

Concurrent Programming in .NET

Jason Bock

Personal Info

- <http://www.jasonbock.net>
- <https://www.twitter.com/jasonbock>
- <https://www.github.com/jasonbock>
- jason.r.bock@outlook.com

Downloads

<https://github.com/JasonBock/ConcurrentProgramming>

<https://github.com/JasonBock/ExpressionEvolver>

<https://github.com/JasonBock/Presentations>

Overview

- Background
- Recommendations

Remember...

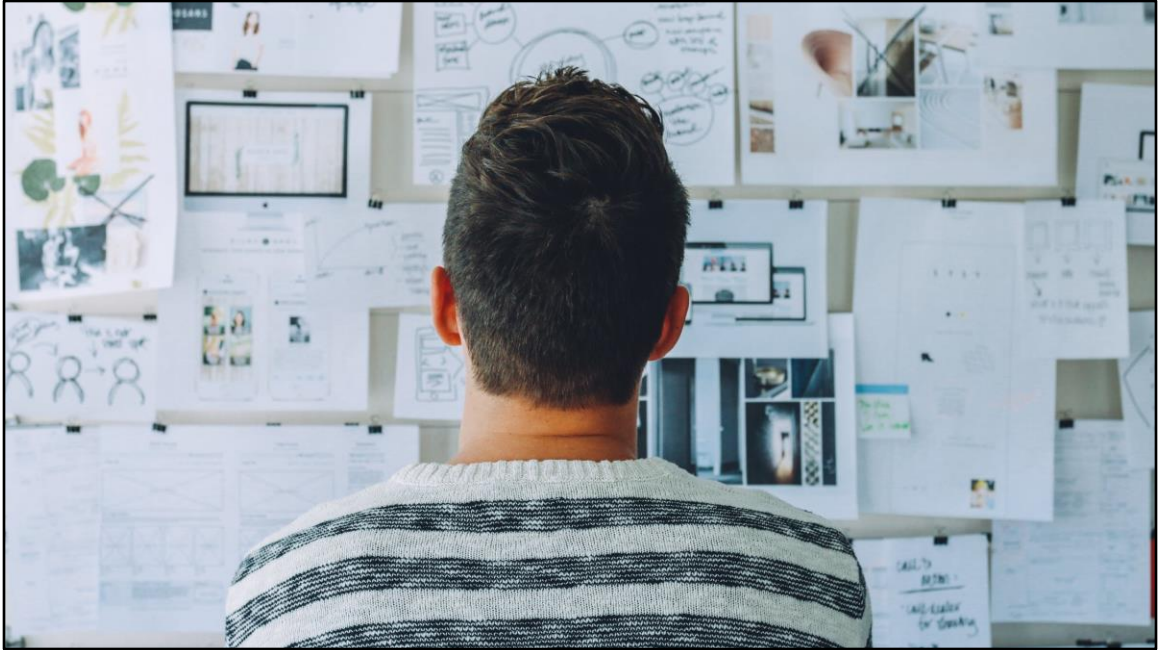
<https://github.com/JasonBock/ConcurrentProgramming>

<https://github.com/JasonBock/ExpressionEvolver>

<https://github.com/JasonBock/Presentations>

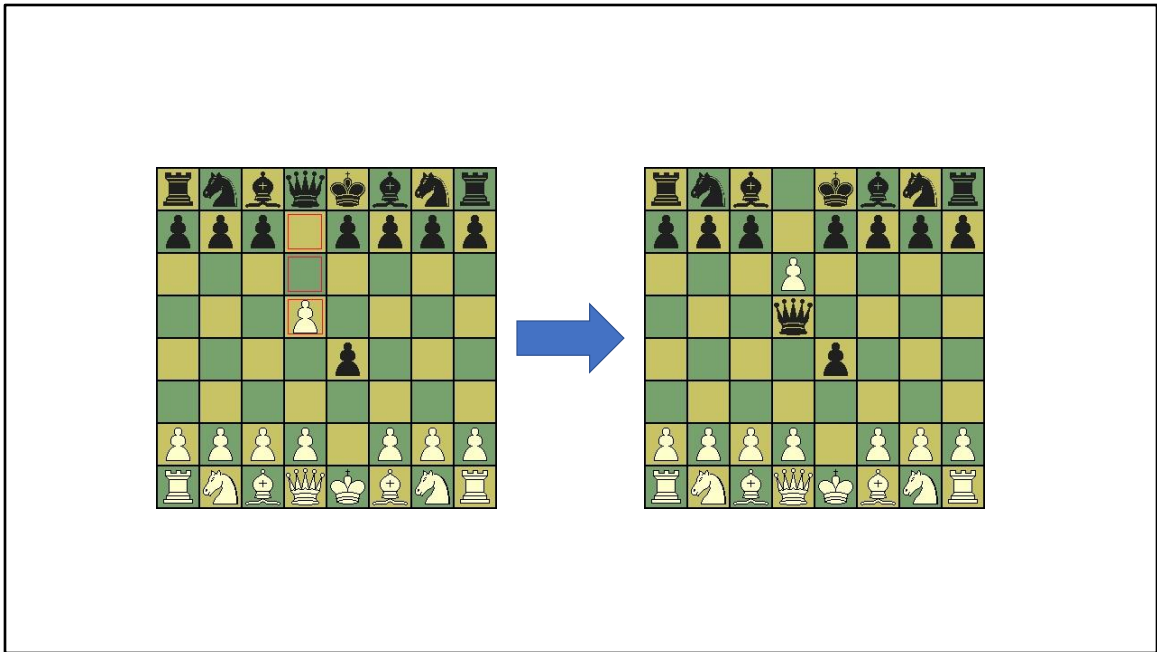
Background

Concurrent Programming in .NET



Writing concurrent applications safely may feel complicated at times. Even with modern techniques, it's still something you have to think about carefully.

<https://www.pexels.com/photo/man-wearing-black-and-white-stripe-shirt-looking-at-white-printer-papers-on-the-wall-212286/>



Lots of processes that we do have been traditionally thought of as sequential. But what happens if we make things, like games, happen the way our universe works: concurrently?

(Side note: people may look at the starting position and wonder how in the world we got there. First move, W to E4 and B to D5. Second move, they try to take each other and move right through each other 😊).

<http://www.jasonbock.net/Article/ConcurrentChess>



Whatever we do, let's make it as easy as we can without breaking stuff badly or making performance tank.

https://unsplash.com/photos/_SEbdtH4ZLM



But first, a comparison of asynchronous, parallel and concurrent work.

“Parallel programs distribute their tasks to multiple processors, that actively work on all of them simultaneously.”

<https://www.quora.com/What-are-the-differences-between-parallel-concurrent-and-asynchronous-programming>

<https://unsplash.com/photos/CQmnJ2-ODIQ>



“Concurrent programs handle tasks that are all in progress at the same time, but it is only necessary to work briefly and separately on each task, so the work can be interleaved in whatever order the tasks require.”

<https://unsplash.com/photos/hiFZxXC6pGw>



“An asynchronous program dispatches tasks to devices that can take care of themselves, leaving the program free to do something else until it receives a signal that the results are finished.”

Recommendations

Concurrent Programming in .NET

Avoid direct usage of Thread and ThreadPool

Understand async/await/Task

Use locks wisely

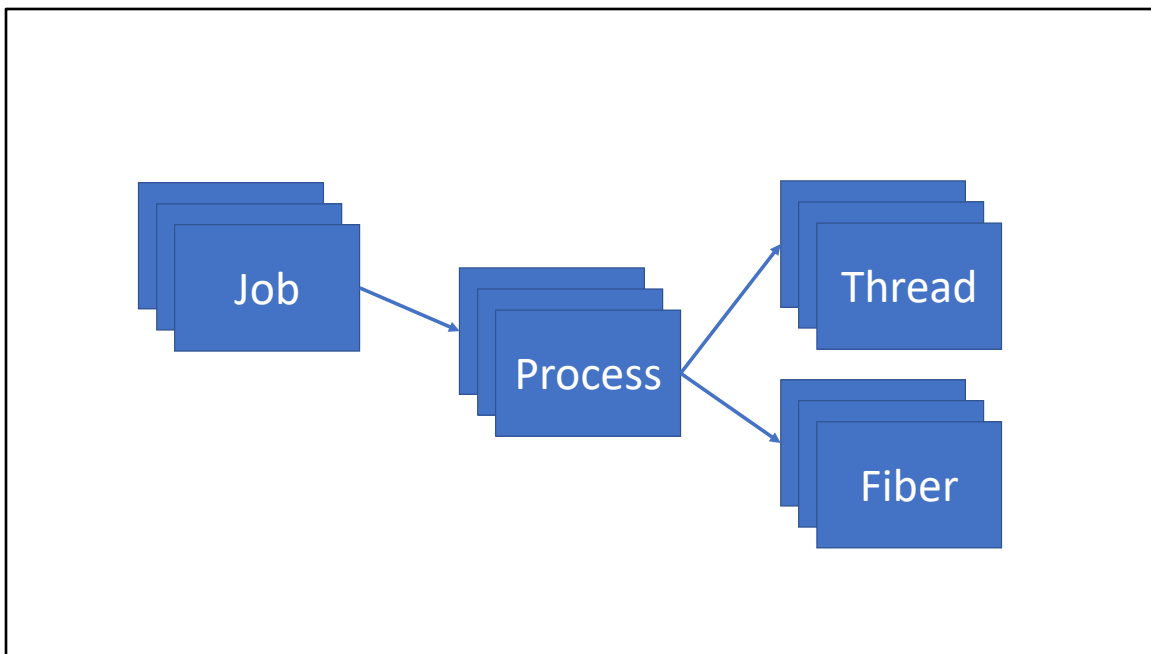
Use concurrent and immutable data structures

Let someone else worry about concurrency (actors)



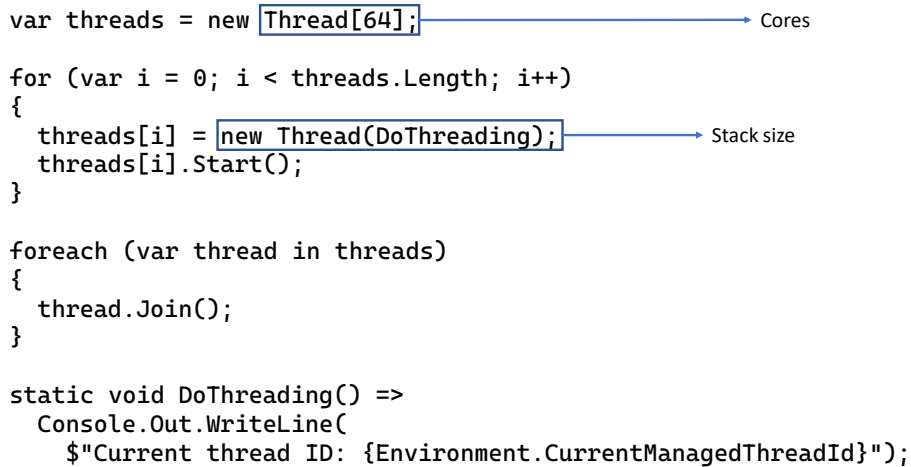
First off, STOP creating threads explicitly! But why?

<https://unsplash.com/photos/Nh6NsnqYVsl>



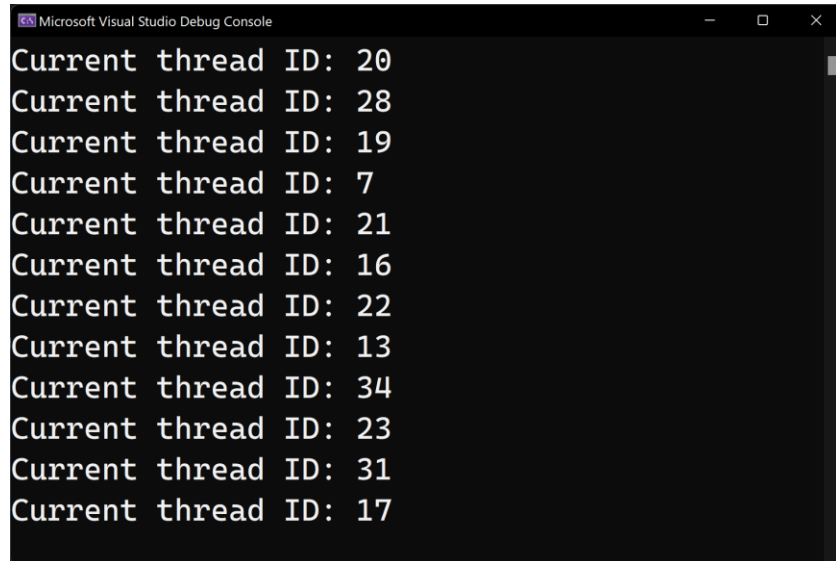
A brief primer. Most people think of programs in Windows as being processes and threads. There are also jobs and fibers, but they're rarely if ever used.

```
var threads = new Thread[64];  
for (var i = 0; i < threads.Length; i++)  
{  
    threads[i] = new Thread(DoThreading);  
    threads[i].Start();  
}  
  
foreach (var thread in threads)  
{  
    thread.Join();  
}  
  
static void DoThreading() =>  
    Console.Out.WriteLine(  
        $"Current thread ID: {Environment.CurrentManagedThreadId}");
```



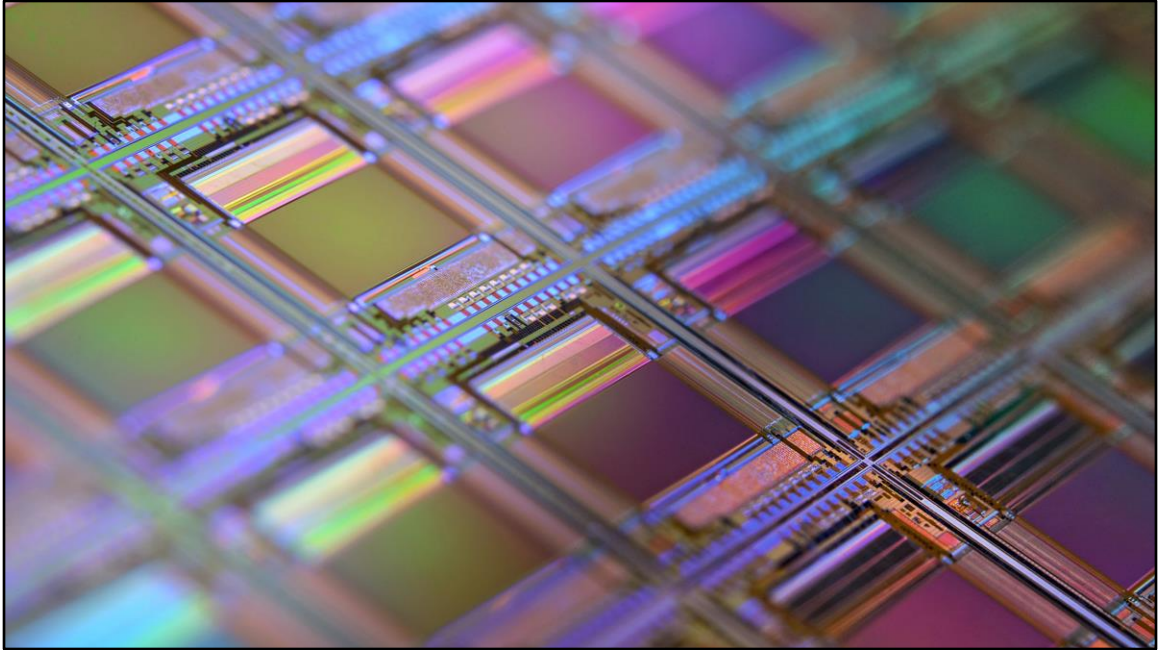
You can easily create threads, but there's two problems here: the stack size and the optimum number of threads.

Note that I don't use `Thread.CurrentThread.ManagedThreadId`. CA1840 suggests the `Environment` approach: <https://docs.microsoft.com/en-us/dotnet/fundamentals/code-analysis/quality-rules/ca1840>

A screenshot of the Microsoft Visual Studio Debug Console window. The window has a title bar with the text "Microsoft Visual Studio Debug Console" and standard window controls (minimize, maximize, close). The console area is dark with white text. It displays 12 lines of output, each starting with "Current thread ID:" followed by a number. The numbers are: 20, 28, 19, 7, 21, 16, 22, 13, 34, 23, 31, and 17. A vertical scrollbar is visible on the right side of the console area.

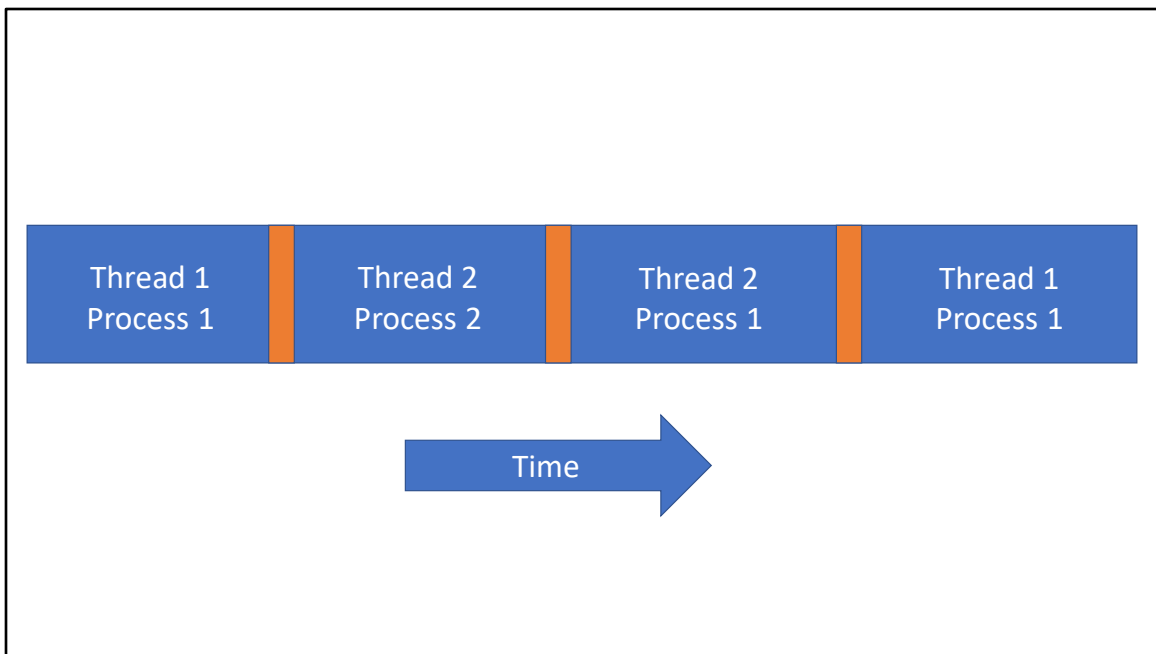
```
Current thread ID: 20
Current thread ID: 28
Current thread ID: 19
Current thread ID: 7
Current thread ID: 21
Current thread ID: 16
Current thread ID: 22
Current thread ID: 13
Current thread ID: 34
Current thread ID: 23
Current thread ID: 31
Current thread ID: 17
```

You can create 64 threads, but you should look at your hardware first.

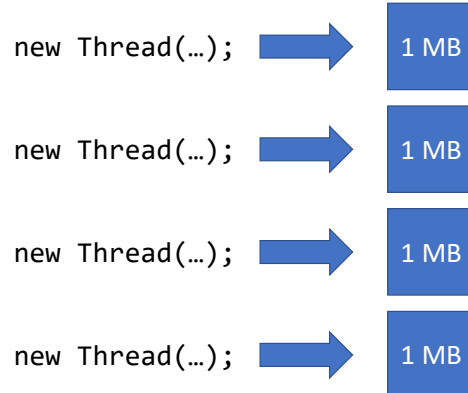


One is create the “right” amount of threads, which should be equal to the number of cores.

<https://unsplash.com/photos/qOx9KsvpqcM>



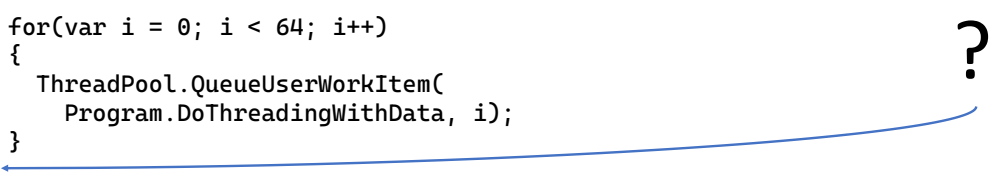
A thread can't dominate a core. The OS will switch out threads so they can all work as needed. Those little boxes in between are context switches, so you can see that they can add up over time.



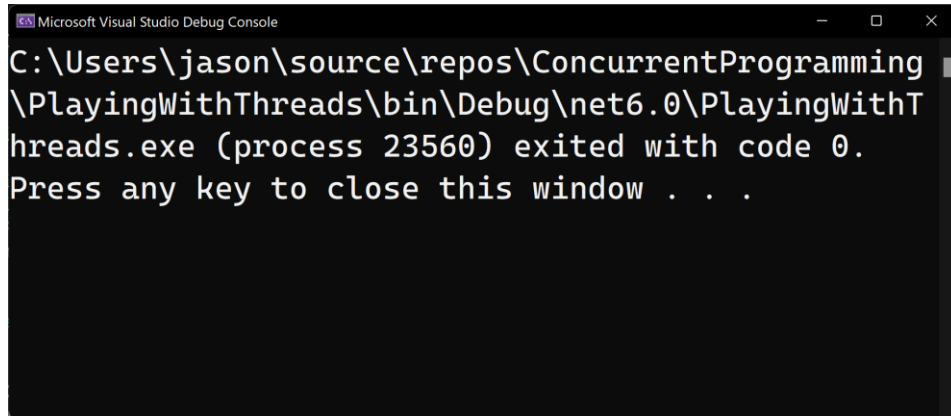
Also, by default, each thread allocates 1 MB of memory on the stack. If you keep making more, you'll burn memory. If you don't reuse them, that's a lot of memory to keep re-allocating.

```
static void CreateNewThreadsViaPool()
{
    for(var i = 0; i < 64; i++)
    {
        ThreadPool.QueueUserWorkItem(
            Program.DoThreadingWithData, i);
    }
}

static void DoThreadingWithData(object id) =>
    Console.Out.WriteLine(
        $"Current id: {id}, thread ID: {Environment.CurrentManagedThreadId}");
```



Using the ThreadPool allows threads to be shared/pooled, so you're not creating actual new threads in .NET. However, note that there is only one object that can be passed to WaitCallback. If you want to pass wait mechanisms to the thread, you have to create a custom object to pass it or share it outside of the method execution.

A screenshot of the Microsoft Visual Studio Debug Console window. The window has a title bar with the text "Microsoft Visual Studio Debug Console" and standard window controls (minimize, maximize, close). The console output is as follows:

```
C:\Users\jason\source\repos\ConcurrentProgramming  
\PlayingWithThreads\bin\Debug\net6.0\PlayingWithT  
hreads.exe (process 23560) exited with code 0.  
Press any key to close this window . . .
```

Otherwise, you'll get output like this.

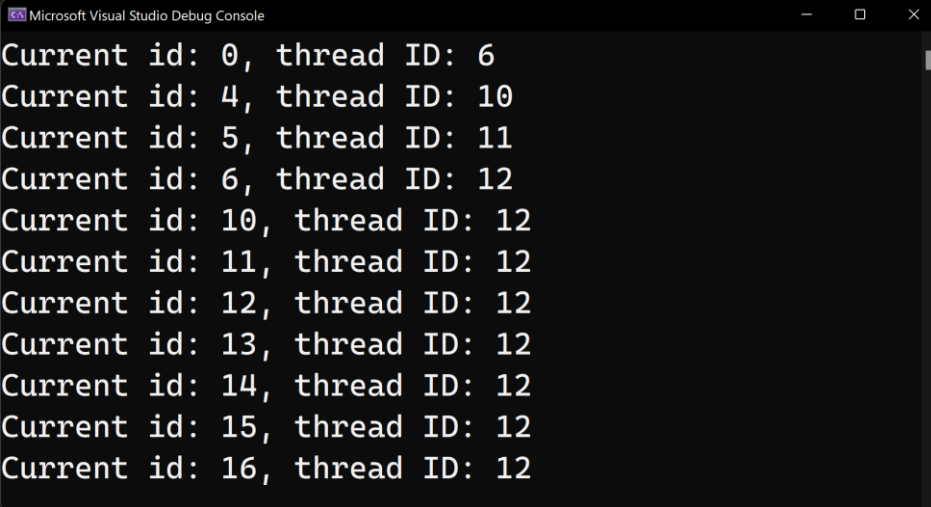
```
static void CreateNewThreadsViaPoolAndWait()
{
    var waits = new EventWaitHandle[64];

    for (var i = 0; i < 64; i++)
    {
        var wait = new ManualResetEvent(false);
        waits[i] = wait;
        ThreadPool.QueueUserWorkItem(
            Program.DoThreadingWithThreadingData, new ThreadingData(wait, i));
    }

    WaitHandle.WaitAll(waits);
}

static void DoThreadingWithThreadingData(object data)
{
    var value = data as ThreadingData;
    Console.Out.WriteLine(
        $"Current id: {value.Id}, thread ID: {Environment.CurrentManagedThreadId}");
    value.Wait.Set();
}
```

Here's one way you can communicate across threads. As you'll see later, using `async/await/Tasks` is much easier.

A screenshot of the Microsoft Visual Studio Debug Console window. The window has a title bar with the text "Microsoft Visual Studio Debug Console" and standard window controls (minimize, maximize, close). The console output shows a series of log messages in a monospaced font. The messages are: "Current id: 0, thread ID: 6", "Current id: 4, thread ID: 10", "Current id: 5, thread ID: 11", "Current id: 6, thread ID: 12", "Current id: 10, thread ID: 12", "Current id: 11, thread ID: 12", "Current id: 12, thread ID: 12", "Current id: 13, thread ID: 12", "Current id: 14, thread ID: 12", "Current id: 15, thread ID: 12", and "Current id: 16, thread ID: 12". The thread ID 12 is reused for multiple different current IDs.

```
Microsoft Visual Studio Debug Console
Current id: 0, thread ID: 6
Current id: 4, thread ID: 10
Current id: 5, thread ID: 11
Current id: 6, thread ID: 12
Current id: 10, thread ID: 12
Current id: 11, thread ID: 12
Current id: 12, thread ID: 12
Current id: 13, thread ID: 12
Current id: 14, thread ID: 12
Current id: 15, thread ID: 12
Current id: 16, thread ID: 12
```

Note that now we can see threads being reused.

Asynchronous Programming Model (APM)

Event-based Asynchronous Pattern (EAP)

Task-based Asynchronous Pattern (TAP)

Over the years different APMs showed up in .NET, but now we have Task, the primary abstraction.

APM: An asynchronous operation that uses the `IAsyncResult` design pattern is implemented as two methods named `Begin OperationName` and `End OperationName` that begin and end the asynchronous operation `OperationName` respectively. For example, the `FileStream` class provides the `BeginRead` and `EndRead` methods to asynchronously read bytes from a file. These methods implement the asynchronous version of the `Read` method.

EAP: A class that supports the Event-based Asynchronous Pattern will have one or more methods named `MethodNameAsync`. These methods may mirror synchronous versions, which perform the same operation on the current thread. The class may also have a `MethodNameCompleted` event and it may have a `MethodNameAsyncCancel` (or simply `CancelAsync`) method.

TAP: TAP uses a single method to represent the initiation and completion of an asynchronous operation. This is in contrast to the Asynchronous Programming Model (APM or `IAsyncResult`) pattern, which requires `Begin` and `End` methods, and in

contrast to the Event-based Asynchronous Pattern (EAP), which requires a method that has the Async suffix and also requires one or more events, event handler delegate types, and EventArgs-derived types. Asynchronous methods in TAP include the Async suffix after the operation name; for example, GetAsync for a get operation. If you're adding a TAP method to a class that already contains that method name with the Async suffix, use the suffix TaskAsync instead. For example, if the class already has a GetAsync method, use the name GetTaskAsync.

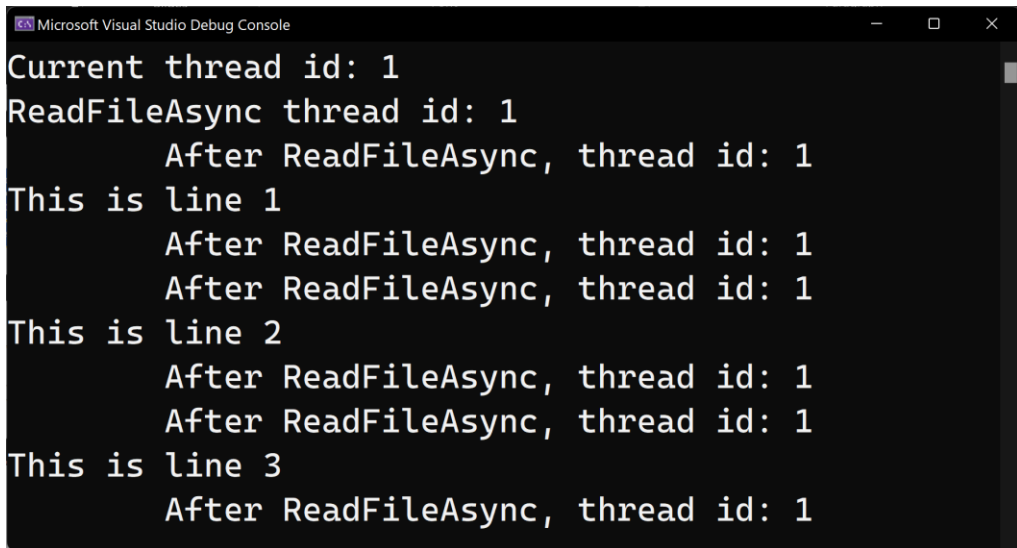
<https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/>

```
await Console.Out.WriteLineAsync(
    $"Current thread id: {Environment.CurrentManagedThreadId}");
await Console.Out.WriteLineAsync(
    $"{nameof(ReadFileAsync)} thread id: {Environment.CurrentManagedThreadId}");

using var stream = new StreamReader("lines.txt");

while (!stream.EndOfStream)
{
    var line = await stream.ReadLineAsync();
    await Console.Out.WriteLineAsync(
        $"\\tAfter stream.ReadLineAsync(), thread id: {Environment.CurrentManagedThreadId}");
    await Console.Out.WriteLineAsync(line);
    await Console.Out.WriteLineAsync(
        $"\\tAfter Console.Out.WriteLineAsync(), thread id: {Environment.CurrentManagedThreadId}");
}
```

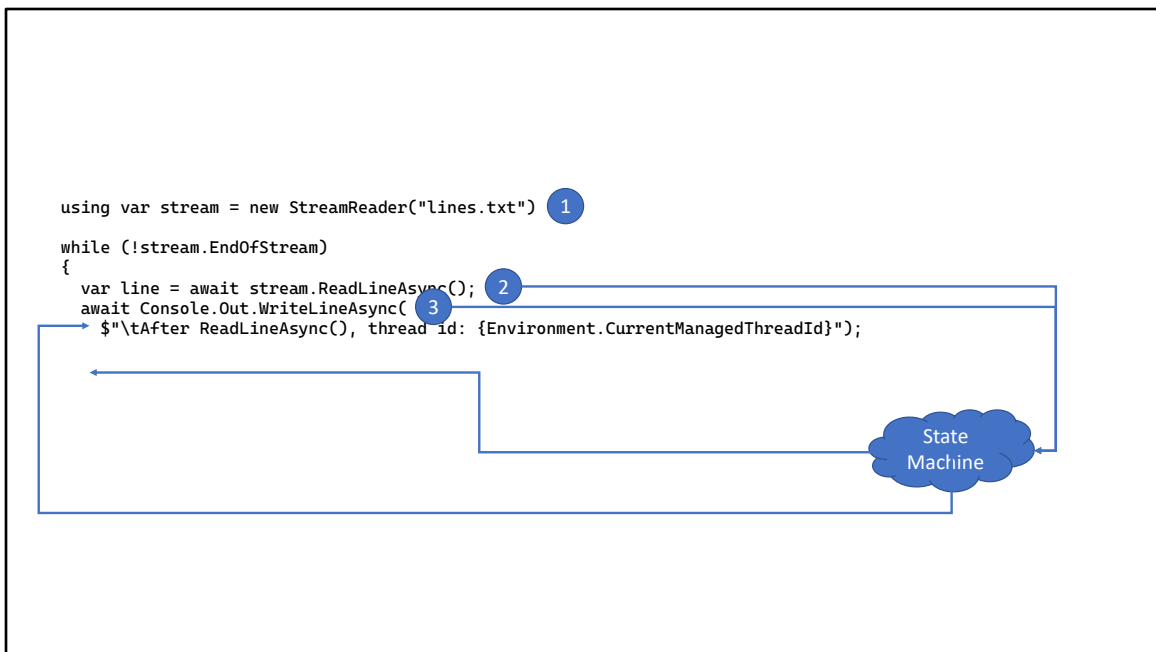
Let's see how async/await with Tasks work



```
Microsoft Visual Studio Debug Console

Current thread id: 1
ReadFileAsync thread id: 1
    After ReadFileAsync, thread id: 1
This is line 1
    After ReadFileAsync, thread id: 1
    After ReadFileAsync, thread id: 1
This is line 2
    After ReadFileAsync, thread id: 1
    After ReadFileAsync, thread id: 1
This is line 3
    After ReadFileAsync, thread id: 1
```

When I run this, the thread ID never changes. But....everything is async, right? That's not what async/await do.



Step 1: Create the object. Straightforward.

Step 2: Await `ReadLineAsync()`. That goes to a state machine

Step 3: Await `WriteLineAsync()`. That goes to a state machine....or does it?

```

using System.Runtime.CompilerServices;
using System.Threading.Tasks;

[AsyncStateMachine(typeof(<<Main>$>d__0))]
private static Task <Main>$(string[] args)
{
    <<Main>$>d__0 stateMachine = default(<<Main>$>d__0);
    stateMachine.<>t__builder = AsyncTaskMethodBuilder.Create();
    stateMachine.<>1__state = -1;
    stateMachine.<>t__builder.Start(ref stateMachine);
    return stateMachine.<>t__builder.Task;
}

```

That's what our program really looks like!

Let's focus on two things. The Start() call initiates a state machine defined by
 <<Main>\$>d__0

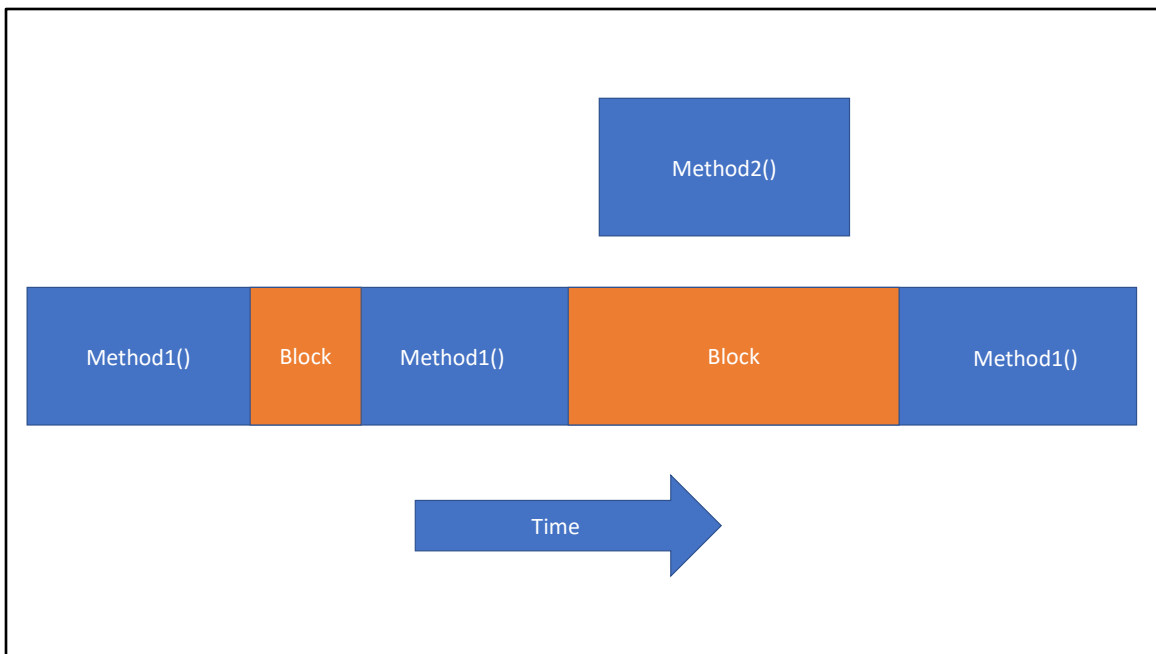
```

[CompilerGenerated]
[StructLayout(LayoutKind.Auto)]
private struct <<Main>>d__0 : IAsyncStateMachine
{

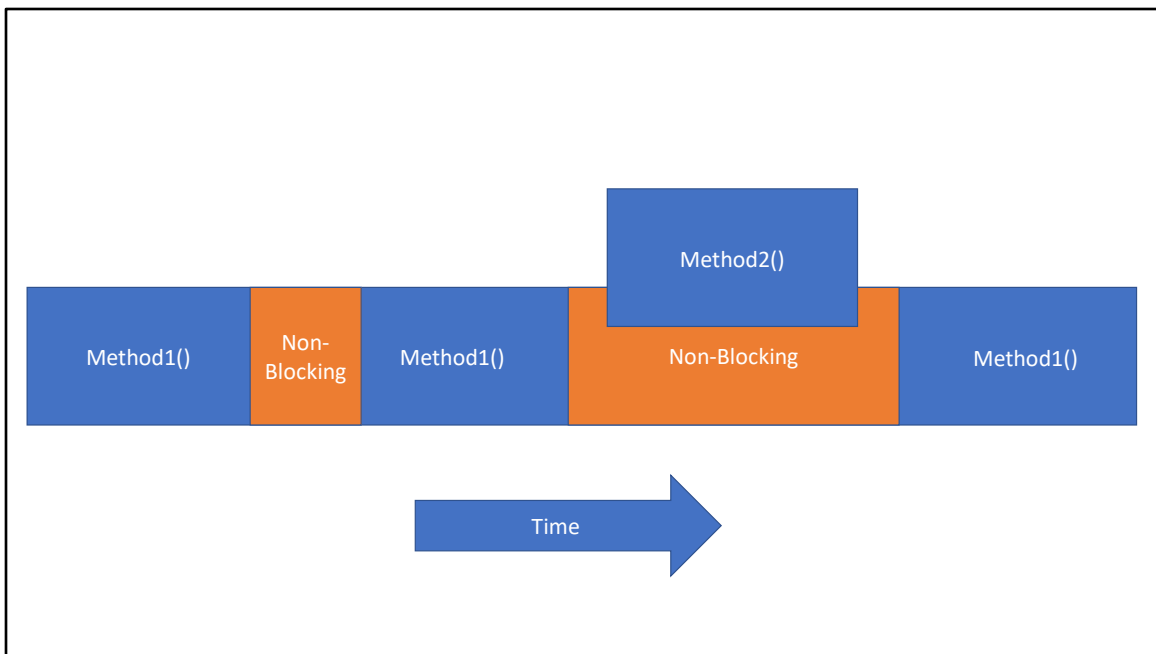
default:
{
    if (!<stream>5__2.EndOfStream)
    {
        awaiter2 =
            <stream>5__2.ReadLineAsync().ConfigureAwait(false).GetAwaiter();
        if (!awaiter2.IsCompleted)
        {

```

That's a class that implements a state machine. Even from a relatively simple piece of code, the state machine can get quite complex to follow. But notice that async invocations will always check the `IsCompleted` value of the returned `Task` first. Therefore, if your async method has a "fast path" to be complete right away....



The beauty of doing true async I/O work is that a thread can (potentially) share multiple workloads. If you block, you're forced to spin up more threads, which doesn't help with scalability.



The beauty of doing true async I/O work is that a thread can (potentially) share multiple workloads. If you block, you're forced to spin up more threads, which doesn't help with scalability.

```
[DebuggerTypeProxy(typeof(System.Threading.Tasks_TaskDebugView))]
[DebuggerDisplay("Id = {Id}, Status = {Status},
    Method = {DebuggerDisplayMethodDescription}")]
public class Task : IAsyncResult, IDisposable
```

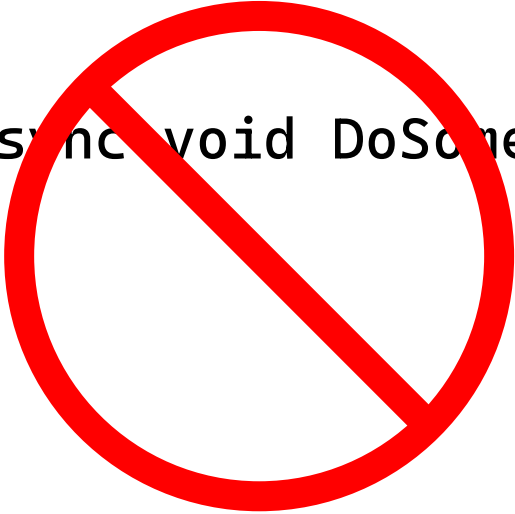
```
var code1 = Task.Run(() => { ... });
var code2 = Task.Run(() => { ... });
var code3 = Task.Run(() => { ... });

await Task.WhenAll(code1, code2, code3);
```

By the way, what IS a Task? It “represents an asynchronous operation”, though keep in mind it can also be used to spawn new Task instances via Task.Run

<https://source.dot.net/#System.Private.CoreLib/Task.cs>

```
public async void DoSomething()  
{  
    ...  
}
```



Now you can see why you should never do “async void”, because there’s nothing the compiler can do with an async method that doesn’t return a task. How do you “continue with” the next piece of code? The **ONLY** time this is acceptable is with event handlers in XAML apps.

```
[MethodImpl(MethodImplOptions.Synchronized)]
public override Task WriteLineAsync(string? value)
{
    WriteLine(value);
    return Task.CompletedTask;
}
```

You should give a fair amount of consideration to introducing asynchronous operations in base types. If the type doesn't want to enforce it, it can return `Task.FromResult` or `Task.CompletedTask`, but this will probably still end up with an async state machine compiled in code.

<https://source.dot.net/#System.Private.CoreLib/TextWriter.cs>

```
[Synchronous]
public override Task<int> DoSomething()
{
    return Task.FromResult(22);
}

...

var result = x.DoSomething().Result;
var resultAwait = await x.DoSomething() // wrong!
```

Side note: adding `MethodImplOptions.Synchronized` doesn't mean the method will not be called without the async machine. Someday I'll create this attribute with the analyzer and make sure people don't call it with `await`; they can simply invoke it and potentially call `.Result` on it without blocking issues.

Demo: Evolving Expressions

Concurrent Programming in .NET

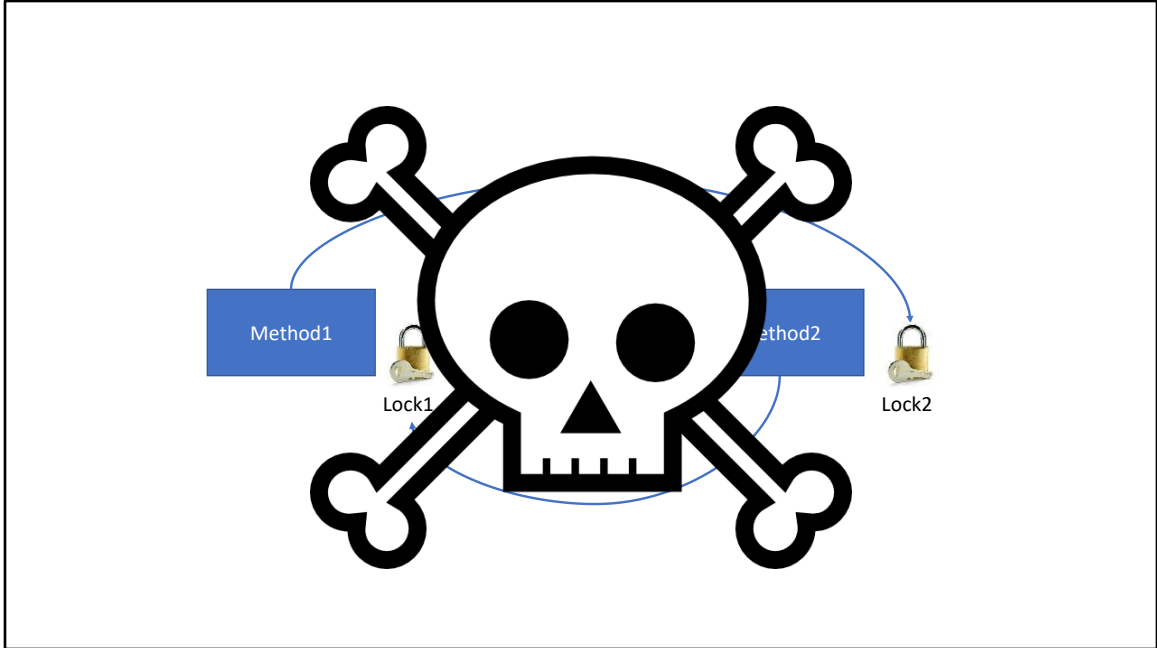
Let's look at my ExpressionEvolver application and how I use Tasks to do things concurrently.



First off, avoid locks if you can. Locks are expensive, and may be a sign that you haven't isolated key functional parts of your application. It isn't something that you can completely eliminate, but if your code is littered with locking strategies, you've got issues.

May want to talk about experiences with locking in applications, like, "I should be the only person able to make changes on this loan."

http://www.openseesay.com/wp-content/uploads/2014/07/lock_and_key.jpg



It's also very easy to create deadlocks. If two pieces of code acquire a lock and then try to lock on those acquired locks, your code is dead. There are ways to mitigate that, but whenever you do locks, you have to be aware of this.

<http://www.clker.com/cliparts/4/K/F/C/a/c/skull-and-crossbones-hi.png>

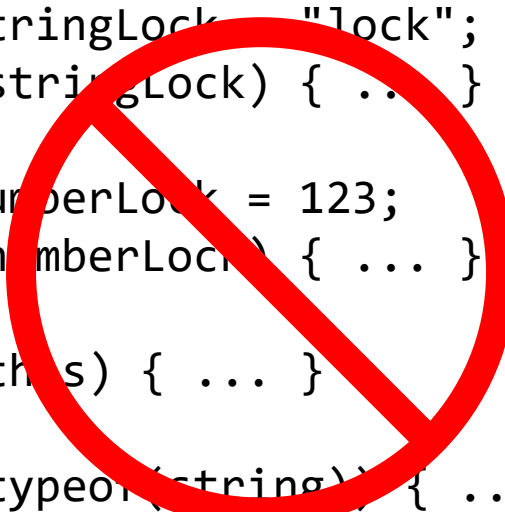

```
public sealed class MonitorLedger
    : ILedger
{
    private readonly object @lock = new object();

    public Ledger(decimal value) => this.Value = value;

    public void Credit(decimal value)
    {
        lock (this.@lock)
        {
            this.Value -= value;
        }
    }
}
```

Here's the simplest example of using a lock

```
var stringLock = "lock";  
lock(stringLock) { ... }  
  
var numberLock = 123;  
lock(numberLock) { ... }  
  
lock(this) { ... }  
  
lock(typeof(string)) { ... }
```



Note that the thing you're locking on matters. Don't do these!

A string lock can accidentally be shared (strings that are the same exact value are shared)

A lock based on a value type are boxed and end up having two different objects being locked on

A "this" lock is not private because others can lock on that object

A "typeof" lock can accidentally be shared if the type is public

These can all lead to unexpected behaviors, potentially deadlocks.

<https://docs.microsoft.com/en-us/dotnet/fundamentals/code-analysis/quality-rules/ca2002>

```
public sealed class IntegerFieldLedger
{
    private int value;

    public IntegerFieldLedger(int value) => this.value = value;

    public void Credit(int value) =>
        Interlocked.Add(ref this.value, value);

    public void Debit(int value) =>
        Interlocked.Add(ref this.value, value * -1);

    public int Value
    {
        get => this.value;
    }
}
```

Depending on the data type you use, you can also use `Interlocked`, which is much faster than doing locking via `Monitor`. However, again, you're limited on specific data types for this to work.

```
C:\WINDOWS\system32\cmd.exe

Method | Mean | Error | StdDev | Allocated
-----|-----|-----|-----|-----
CreditAndDebitMonitorLedger | 47.75 ns | 0.8496 ns | 0.7947 ns | 0 B
CreditAndDebitInterlockedLedger | 16.91 ns | 0.1267 ns | 0.1123 ns | 0 B

// * Hints *
Outliers
InterlockedPerformance.CreditAndDebitInterlockedLedger: Default -> 1 outlier
was removed
```

As you can see, it is faster, but it has its limitations.

```
public sealed class SpinLockLedger : ILedger
{
    private SpinLock @lock = new SpinLock();

    public void Credit(decimal value)
    {
        var isLockAcquired = false;

        try
        {
            this.@lock.Enter(ref isLockAcquired);
            this.Value -= value;
        }
        finally
        {
            if (isLockAcquired)
            {
                this.@lock.Exit();
            }
        }
    }
}
```

You can use another technique to do locking, which is to use a SpinLock

```
C:\WINDOWS\system32\cmd.exe

Method | Mean | Error | StdDev | Allocated |
-----|-----|-----|-----|-----|
CreditAndDebitMonitorLedger | 81.98 ns | 0.6623 ns | 0.5871 ns | 0 B |
CreditAndDebitSpinLockLedger | 176.03 ns | 3.4909 ns | 4.4148 ns | 0 B |

// * Hints *
Outliers
SpinLockPerformance.CreditAndDebitMonitorLedger: Default -> 1 outlier was removed
SpinLockPerformance.CreditAndDebitSpinLockLedger: Default -> 1 outlier was removed
```

But take care with performance. Make sure it's worth it.

- In general, while holding a spin lock, one should avoid any of these actions:
 - Blocking
 - Calling anything that itself may block
 - Holding more than one spin lock at once
 - Making dynamically dispatched calls (interface and virtuals)
 - Making statically dispatched calls into any code one doesn't own
 - Allocating memory

Also, read the docs and don't use a SpinLock if you're in these scenarios

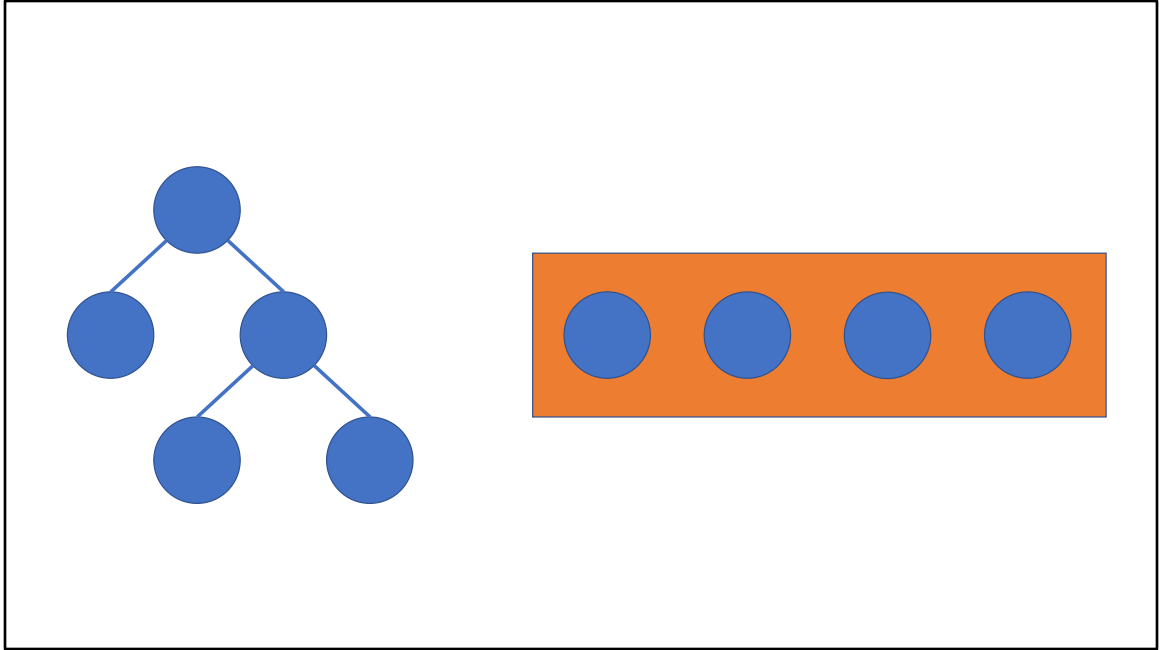
And don't store SpinLock instances in readonly fields.

<https://docs.microsoft.com/en-us/dotnet/api/system.threading.spinlock?view=netframework-4.7>

Demo: Benchmarking Code

Concurrent Programming in .NET

Let's look at how you can use Benchmark to diagnose perf issues.



Another aspect to keep in mind is the data structures you're using. In recent times, new data structures have been released that can help in certain concurrent scenarios.

```
var guidStack = new Stack<Guid>();

for (var i = 0; i < 100; i++)
{
    guidStack.Push(Guid.NewGuid());
}

Console.WriteLine(
    $"{nameof(guidStack)}.Pop() is {guidStack.Pop()}");
```

For example, take a look at this simple code to create a stack.

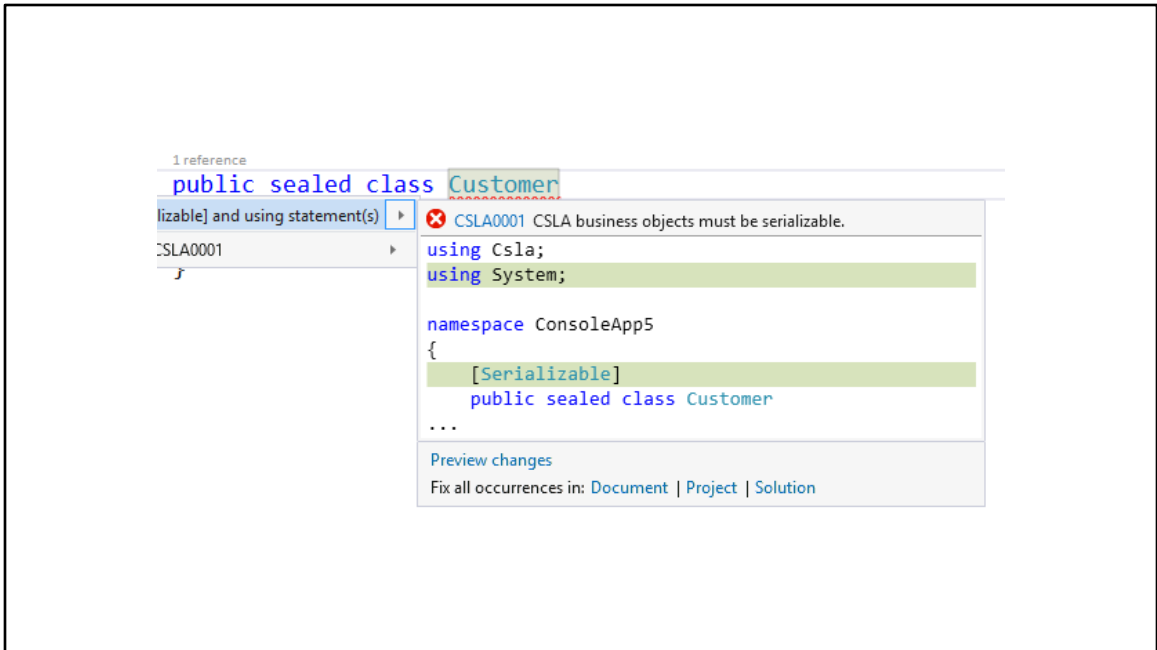
```
var guidsInImmutableStack = ImmutableStack<Guid>.Empty;

for (var i = 0; i < 100; i++)
{
    guidsInImmutableStack =
        guidsInImmutableStack.Push(Guid.NewGuid());
}

guidsInImmutableStack = guidsInImmutableStack.Pop(
    out var immutableResult);

Console.Out.WriteLine(
    $"{nameof(guidsInImmutableStack)}.Pop() is
    {immutableResult}");
```

You can also make immutable stacks (think Roslyn syntax trees). Remember to capture return values! Note that the `.Pop()` SHOULD be captured as well, though in this case we don't care.



And why are immutable structures used by the Compiler API? Many reasons, but this one illustrates it very well. A code fix doesn't change the code in the code editor; it's a copy and it'll only be committed if the developer accepts it.



The best code is the code you don't write. Nothing can be truer with concurrent programming. If you don't have to do it and you can be lazy, go for it!



Orleans

Actor frameworks are great because all of the communication with and between actors are asynchronous, but the message processing itself is synchronous (unless the handler itself is async). There are actor frameworks for .NET, like Orleans and Akka.

<https://dotnet.github.io/orleans/>

Demo: Using Orleans

Concurrent Programming in .NET

Let's look at how one can use Orleans to handle messages

Concurrent Programming in .NET

Jason Bock

Remember...

- <https://github.com/JasonBock/ConcurrentProgramming>
- <https://github.com/JasonBock/ExpressionEvolver>
- <https://github.com/JasonBock/Presentations>
- References in the notes on this slide

References

- * [Async Guidance](<https://github.com/davidfowl/AspNetCoreDiagnosticScenarios/blob/master/AsyncGuidance.md>)
- * [Eric Lippert Async Articles](<https://ericlippert.com/category/asynchrony/>)
- * [Stephen Cleary blog](<https://blog.stephencleary.com/>)
- * [Concurrency Visualizer for Visual Studio 2019](<https://marketplace.visualstudio.com/items?itemName=Diagnostics.DiagnosticsConcurrencyVisualizer2019>)
- * [Async ValueTask Pooling in .NET 5](<https://devblogs.microsoft.com/dotnet/async-valuetask-pooling-in-net-5/>)
- * [ConfigureAwait FAQ](<https://devblogs.microsoft.com/dotnet/configureawait-faq/>)
- * [An Introduction to System.Threading.Channels](<https://www.stevejgordon.co.uk/an-introduction-to-system-threading-channels>)
- * [PooledAwait](<https://github.com/mgravell/PooledAwait>)
- * [await the async Letdown](<https://odetocode.com/blogs/scott/archive/2019/03/04/await-the-async->

letdown.aspx)

- * [C# Async Antipatterns](https://markheath.net/post/async-antipatterns)
- * [The Ultimate Guide to Asynchronous Programming in C# and ASP.NET](https://exceptionnotfound.net/async-await-in-asp-net-csharp-ultimate-guide/)
- * [A Practical Example Of Asynchronous Programming in C# and ASP.NET](https://exceptionnotfound.net/asynchronous-programming-asp-net-csharp-practical-guide-refactoring/)
- * [IEnumerable, Writing Operators](https://github.com/akarnokd/async-enumerable-dotnet/wiki/Writing-operators)
- * [Understanding the Whys, Whats, and Whens of ValueTask](https://devblogs.microsoft.com/dotnet/understanding-the-whys-whats-and-whens-of-valuetask/)
- * [The danger of TaskCompletionSource<T> class](https://devblogs.microsoft.com/premier-developer/the-danger-of-taskcompletingsourcet-class/)
- * [I didn't understand why people struggled with (.NET's) async](https://www.productiverage.com/i-didnt-understand-why-people-struggled-with-nets-async)
- * [Asynchronous Programming in .NET](https://rubikscore.net/2018/08/06/asynchronous-programming-in-net/)
- * [Async in depth](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/)
- * Asynchronous Programming in .NET
 - * [Motivation and Unit Testing](https://rubikscore.net/2018/05/21/asynchronous-programming-in-net-motivation-and-unit-testing/)
 - * [Common Mistakes and Best Practices](https://rubikscore.net/2018/05/28/asynchronous-programming-in-net-common-mistakes-and-best-practices/)
 - * [Task-based Asynchronous Pattern (TAP)](https://rubikscore.net/2018/06/04/asynchronous-programming-in-net-task-based-asynchronous-pattern-tap/)
 - * [Benefits and Tradeoffs of Using ValueTask](https://rubikscore.net/2018/06/11/asynchronous-programming-in-net-benefits-and-tradeoffs-of-using-valuetask/)
- * [15 Years of Concurrency](http://joeduffyblog.com/2016/11/30/15-years-of-concurrency/)
- * [Constraining Concurrent Threads in C#](https://markheath.net/post/constraining-concurrent-threads-csharp)
- * [Dissecting the async methods in C#](https://devblogs.microsoft.com/premier-developer/dissecting-the-async-methods-in-c/)
- * [Async/Await](Best Practices in Asynchronous Programming -

<https://msdn.microsoft.com/en-us/magazine/jj991977.aspx>)

- * [Asynchronous Programming Patterns](<https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/>)

- * [SpinLock](<https://docs.microsoft.com/en-us/dotnet/standard/threading/spinlock>)

- * [Task-based Asynchronous Pattern (TAP)](<https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap>)

- * [Do I need to dispose of Tasks?](<https://devblogs.microsoft.com/pfxteam/do-i-need-to-dispose-of-tasks/>)

- * [CallContext in .NET Core](<http://www.cazzulino.com/callcontext-netstandard-netcore.html>)

- * Async from the Outside, From the inside

- * [Part 1](<https://www.infoq.com/presentations/C-Sharp-Async-From-the-Outside-From-the-Inside>)

- * [Part 2](<https://www.infoq.com/presentations/C-Sharp-Async-From-the-Outside-From-the-Inside-2>)

- * [The curious case of async, await, and IDisposable](<http://thebillwagner.com/Blog/Item/2017-05-03-ThecuriouscaseofasyncawaitandIDisposable>)

- * [Maximizing Throughput - The Overhead of 1 Million Tasks](<https://www.danielcrabtree.com/blog/248/maximizing-throughput-the-overhead-of-1-million-tasks>)

- * [The CLR Thread Pool 'Thread Injection' Algorithm](<http://mattwarren.org/2017/04/13/The-CLR-Thread-Pool-Thread-Injection-Algorithm/>)

- * Common Multithreading Mistakes in C#

- * [Part 1](https://benbowen.blog/post/cmmics_i/)

- * [Part 2](https://benbowen.blog/post/cmmics_ii/)

- * [Part 3](https://benbowen.blog/post/cmmics_iii/)

- * [Part 4](https://benbowen.blog/post/cmmics_iv/)

- * [Job Objects](<https://docs.microsoft.com/en-us/windows/win32/procthread/job-objects>)