Concurrency programming in .Net C#

This document will explain 3 different methods available for concurrency

In .NET C#, there are several concurrency options available beyond Parallel, Threads, and Task.WhenAll. These include:

* **SemaphoreSlim**: This can be used to limit the number of concurrent operations by controlling access to a resource. It allows you to wait for a signal before proceeding, which can be useful for throttling the number of tasks running at the same time.
* **Parallel.ForEach**: This method is used for parallel iteration over a collection. It can be configured with a MaxDegreeOfParallelism to control the number of concurrent operations. However, it is designed for synchronous operations and should be used with caution with asynchronous code.
* **Bulkhead policy from Polly**: This is a pattern for isolating failures and limiting the number of concurrent calls. It can be used to ensure that a certain number of operations are executed concurrently, and it can also queue up additional calls if the limit is reached.
* **TPL Dataflow**: This is a library for building data processing pipelines. It provides blocks that can be used to control the flow of data and the number of concurrent operations. For example, the ActionBlock can be used to process items in a pipeline with a specified degree of parallelism.
* **BlockingCollection**: This is a thread-safe collection that can be used for producer-consumer scenarios. It can be used to control the number of concurrent operations by limiting the number of items that can be added to the collection at any one time.
* **Task.Run**: This method is used to run a task on a thread pool thread. It can be used to execute code asynchronously and can be combined with other concurrency constructs to control the number of concurrent operations.
* **Task.WhenAll**: This method is used to wait for all tasks to complete. It can be used to execute multiple tasks in parallel and wait for them to finish.
* **Threads/ThreadStart**

These are just a few of the concurrency options available in .NET C#. The choice of which one to use depends on the specific requirements of the application.

**--Exploring BlockingCollection in C#: A Comprehensive Guide**

--System.Collections.Concurrent

READING:

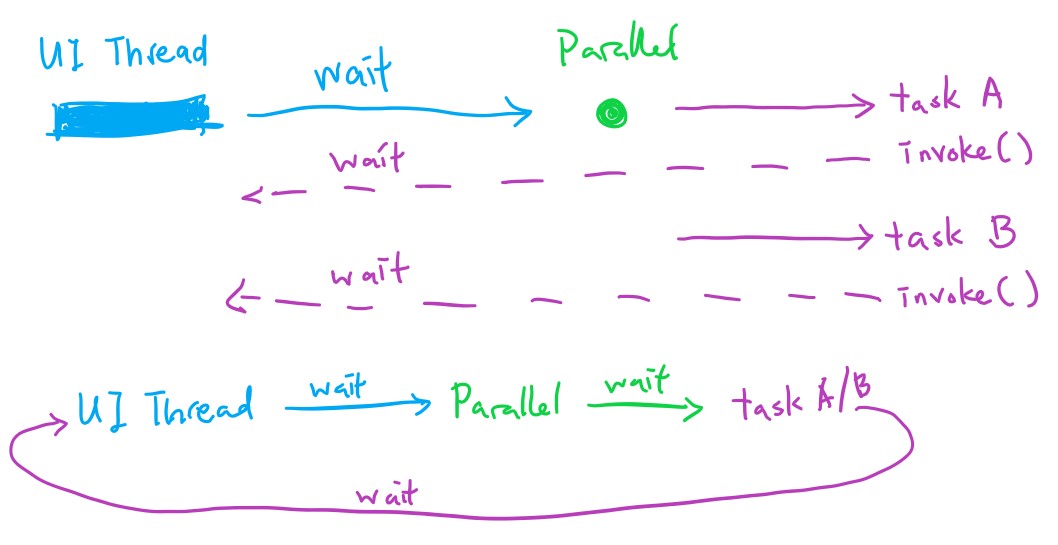
How to find and limit number of cores in system for usage

<https://stackoverflow.com/questions/1542213/how-to-find-the-number-of-cpu-cores-via-net-c>

<https://stackoverflow.com/questions/5512312/is-it-possible-to-limit-the-cores-for-parallel-foreach>

# Parallel Deadlock

Using Invoke() from UI main thread in Parallel Task() cause deadlock.



Why is is deadlocked?

1. UI thread calls Parallel() creating the Tasks(), stops moving until Parallel is complete.

2. Inside Tasks progress update text in UI so call Invoke() to tell UI thread to update the textbox, stops moving until UI updates the text

3. Deadlock/Cyclic dependency = UI is waiting for Parallel(Tasks) to complete, Tasks is waiting UI to do something. UI can’t move because of Tasks, Tasks can’t move because of Invoke() to UI.

## Solution to Parallel Deadlock

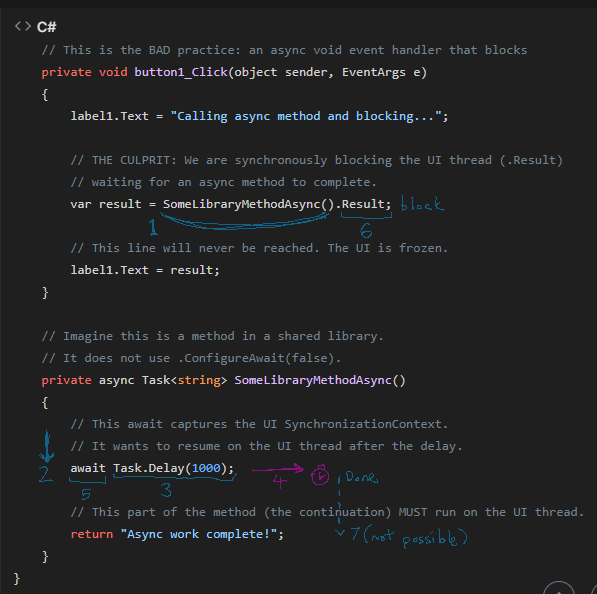
1. use BeginInvoke() instead of Invoke()

2. async/await Task.Run call Parallel

3. Thread/ThreadStart – note: Task() is more performant for modern .Net code

# Task Deadlock

Task is part of the TPL library from .net, it is used async/await by design so it’s still used within 1 thread.



The deadlock scenario.

1. SomeLibMethod() is called
2. Task.Delay(1000) is called, this creates a background timer to run, then await goes back to original caller. Which is step 1
3. .Result is now waiting/blocking because await was called.
4. Background timer is done counting, returning to main thread to try to continue to the next line (line: return Async…)
5. LOCKED: main thread .Result is waiting on Task to complete, SomeLibMethod() is waiting on main thread to move forward. The main thread cursor is now logically stuck.

## Solution Emphasis using ConfigureAwait()

Using .ConfigureAwait(false); will eliminate the deadlock scenario above.

await Task.Delay(500).ConfigureAwait(false);

Once configurewait() is called, the lines after this is completed with a new thread from the thread pool. So main thread is no longer locked because the SomeLibMethod() can return by using another thread. Which means .Result will return to allow main thread to finish the rest of the code

# Task Deadlock #2

Another example for Task deadlock, any time async/await

TODO document these items:

1. Interlocked eg. Interlocked.Add, etc…
2. SemaphoreSlim,
   1. Release() is cumulative, need to track number released so far out of total allowed
3. Invoke vs BeginInvoke
   1. MethodInvoke() – faster but env specific vs new Action() – slower but wider use, this can also be used as a Callback system

*// BeginInvoke(new Action<string>(updateLabelMain), msg); // slowest*

*// BeginInvoke(new MethodInvoker(() => updateLabelMain(msg))); // slower*

*// BeginInvoke((MethodInvoker)(() => updateLabelMain(msg))); // medium*

*// BeginInvoke(() => updateLabelMain(msg)); // fastest + modern way*

1. AsyncManualResetEvent
2. CancellationTokenSource
3. ManualResetEventSlim vs ManualResetEvent (and what each of them is)
4. Parallel() vs Task() vs Thread()
5. TaskScheduler
6. SynchronizationContext
7. MaxDegreeOfParallelism