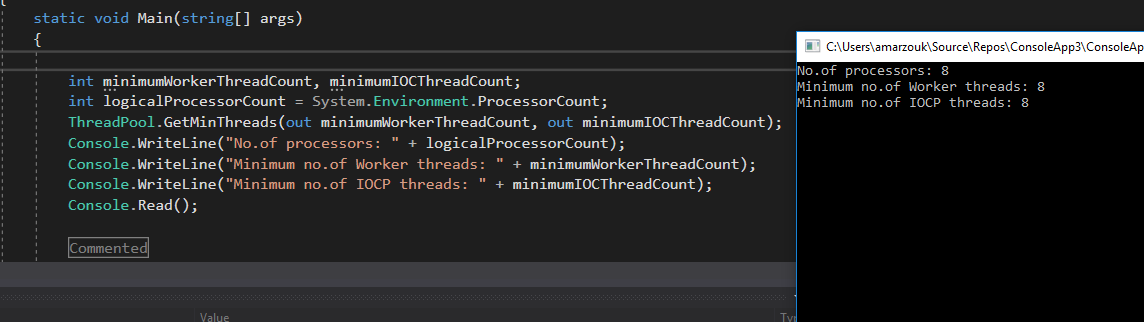
In the .Net Framework, the CLR is responsible for meting out resources to running applications. In particular, the CLR thread pool determines when threads are to be added or taken away. Understanding how this works will help you determine how to configure your ASP.Net application for optimal performance.

The CLR thread pool contains two kinds of threads—the worker threads and the I/O completion port or IOCP threads. That means your ASP.Net worker process contains two thread pools: the worker thread pool and the IOCP thread pool. Naturally, these pools have different purposes.

When you use methods like Task.Run, TaskFactory.StartNew, and ThreadPool.QueueUserWorkItem, the runtime takes advantage of worker threads for processing. When you make asynchronous I/O calls in your application, or your application accesses the file system, databases, web services, etc., then the runtime uses IOCP threads. Note too that each application domain has its own thread pool.

**2-4-1- Thread Injection Strategies:**

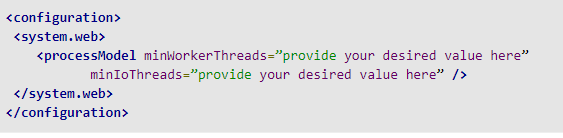
* The .Net thread pool starts injecting new threads whenever the number of busy threads becomes equal to the number of configured minimum threads in the thread pool.
* The default value of the minimum setting, which is the minimum number of both worker and IOCP threads, is determined by the number of processors in your system. Hence, if your system has four cores, you would have four worker threads and four IOCP threads by default.
* The .Net thread pool then injects additional worker threads on demand if existing threads are utilized and there is still work to be done. By the same token, if the demand for resources falls, the thread pool will begin taking threads away.
* Executing the following code snippet would display the number of logical processors in your system and the minimum number of worker and IOCP threads available.



* The .Net thread pool manages threads using its built-in heuristics. The strategies adopted include starvation avoidance and a hill-climbing algorithm. In the former case, the .Net thread pool continues to add worker threads if there is no visible progress on the queued items. In the latter case, the .Net thread pool tries to maximize the throughput using as few threads as possible.
* The .Net thread pool injects or removes threads at intervals of 500 milliseconds or as a thread becomes free, whichever comes first. Now, based on the feedback available to the runtime, the .Net thread pool either removes threads or adds threads to maximize the throughput. If adding a thread does not increase throughput, it takes a thread away. This is the CLR’s hill-climbing technique in action.
* Now suppose you are running your ASP.Net application on IIS and your web server has a total of four CPUs. Assume that at any given point in time, there are 24 requests to be processed. By default, the runtime would create four threads, which would be available to service the first four requests. Because no additional threads will be added until 500 milliseconds have elapsed, the other 20 requests will have to wait in the queue. After 500 milliseconds have passed, a new thread is created.
* As you can see, it will take many 500ms intervals to catch up with the workload. This is a good reason for using asynchronous programming. With async programming, threads aren’t blocked while requests are being handled, so the four threads would be freed up almost immediately.

**2-4-2- Recommended thread settings:**

* Given the way the .Net thread pool works and what we have discussed thus far, it is strongly recommended that you change the minimum configuration value—the default value—for both worker and IOCP threads. To do this in ASP.Net, you should change the minWorkerThreads and minIoThreads configuration settings under the <processModel> configuration element in the machine.config file in your system.



* You can set the minimum configuration values for both worker and IOCP threads to any value between one and 50. A good approach is to take a user mode process dump of the IIS worker process (W3wp.exe) and then use the !threadpool command to report the total number of worker threads. Once you know this value, simply divide it by the number of processor cores on your system to determine the minimum worker and IOCP thread settings. For example, if the total count of worker threads is 100 and you have four processors in your system, you can set the minimum values for both worker and IOCP threads to 25.
* To change the default minimum thread settings outside of ASP.Net, you can use the ThreadPool.SetMinThreads() method.
* With the goal of better thread management and improved performance, the CLR thread pool has been improved with each version of the CLR. As an example, with .Net Framework 4, the CLR gained thread stealing algorithms and support for concurrency and parallelism. With each new version of the CLR, the .Net thread pool is getting smarter about optimizing the throughput by creating and destroying threads as needed. In the meantime, you’ll want to experiment with different minimum thread settings to get the best performance from your .Net application.

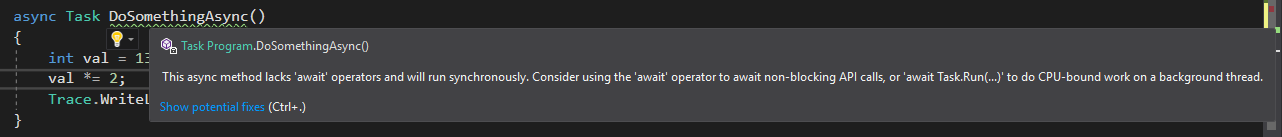
**2-5- Understanding async await:**

**2-5-1- Intro:**

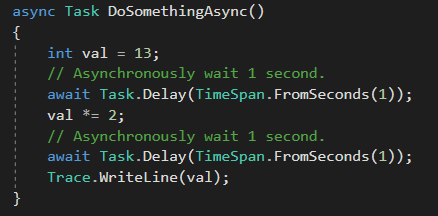
* As we illustrated before, async and await can be used to enable asynchronous (not necessary to be multithreaded) programming for example for something like I/O bounded or for CPU-bounded.
* Modern asynchronous .NET applications use two keywords: **async** and **await**. The async keyword is added to a method declaration, and its primary purpose is to enable the await keyword within that method (the keywords were introduced as a pair for backward-compatibility reasons). An async method should return **Task<T>** if it returns a value, or **Task** if it does not return a value. These task types represent **futures**; they notify the calling code when the async method completes.
* Note: **Avoid async void!** It is possible to have an async method return void, but you should only do this if you’re writing an async event handler. A regular async method without a return value should return Task, not void.

**2-5-2- Async Await anatomy:**

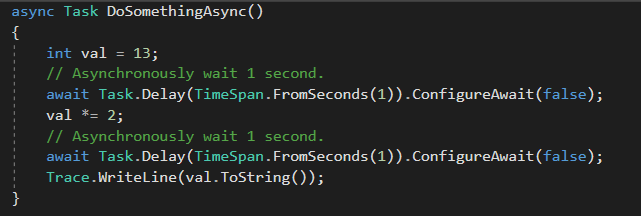
* If you didn’t write await that’s a clearly description about what’s happened:



* **For the below example:**



* An async method begins executing synchronously, just like any other method. Within an async method, the await keyword performs an asynchronous wait on its argument. First, it checks whether the operation is already complete; if it is, it continues executing (synchronously). Otherwise, it will pause the async method and return an incomplete task. When that operation completes some time later, the async method will resume executing.
* You can think of an async method as having several synchronous portions, broken up by await statements. The first synchronous portion executes on whatever thread calls the method, **but where do the other synchronous portions execute? The answer is a bit complicated.**
* - When you await a task (the most common scenario), a context is captured when the await decides to pause the method. This context is the current SynchronizationContext unless it is null, in which case the context is the current TaskScheduler. The method resumes executing within that captured context. Usually, this context is the UI context (if you’re on the UI thread), an ASP.NET request context (if you’re processing an ASP.NET request), or the thread pool context (most other situations). So, in the preceding code, all the synchronous portions will attempt to resume on the original context. If you call DoSomethingAsync from a UI thread, each of its synchronous portions will run on that UI thread; but if you call it from a thread-pool thread, each of its synchronous portions will run on a thread-pool thread. You can avoid this default behavior by awaiting the result of the ConfigureAwait extension method and passing false for the continueOnCapturedContext parameter. The following code will start on the calling thread, and after it is paused by an await, it will resume on a thread-pool thread:



* It’s good practice to always call ConfigureAwait in your core “library” methods, and only resume the context when you need it—in your outer “user interface” methods.

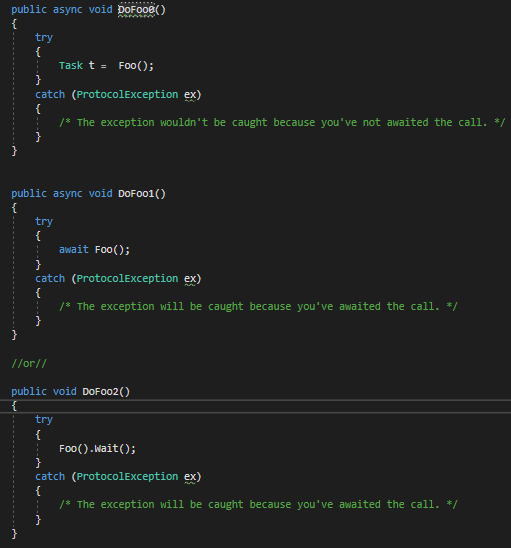
**2-5-3- await keyword can works with anything follow the awaitable pattern not only with Task or Task<T> like in Windows store case but most of times we will works with Tasks.**

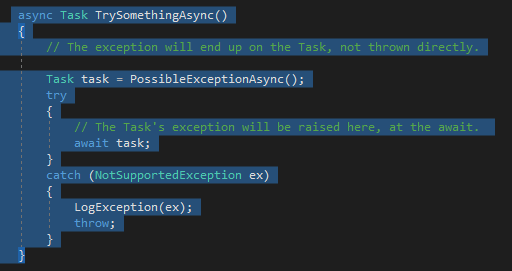
**2-5-4- There are two basic ways to create a Task instance:**

* There are two basic ways to create a Task instance. Some tasks represent actual code that a CPU has to execute; these computational tasks should be created by calling Task.Run (or TaskFactory.StartNew if you need them to run on a particular scheduler). Other tasks represent a notification; these event-based tasks are created by TaskComple tionSource<T> (or one of its shortcuts). Most I/O tasks use TaskCompletionSource<T>.

**2-5-4- Error Handling:**

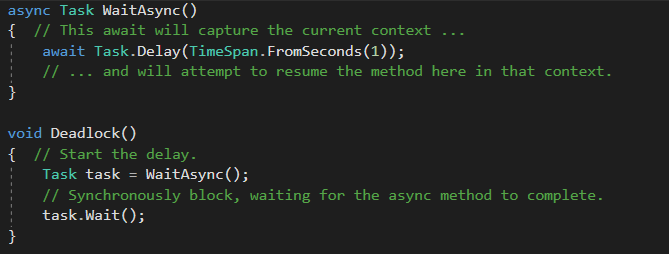
* In short, if there is an exception it will throw as expected but when you used await keyword and if you don’t used it the returned Task will contains the exception but not thrown it.





**2-5-5- Deadlock may be happen:**

* There’s one other important guideline when it comes to async methods: once you start using async, it’s best to allow it to grow through your code. If you call an async method, you should (eventually) await the task it returns. Resist the temptation of calling Task.Wait or Task<T>.Result; this could cause a deadlock. Consider this method:
* If you use async, it’s best to use async all the way.
* For example:



* This code will deadlock if called from a UI or ASP.NET context. This is because both of those contexts only allow one thread in at a time. Deadlock will call WaitAsync, which begins the delay. Deadlock then (synchronously) waits for that method to complete, blocking the context thread. When the delay completes, await attempts to resume.
* WaitAsync within the captured context, but it cannot because there is already a thread blocked in the context, and the context only allows one thread at a time. Deadlock can be prevented two ways: you can use ConfigureAwait(false) within WaitAsync (which causes await to ignore its context), or you can await the call to WaitAsync (making Deadlock into an async method).