Diagnosing async C# code

Sergey Teplyakov

Principal software engineer

Microsoft

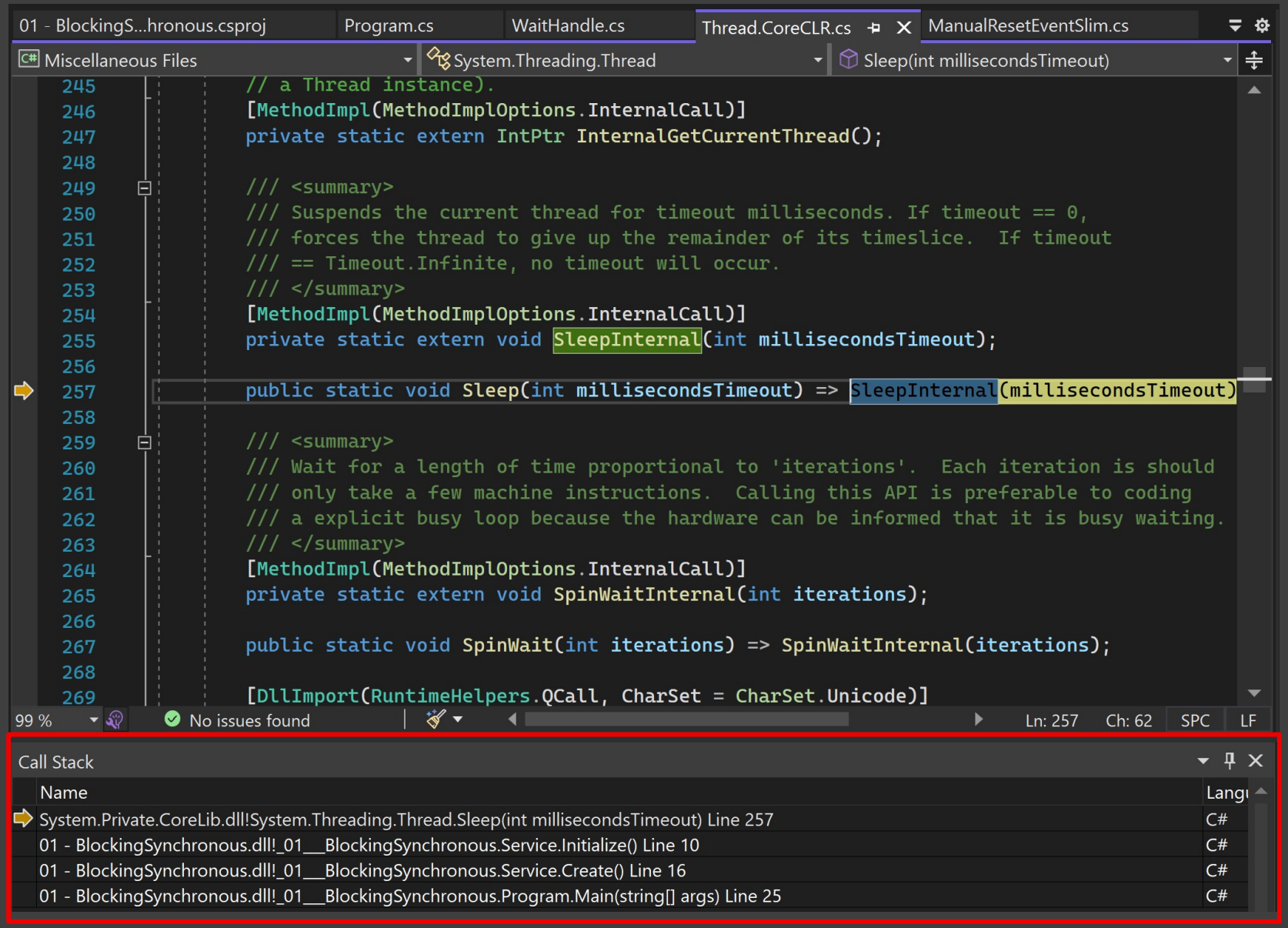
Debugging is easy! Right?

Stack traces for the rescue!

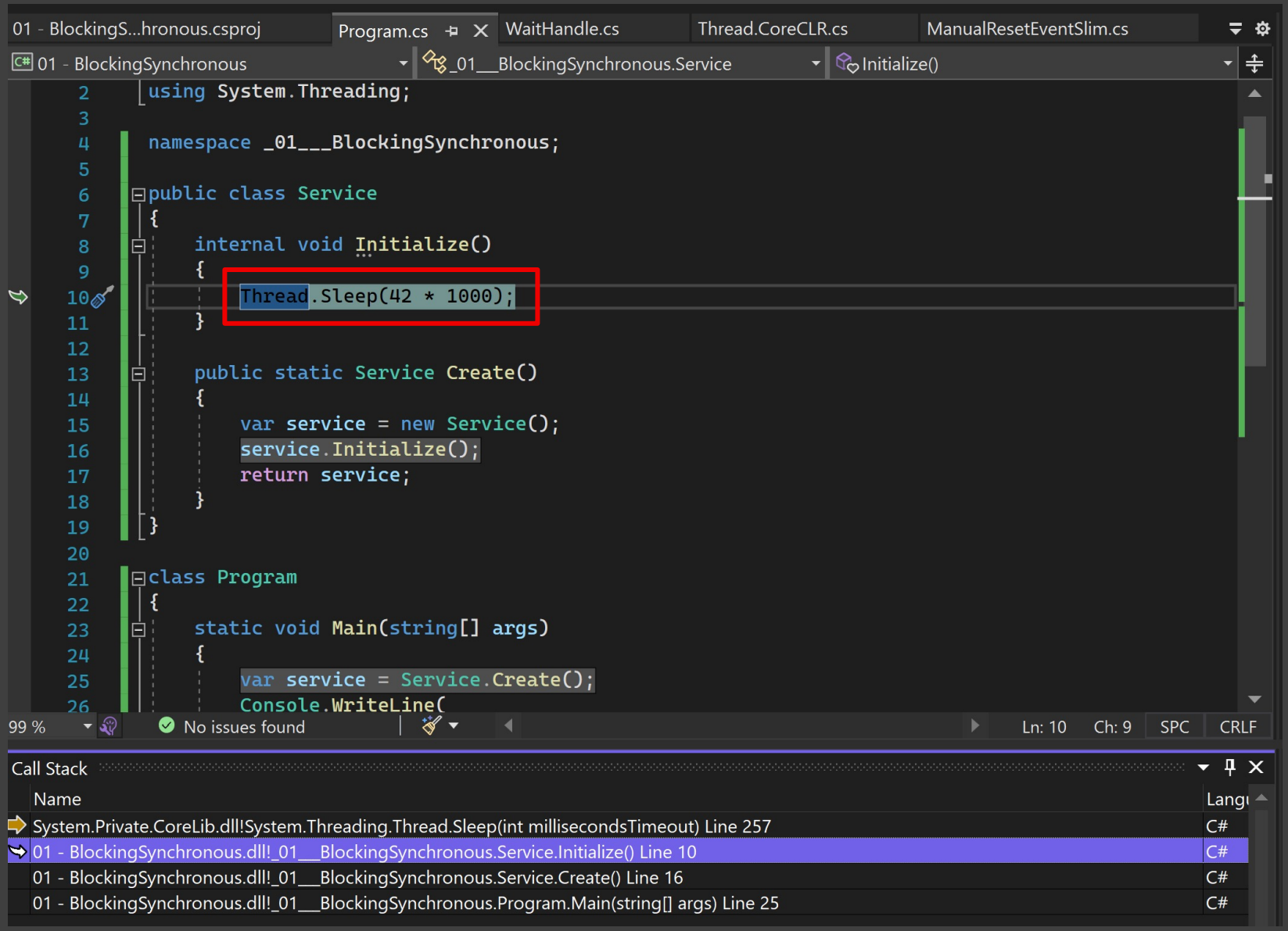
Diagnosing blocking synchronous call

```
public class Service
{
    internal void Initialize()
    {
        Thread.Sleep(42 * 1000);
    }

    public static Service Create()
    {
        var service = new Service();
        service.Initialize();
        return service;
    }
}

class Program
{
    static void Main(string[] args)
    {
        var service = Service.Create();
        Console.WriteLine(
            $"Service {service} is constructed.");
    }
}
```

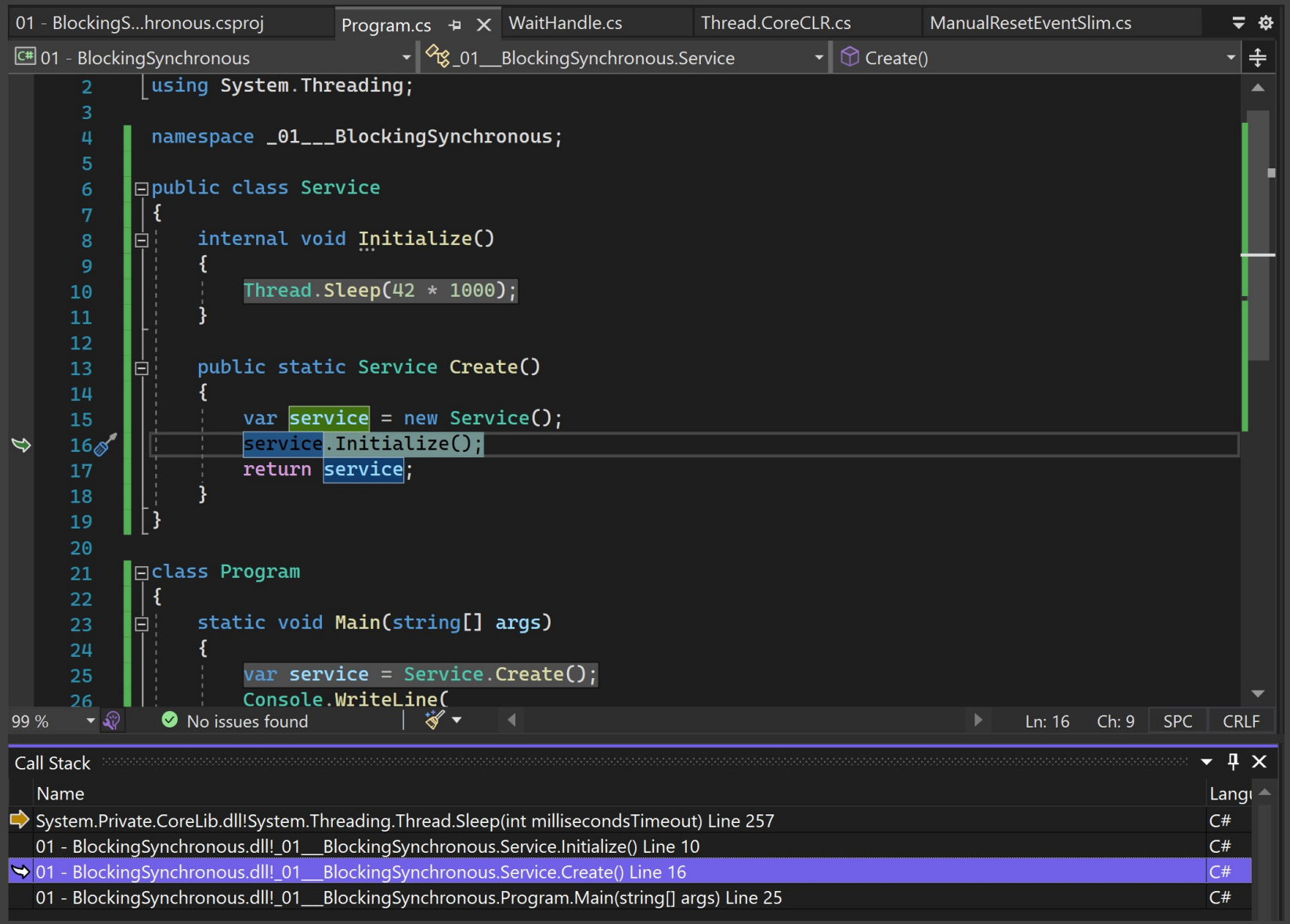
Diagnosing blocking synchronous call



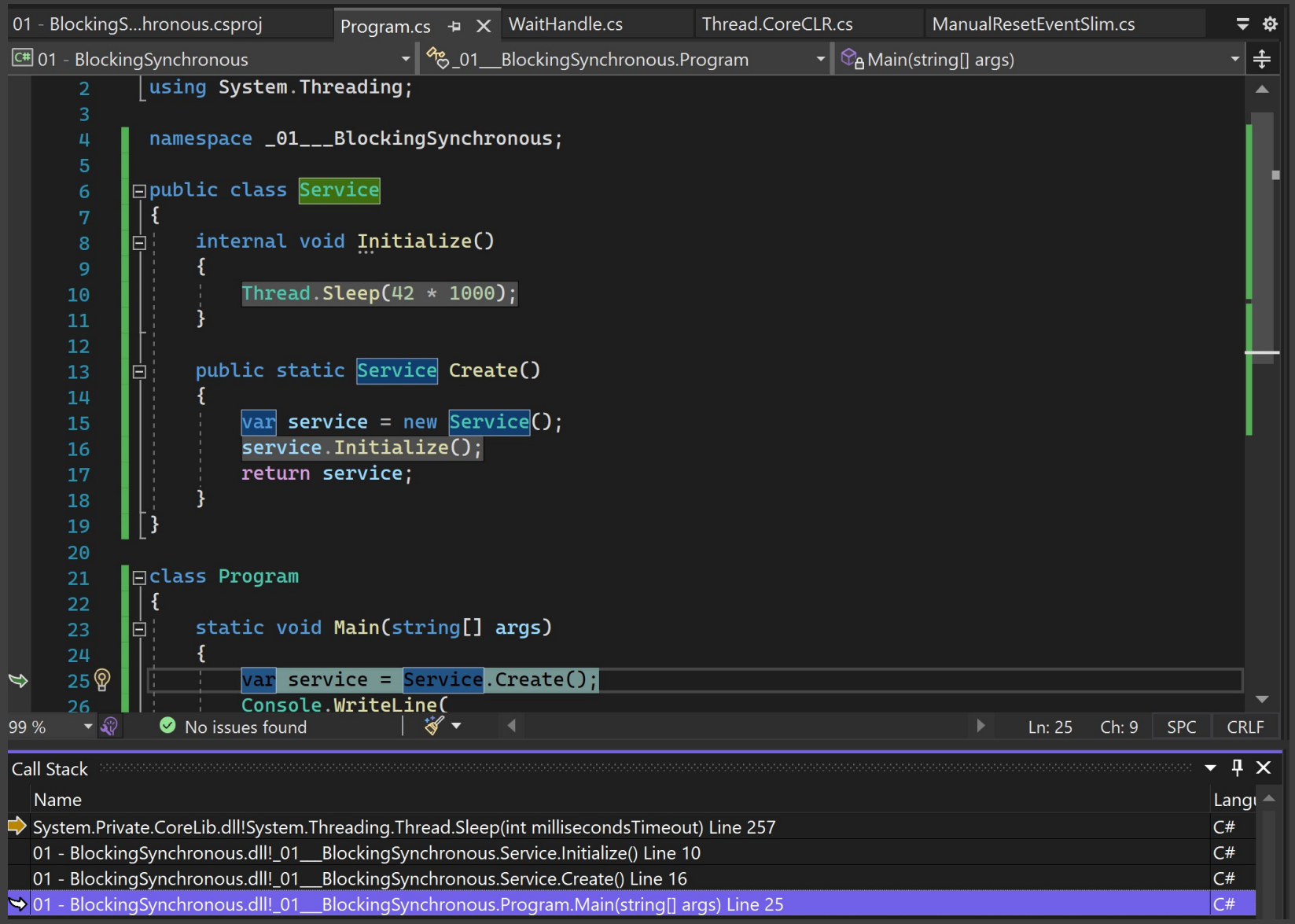
Diagnosing blocking synchronous call

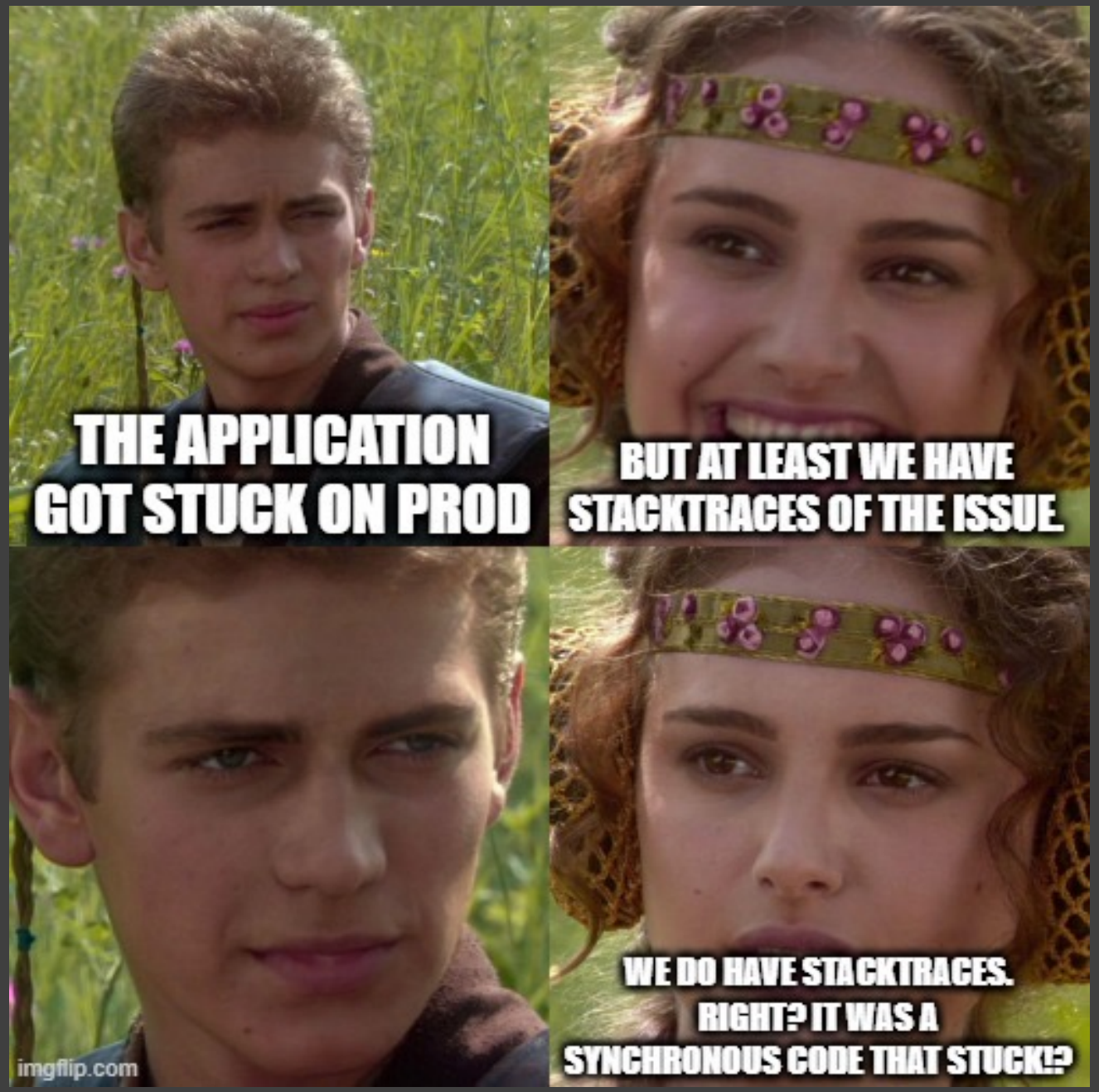


Diagnosing blocking synchronous call



Diagnosing blocking synchronous call




THE APPLICATION GOT STUCK ON PROD
BUT AT LEAST WE HAVE STACKTRACES OF THE ISSUE.
WE DO HAVE STACKTRACES. RIGHT? IT WAS A SYNCHRONOUS CODE THAT STUCK!?
imgflip.com

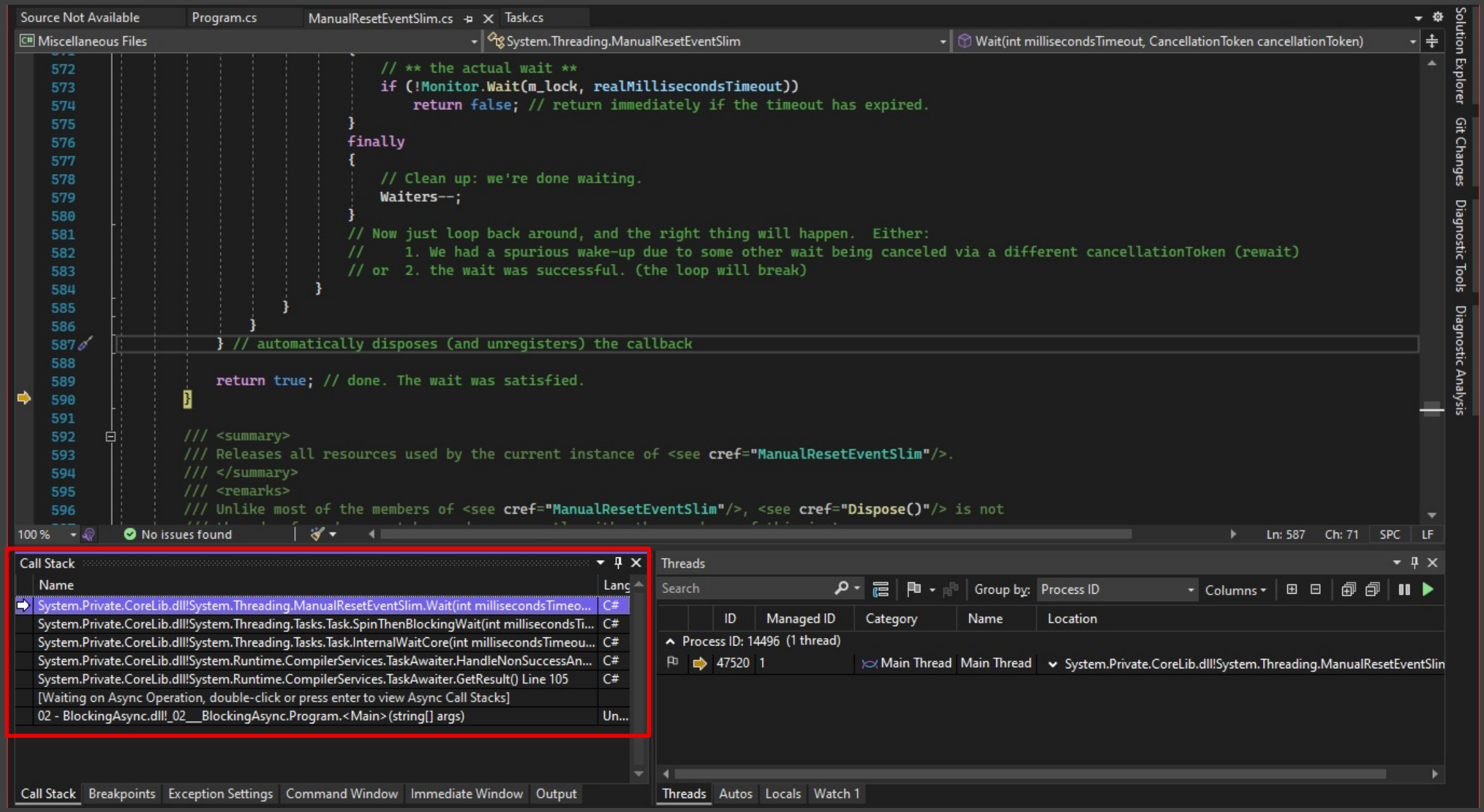
Let's look at a very simple async case

```
public class Service
{
    internal void Initialize()
    {
        Thread.Sleep(42 * 1000);
    }

    public static Service Create()
    {
        var service = new Service();
        service.Initialize();
        return service;
    }
}

class Program
{
    static void Main(string[] args)
    {
        var service = Service.Create();
        Console.WriteLine(
            $"Service {service} is constructed.");
    }
}
```

Sync 2 async

```
public class Service
{
    internal async Task InitializeAsync()
    {
        await Task.Delay(42 * 1000);
    }

    public static async Task<Service> CreateAsync()
    {
        var service = new Service();
        await service.InitializeAsync();
        return service;
    }
}

class Program
{
    static async Task Main(string[] args)
    {
        var service = await Service.CreateAsync();
        Console.WriteLine(
            $"Service {service} is constructed.");
    }
}
```

Debugging async code



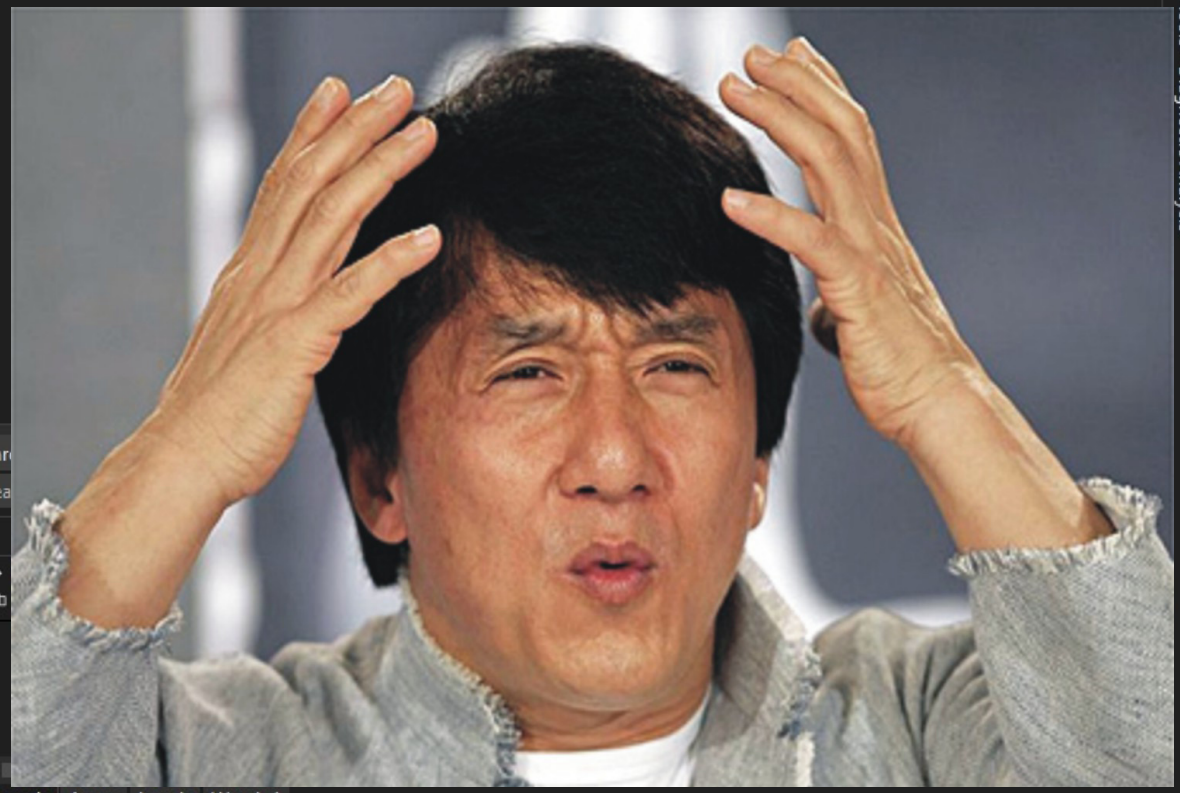
Where is our code?!?!?

Source Not Available | Program.cs | ManualResetEventSlim.cs | Task.cs

Source Not Available

Source information is missing from the debug information for this module

You can view disassembly in the Disassembly window. To always view disassembly for missing source files, change the setting in the Options dialog.



Call Stack

Name	Lang
System.Private.CoreLib.dll!System.Threading.ManualResetEventSlim.Wait(int millisecondsTimeo...	C#
System.Private.CoreLib.dll!System.Threading.Tasks.Task.SpinThenBlockingWait(int millisecondsTi...	C#
System.Private.CoreLib.dll!System.Threading.Tasks.Task.InternalWaitCore(int millisecondsTimeou...	C#
System.Private.CoreLib.dll!System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAn...	C#
System.Private.CoreLib.dll!System.Runtime.CompilerServices.TaskAwaiter.GetResult() Line 105	C#
[Waiting on Async Operation, double-click or press enter to view Async Call Stacks]	
02 - BlockingAsync.dll!_02__BlockingAsync.Program.<Main>(string[] args)	Un...

Call Stack | Breakpoints | Exception Settings | Command Window | Immediate Window | Output

Threads | Autos | Locals | Watch 1

Solution Explorer | Git Changes | Diagnostic Tools | Diagnostic Analysis

Async Main: the new feature in C# 7

- Async entry points are very common
- Starting with C# 7.1 the main method can be asynchronous

Async main

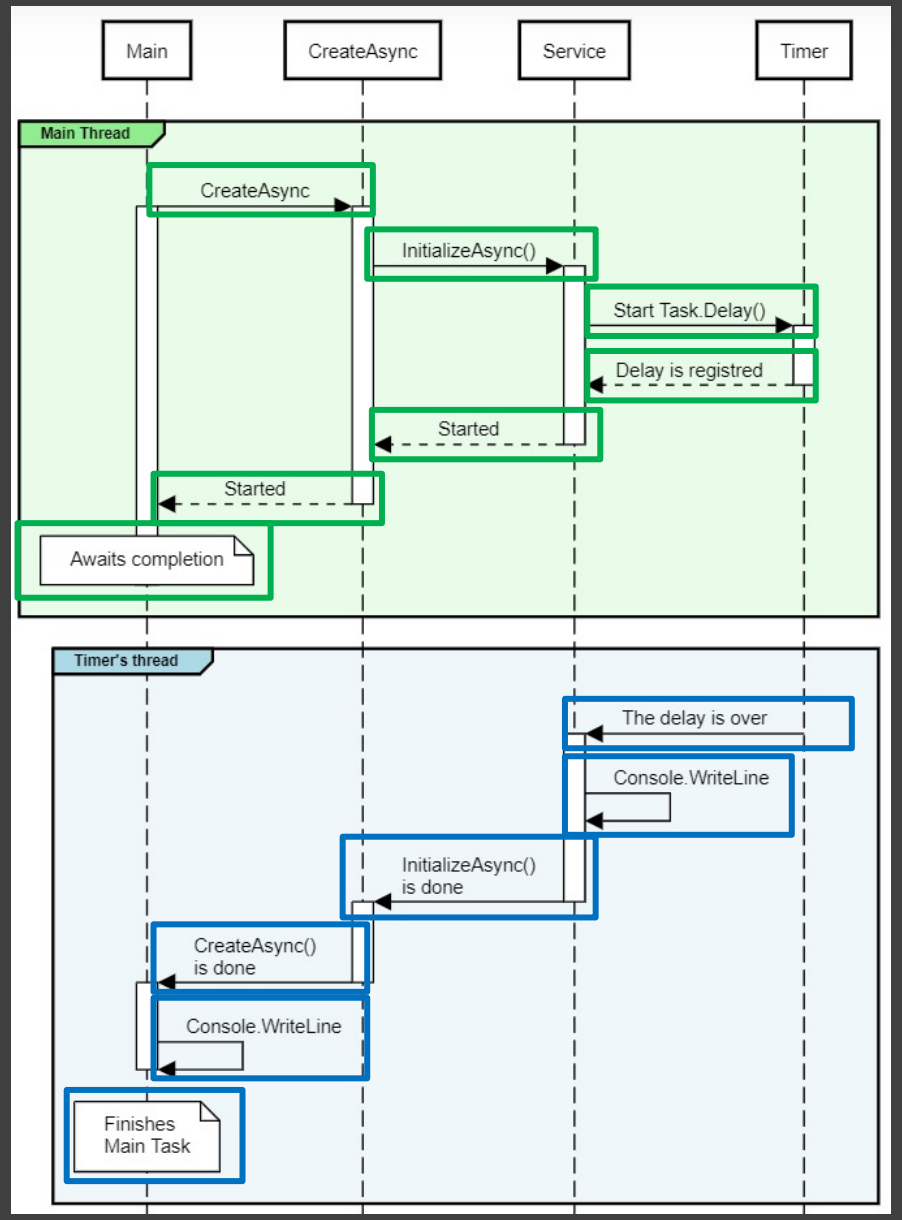
```
class Program
{
    static async Task Main(string[] args)
    {
        var service = await Service.CreateAsync();
        Console.WriteLine($"Service {service} is constructed.");
    }
}
```

Translated code

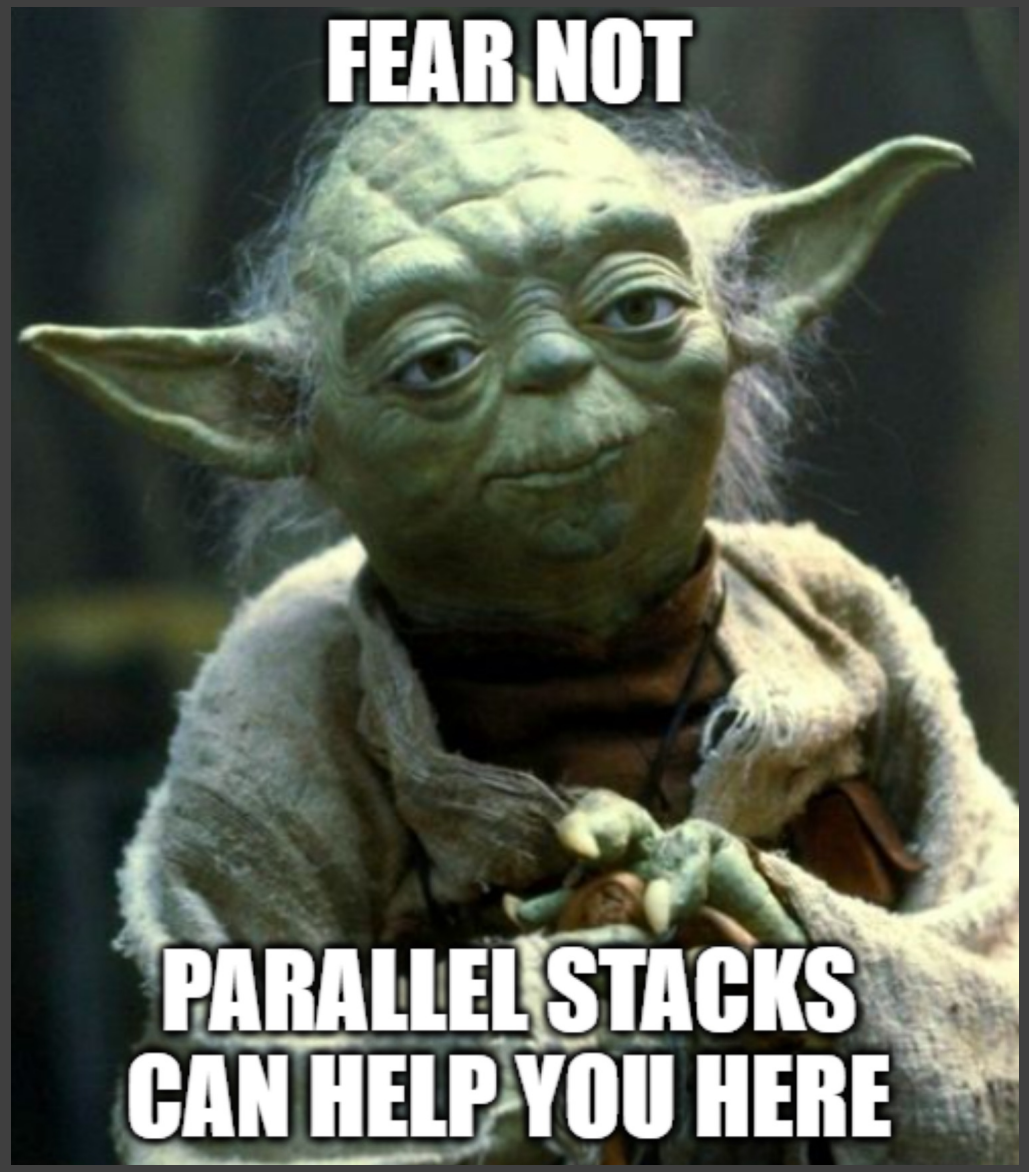
```
class Program
{
    static void Main(string[] args)
    {
        MainAsync(args).GetAwaiter().GetResult();
    }

    static async Task MainAsync(string[] args)
    {
        var service = await Service.CreateAsync();
        Console.WriteLine($"Service {service} is constructed.");
    }
}
```

TLDR; the async flow: TODO: fix

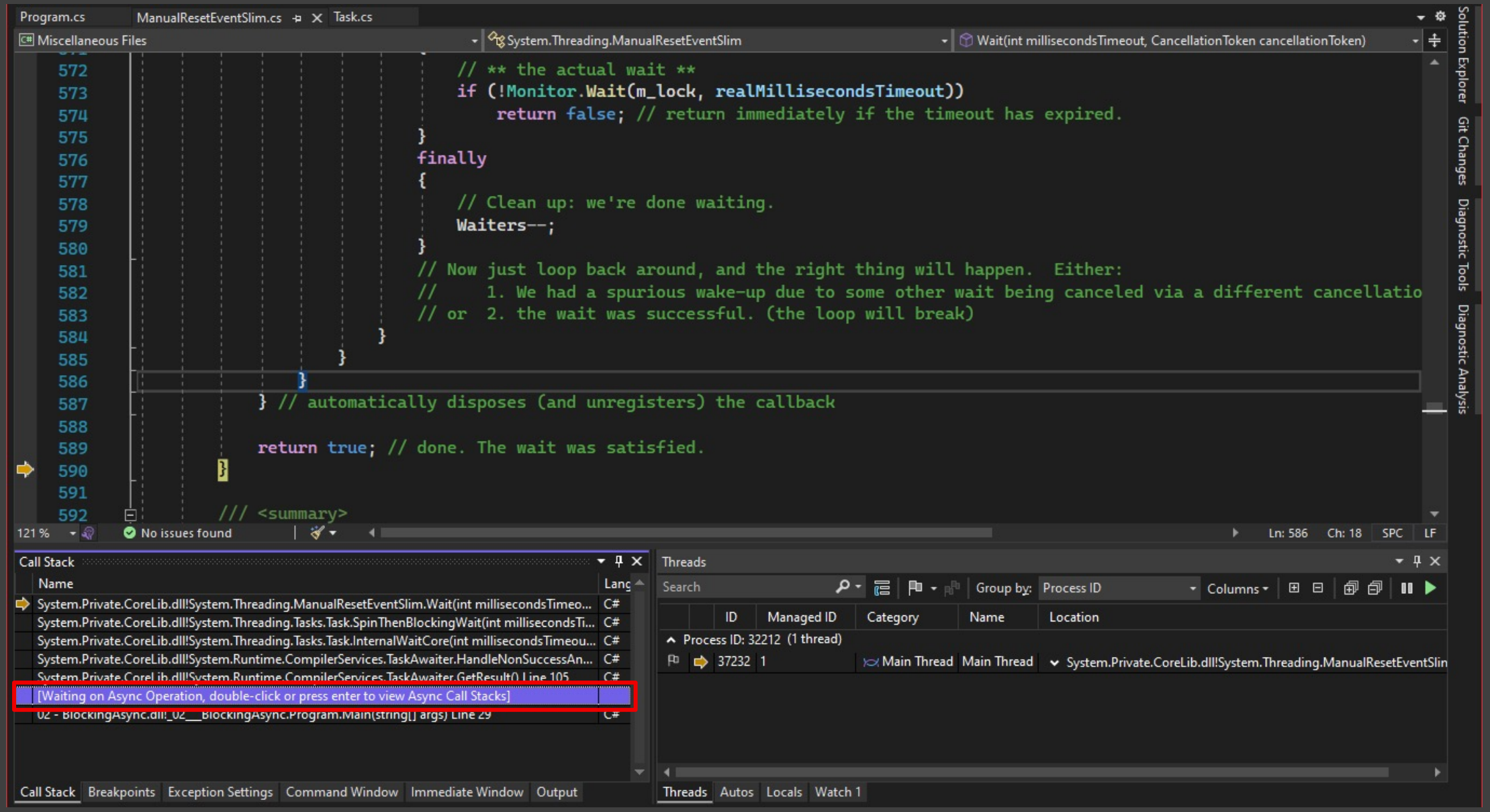
```
public class Service
{
    internal async Task InitializeAsync()
    {
        var delayTask = Task.Delay(42 * 1000);
        await delayTask;
        Console.WriteLine("InitializeAsync is done.");
    }

    public static async Task<Service> CreateAsync()
    {
        var service = new Service();
        var initializeTask = service.InitializeAsync();
        await initializeTask;
        return service;
    }
}

class Program
{
    static async Task Main(string[] args)
    {
        var createServiceTask = Service.CreateAsync();
        var service = await createServiceTask;
        Console.WriteLine($"Service {service} is constructed.");
    }
}
```



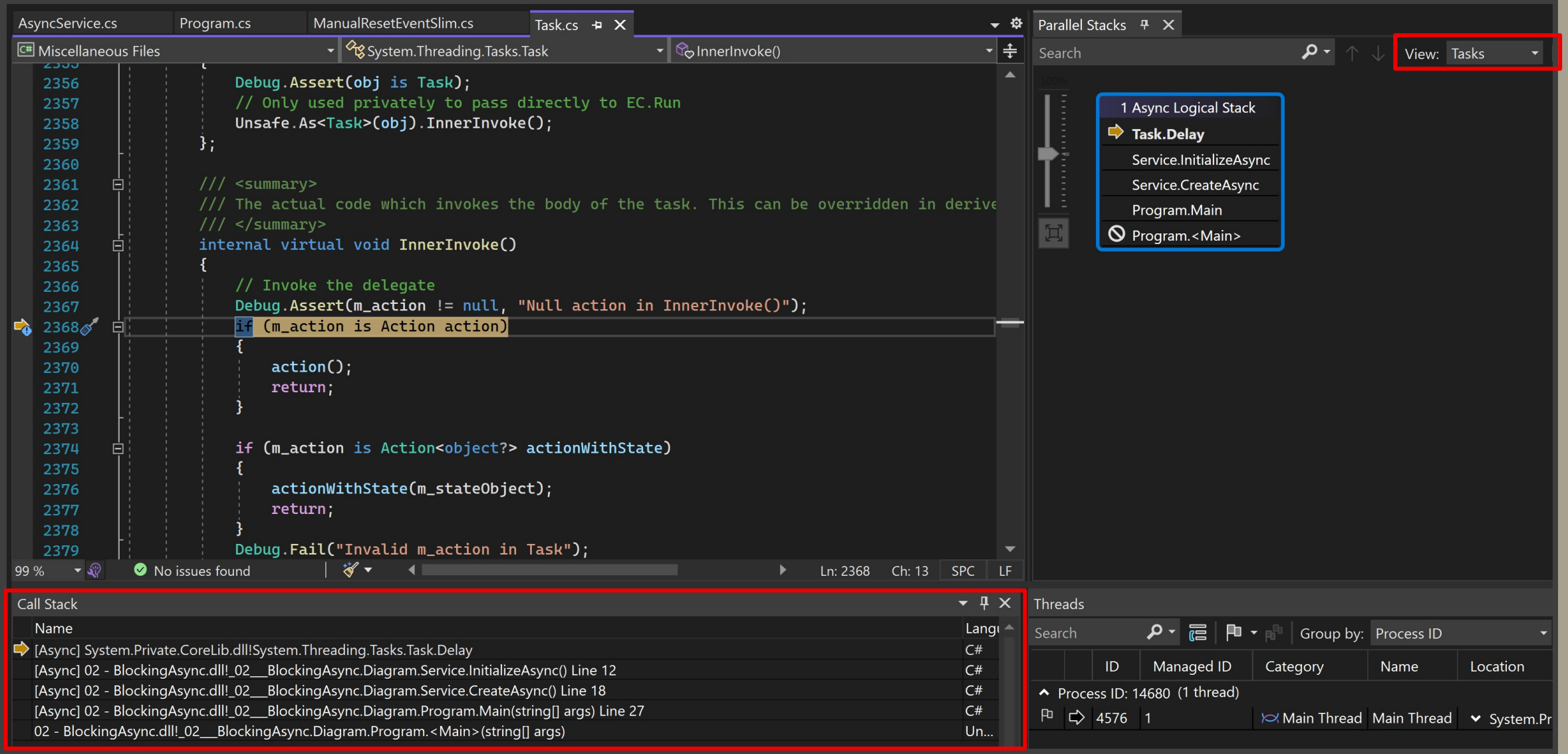
And how we should debug it?



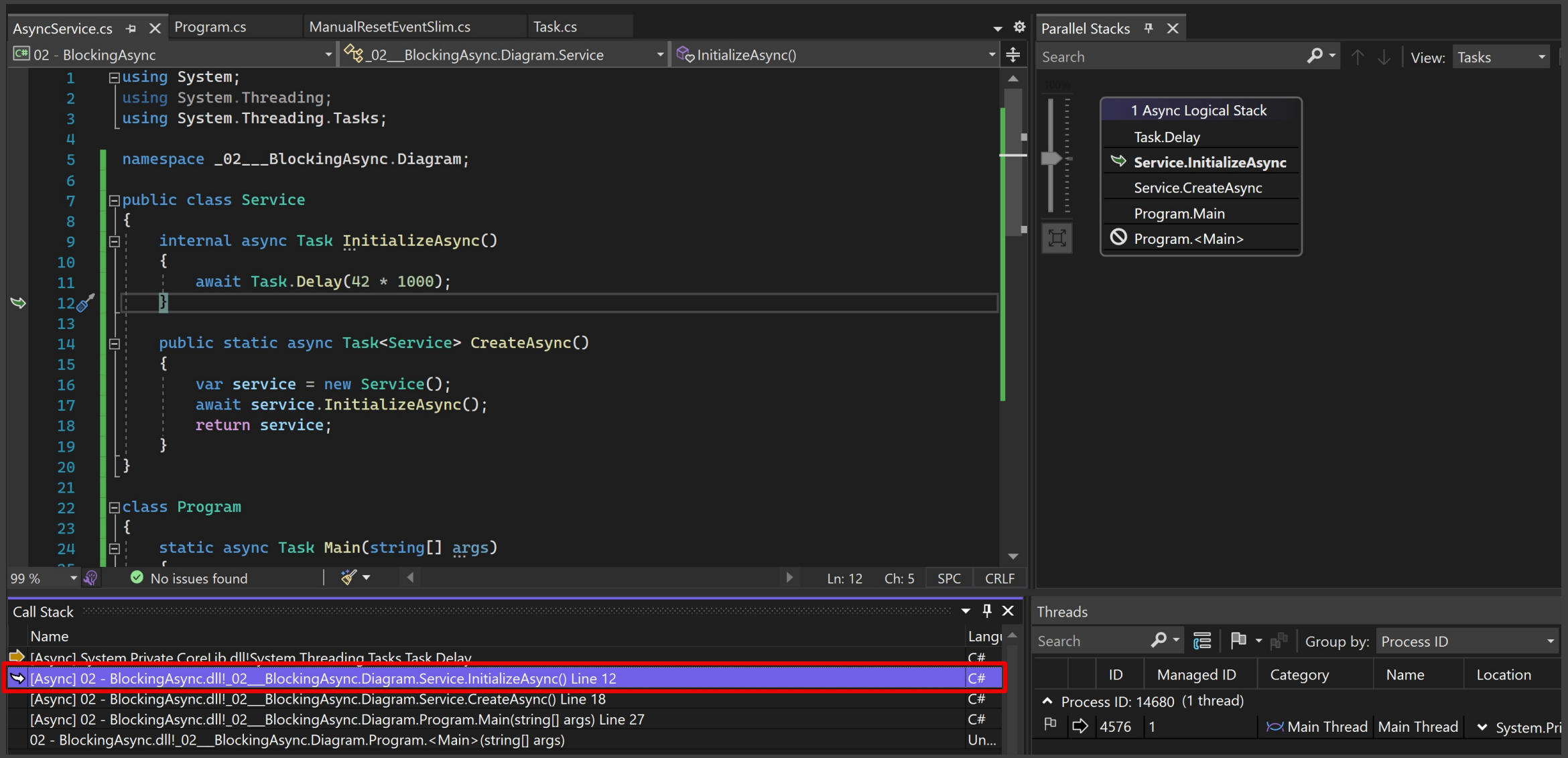
“Call Stacks” to “Async Call Stacks”



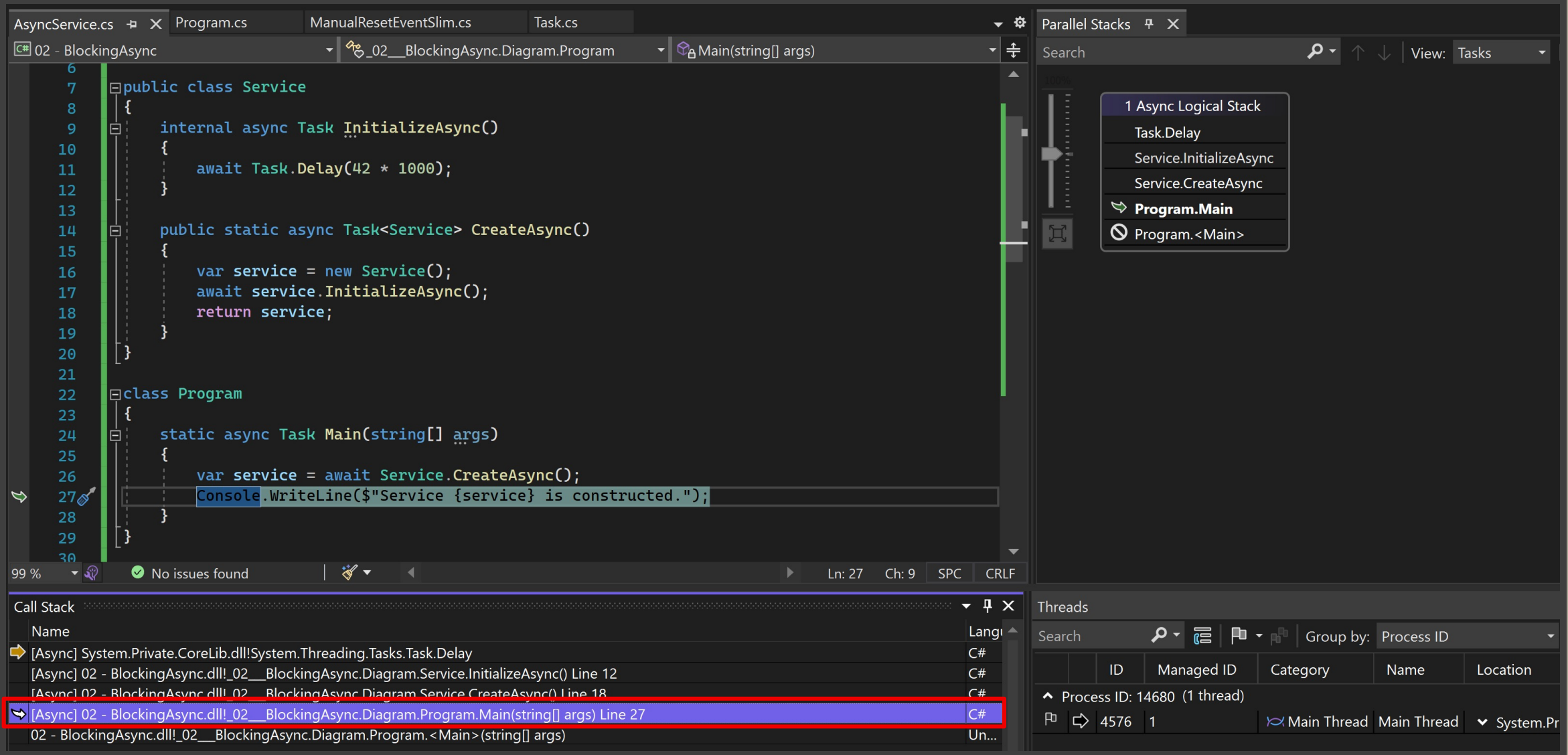
Async call stack



Async call stack



Async call stack



Sync vs. Async Call stacks

Sync version

```
6   public class Service
7   {
8       internal void Initialize()
9       {
10          Thread.Sleep(42 * 1000);
11      }
12
13      public static Service Create()
14      {
15          var service = new Service();
16          service.Initialize();
17          return service;
18      }
19  }
20
21  class Program
22  {
23      static void Main(string[] args)
24      {
25          var service = Service.Create();
26          Console.WriteLine($"Service {service} is constructed.");
27      }
28  }
29
```

99 % No issues found

Call Stack

Name
System.Private.CoreLib.dll!System.Threading.Thread.Sleep(int millisecondsTimeout) Line 257
01 - BlockingSynchronous.dll!_01__BlockingSynchronous.Service.Initialize() Line 10
01 - BlockingSynchronous.dll!_01__BlockingSynchronous.Service.Create() Line 16
01 - BlockingSynchronous.dll!_01__BlockingSynchronous.Program.Main(string[] args) Line 25

Async version

```
7   public class Service
8   {
9       internal async Task InitializeAsync()
10      {
11          await Task.Delay(42 * 1000);
12      }
13
14      public static async Task<Service> CreateAsync()
15      {
16          var service = new Service();
17          await service.InitializeAsync();
18          return service;
19      }
20  }
21
22  class Program
23  {
24      static async Task Main(string[] args)
25      {
26          var service = await Service.CreateAsync();
27          Console.WriteLine($"Service {service} is constructed.");
28      }
29  }
```

99 % No issues found

Call Stack

Name
[Async] System.Private.CoreLib.dll!System.Threading.Tasks.Task.Delay
[Async] 02 - BlockingAsync.dll!_02__BlockingAsync.Diagram.Service.InitializeAsync() Line 12
[Async] 02 - BlockingAsync.dll!_02__BlockingAsync.Diagram.Service.CreateAsync() Line 18
[Async] 02 - BlockingAsync.dll!_02__BlockingAsync.Diagram.Program.Main(string[] args) Line 27
02 - BlockingAsync.dll!_02__BlockingAsync.Diagram.Program.<Main>(string[] args)

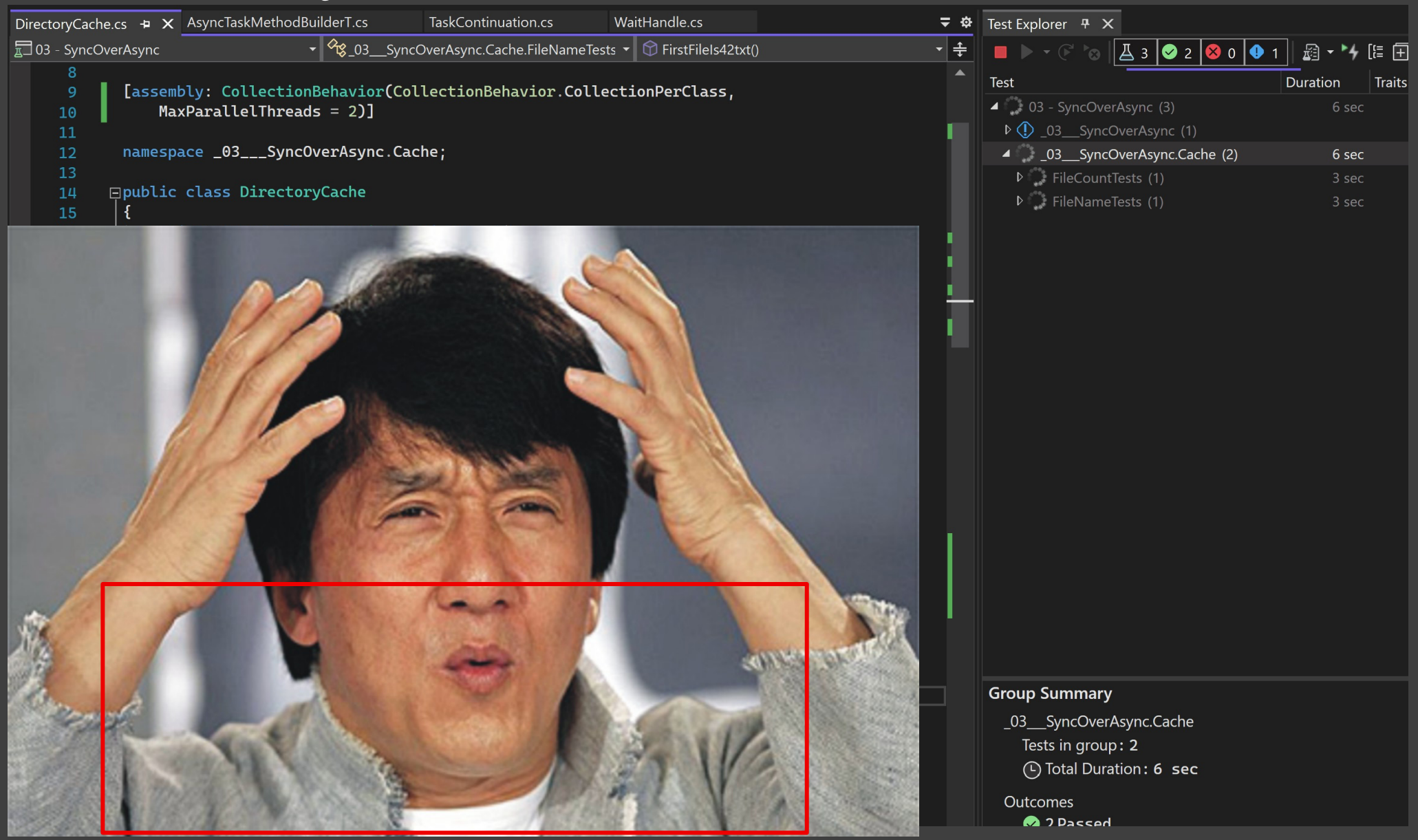
Async code is viral!

Is sync over async dangerous?

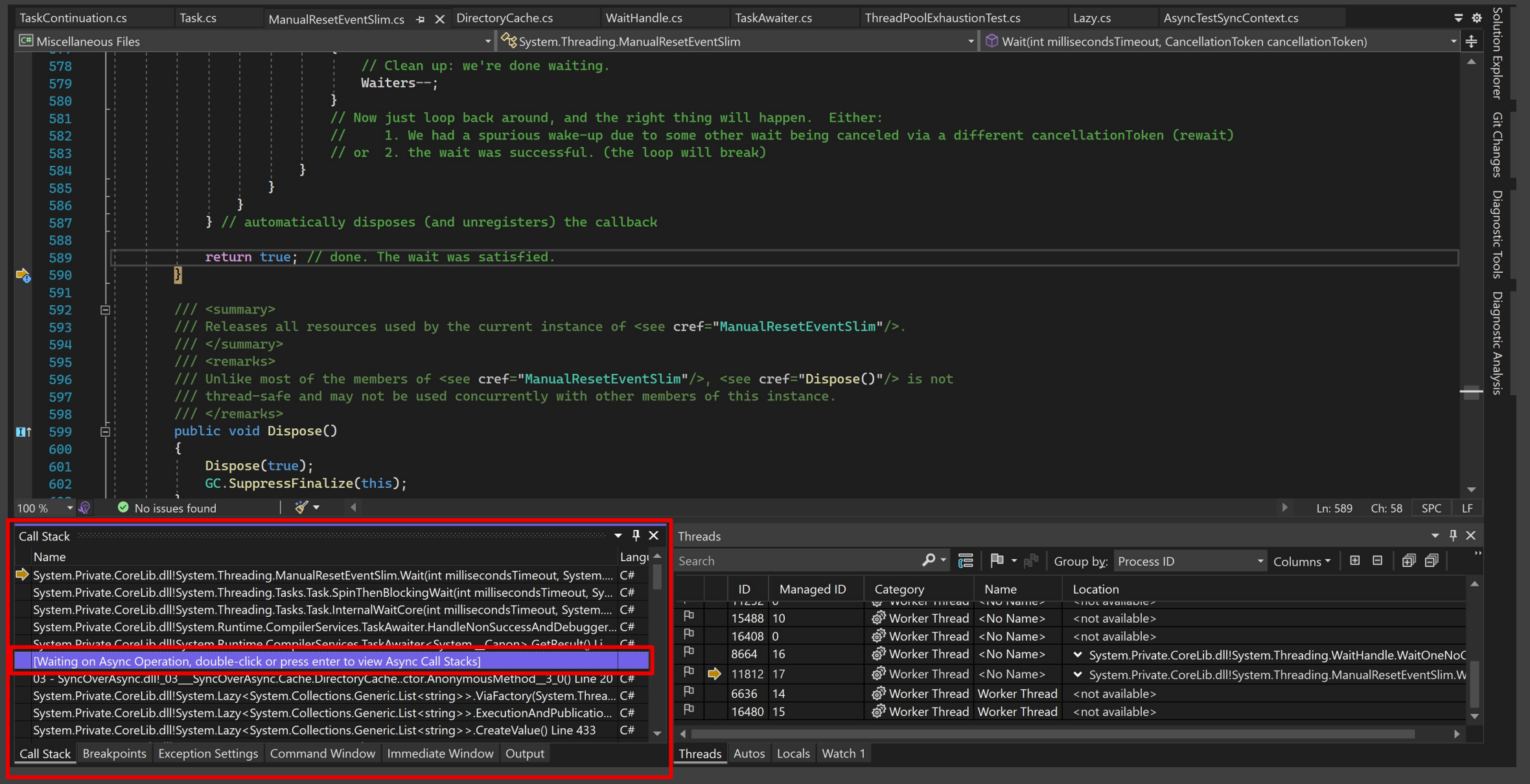
- A simple file system cache
- Initialized lazily to avoid unnecessary work
- Used in many synchronous paths
- Used only on the backend

```
public class DirectoryCache
{
    private readonly Lazy<List<string>> _files;

    public List<string> Files => _files.Value;

    public DirectoryCache()
    {
        _files = new Lazy<List<string>>(() => ReadFilesAsync().GetAwaiter().GetResult());
    }

    private static async Task<List<string>> ReadFilesAsync()
    {
        await Task.Delay(3 * 1000);
        return new List<string> { "42.txt" };
    }
}
```

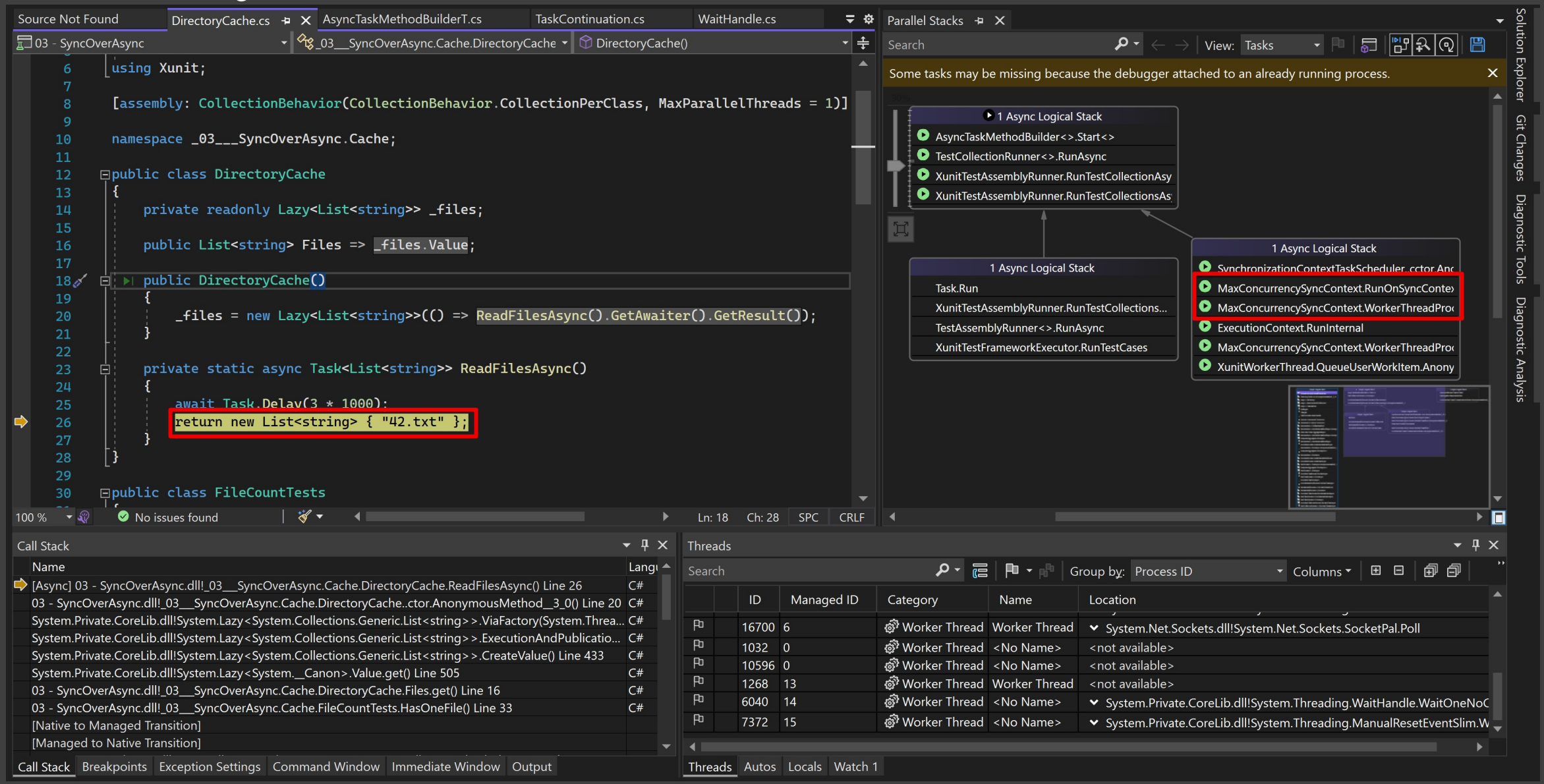
Let's try it out!



Regular stack trace is not very helpful!



Async stack trace & Parallel Tasks



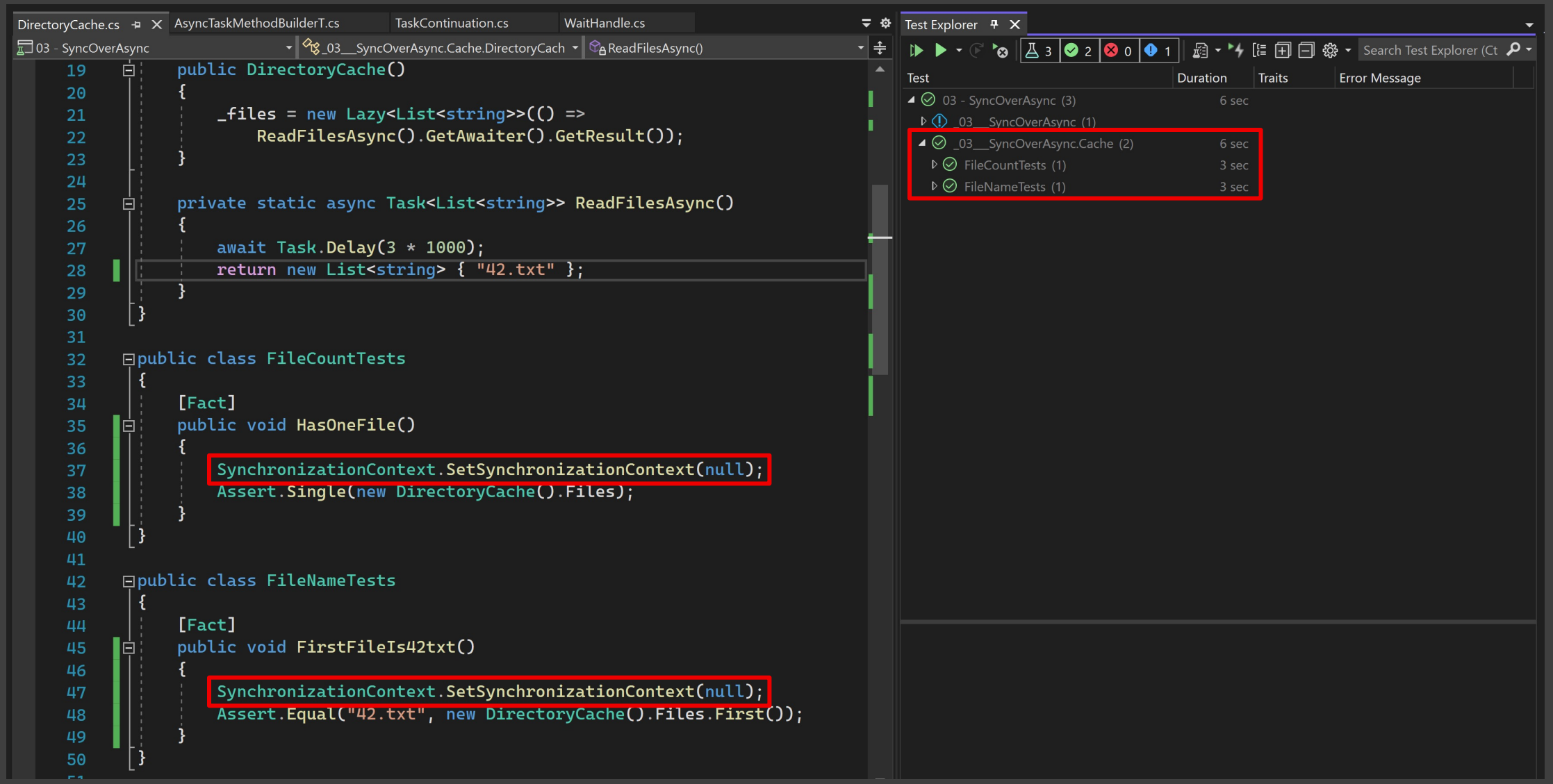
SynchronizationContext is not a purely UI thing...

- Xunit relies on SynchronizationContext for limiting the number of running tests!

```
namespace Xunit.Sdk
{
    /// <summary>
    /// An implementation of <see cref="SynchronizationContext"/> which runs work on custom threads
    /// rather than in the thread pool, and limits the number of in-flight actions.
    /// </summary>
    public class MaxConcurrencySyncContext : SynchronizationContext, IDisposable
    {
        bool disposed = false;
        readonly ManualResetEvent terminate = new ManualResetEvent(false);
        readonly List<XunitWorkerThread> workerThreads;
        readonly ConcurrentQueue<Tuple<SendOrPostCallback, object, object>> workQueue = new ConcurrentQ
        readonly AutoResetEvent workReady = new AutoResetEvent(false);
```

```
[assembly: CollectionBehavior(CollectionBehavior.CollectionPerClass, MaxParallelThreads = 2)]
```

Detaching the sync context



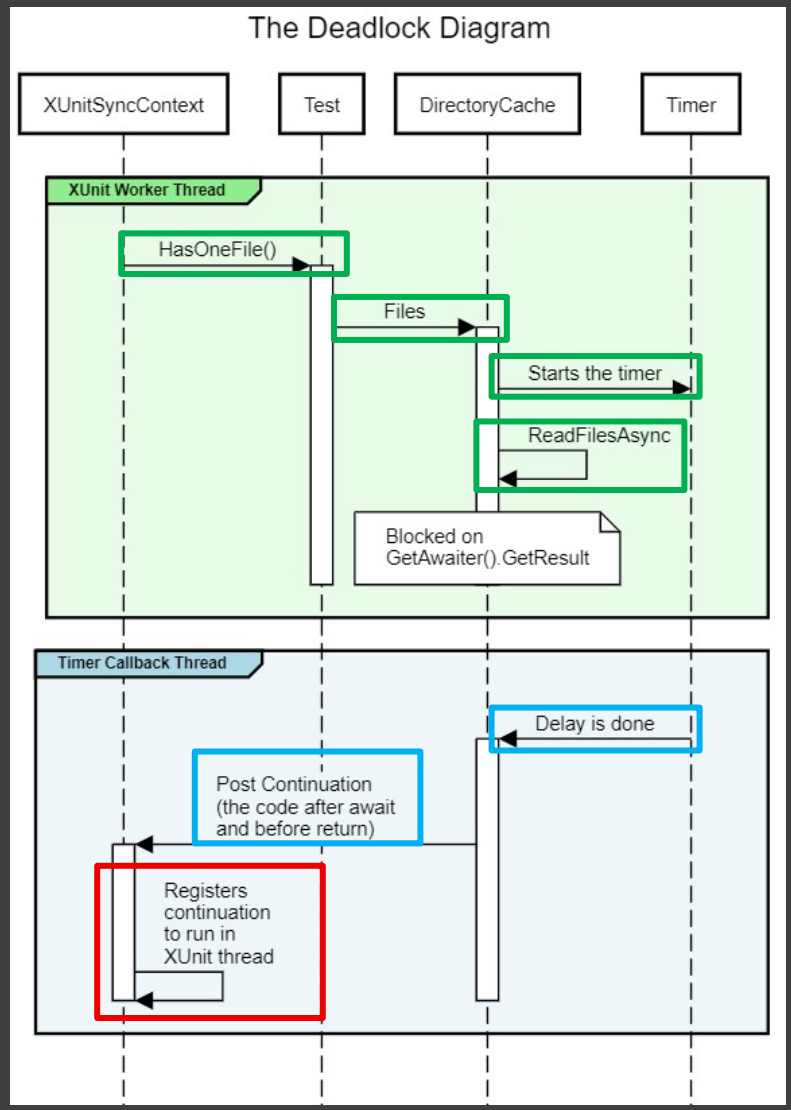
How to fix the issue properly?

- Stop using blocking operations.
  - Preferred, but not always possible.
- Stop using xUnit ☺ (not really!) or change its concurrency limits!
  - The same issue may happen elsewhere as well.
- Use ConfigureAwait(false)
  - Won't fix all the possible issues.
  - A reasonable first step.

```
private static async Task<List<string>> ReadFilesAsync()
{
    // Configuring the runtime to schedule the continuation
    // of 'Task.Delay' in a thread pool thread,
    // instead of running it in the Sync Context
    // captured when this 'Task.Delay' was started.
    await Task.Delay(3 * 1000).ConfigureAwait(false);

    return new List<string> { "42.txt" };
}
```

What is going on at runtime?

```
public class DirectoryCache
{
    private readonly Lazy<List<string>> _files;

    public List<string> Files => _files.Value;

    public DirectoryCache()
    {
        _files = new Lazy<List<string>>(() =>
            ReadFilesAsync().GetAwaiter().GetResult());
    }

    private static async Task<List<string>> ReadFilesAsync()
    {
        await Task.Delay(3 * 1000);

        // When the continuation of Task.Delay finishes, by default
        // all the continautions are invoked in captured sync context,
        // and if the continuation can't be scheduled because the sync
        // context ran out of thread, will get a deadlock!
        return new List<string> { "42.txt" };
    }
}

public class FileCountTests
{
    [Fact]
    public void HasOneFile() =>
        Assert.Single(new DirectoryCache().Files);
}
```



Lessons learned

- This is based on a real issue.
- Parallel Tasks is an invaluable tool for figuring out issues in async code.
- Understanding of how async works is very important.
- Always use ‘ConfigureAwait(false)’ in the library code.
- Always use ‘ConfigureAwait(false)’ if the code will be used with Sync Contexts.
- Consider using ‘ConfigureAwait(false)’ for service code as well.
- Sync Contexts can be used outside the UI world.

What is TaskCompletionSource?

- TCS allows controlling Task's lifetime manually
- A tool for implementing async API based on non-task-based implementations
- Core building block for producer-consumer queues, communication protocols etc.

```
public class TaskCompletionSource<T>
{
    public void SetResult<T>(T result) { }
    public void SetException(Exception e) { }
    public void SetCanceled() { }
    public Task Task { get; }
}
```

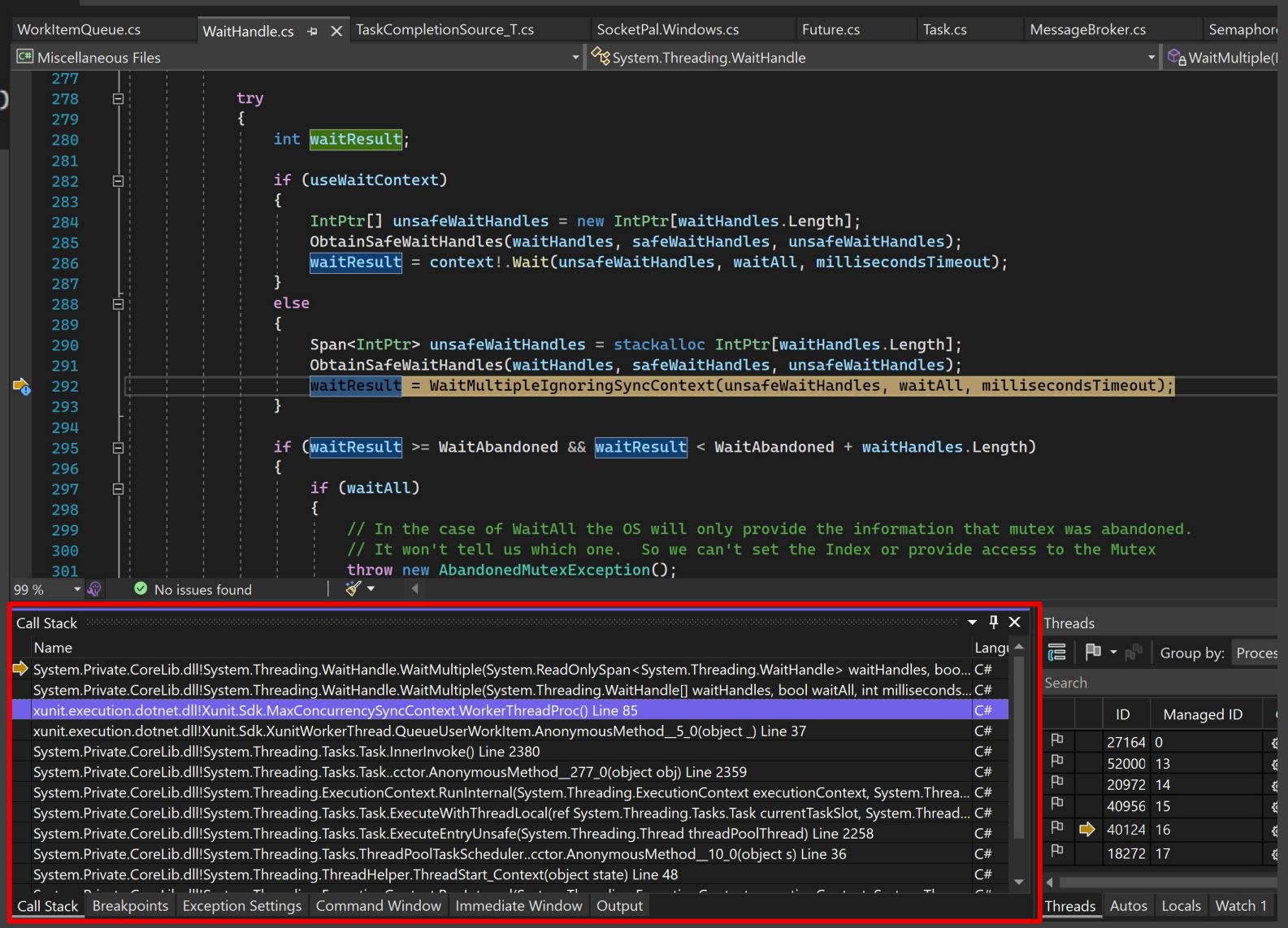
A classical usage of TaskCompletionSource

- A simple work item queue
- Based on BlockingCollection<T> and TaskCompletionSource<T>

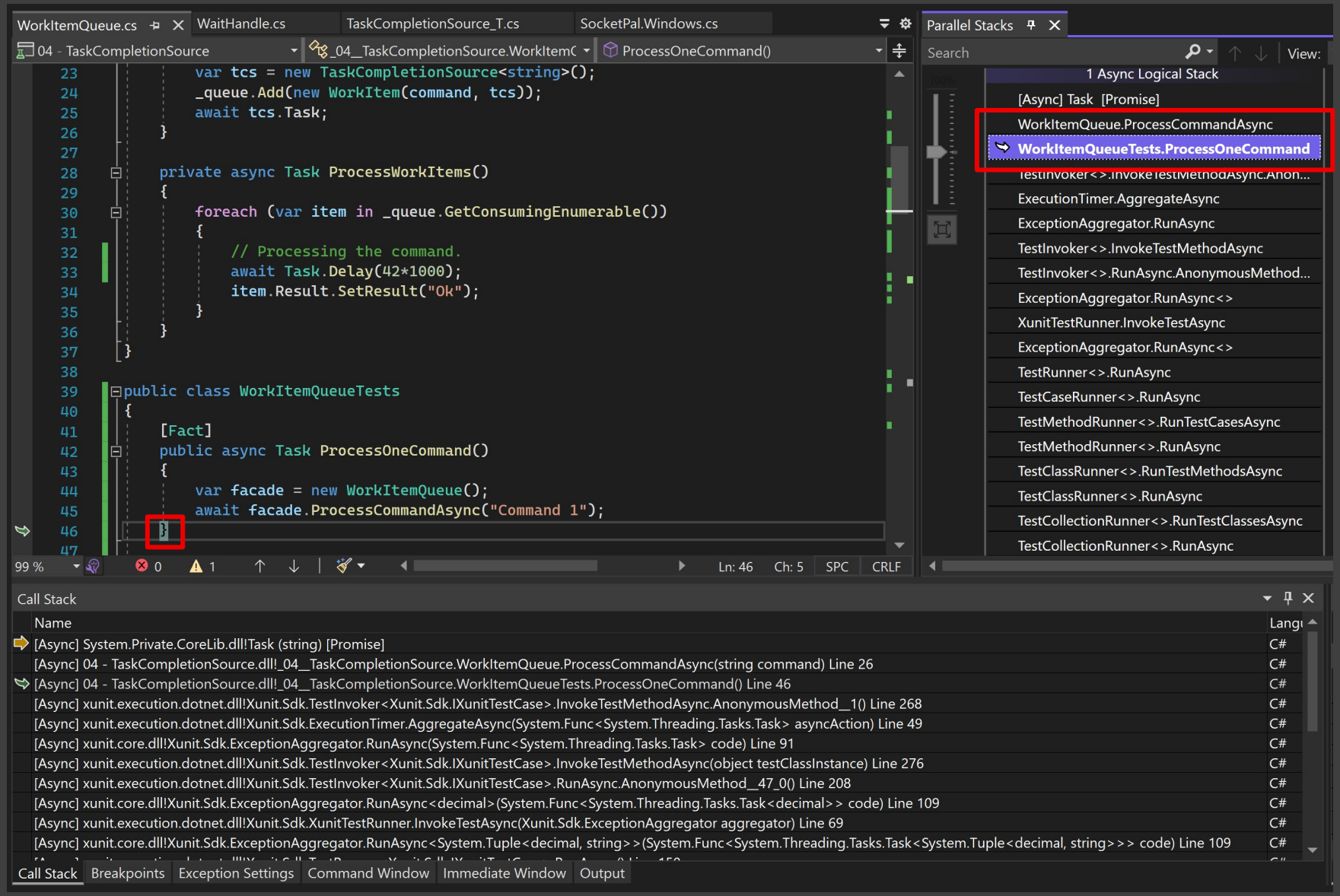
```
public class WorkItemQueue
{
    private record WorkItem(string Command, TaskCompletionSource<string> Result);
    private readonly BlockingCollection<WorkItem> _queue = new ();

    public Task ProcessItemsTask { get; }

    public WorkItemQueue() => ProcessItemsTask = Task.Run(ProcessWorkItems);

    public async Task ProcessCommandAsync(string command)
    {
        var tcs = new TaskCompletionSource<string>();
        _queue.Add(new WorkItem(command, tcs));
        await tcs.Task;
    }

    private async Task ProcessWorkItems()
    {
        foreach (var item in _queue.GetConsumingEnumerable())
        {
            // Processing the command.
            await Task.Delay(42*1000);
            item.Result.SetResult("Ok");
        }
    }
}
```

Debugging an issue with TaskCompletionSource

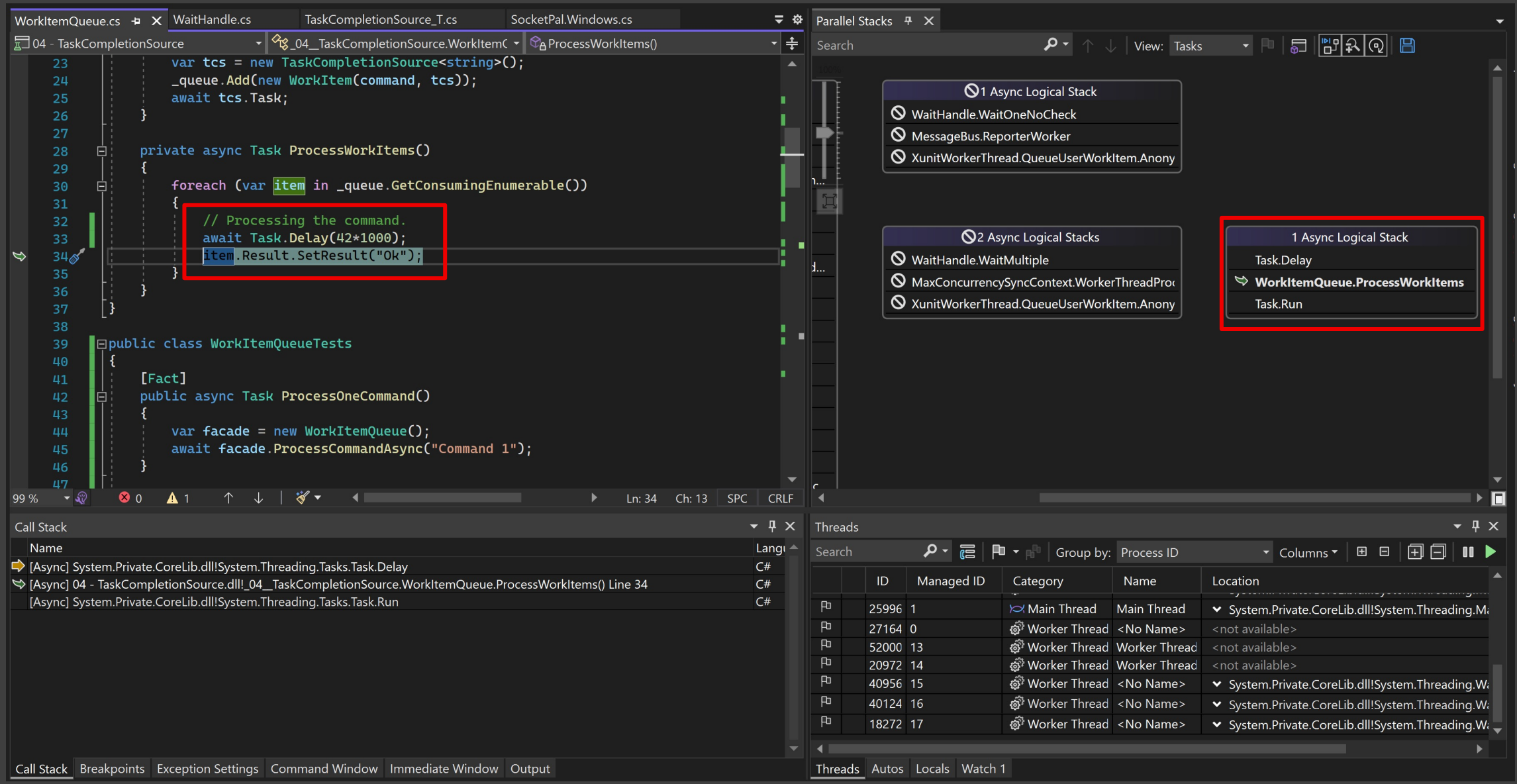
```
[Fact]
public async Task ProcessOneCommand()
{
    var facade = new WorkItemQueue();
    await facade.ProcessCommandAsync("Command 1")
}
```



Async stacks to the rescue! (Again!)

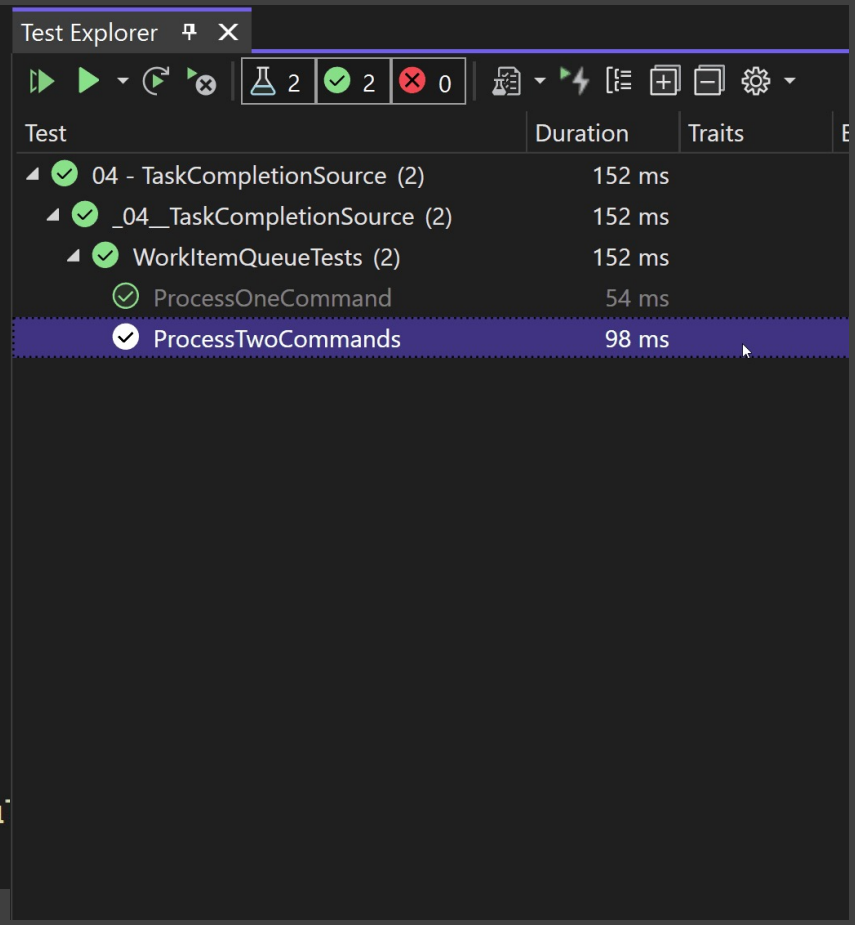


Let's debug it! Stack traces are not helpful;)



Let's add one more test! Shall we?

```
    private async Task ProcessWorkItems()
    {
        foreach (var item in _queue.GetConsumingEnumerable())
        {
            // Processing the command.
            await Task.Delay(42);
            item.Result.SetResult("OK");
        }
    }
}

public class WorkItemQueueTests
{
    [Fact]
    public async Task ProcessTwoCommands()
    {
        var queue = new WorkItemQueue();
        await queue.ProcessCommandAsync("Command 1");
        queue.ProcessCommandAsync("Command 2").GetAwaiter().GetResu
    }
```



Production after deploying a new version

Why the tests were fine?

```
private async Task ProcessWorkItems()
{
    foreach (var item in _queue.GetConsumingEnumerable())
    {
        // Processing the command.
        await Task.Delay(42);
```

Test Explorer

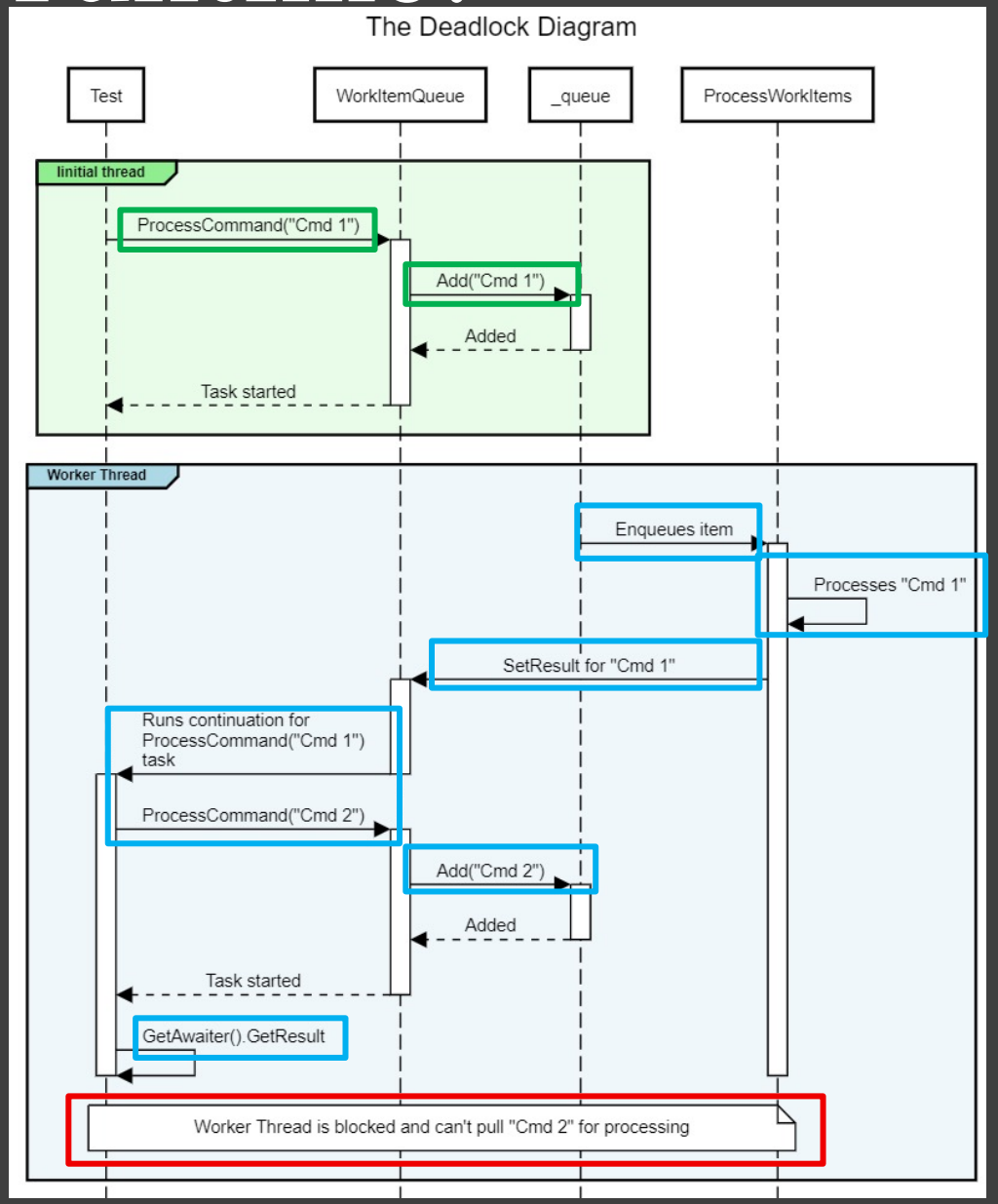
2 2 0

Test	Duration	Traits
04 - TaskCompletionSource (2)	159 ms	
_04__TaskCompletionSource (2)	159 ms	
WorkItemQueueTests (2)	159 ms	
ProcessOneCommand	54 ms	
ProcessTwoCommands	105 ms	

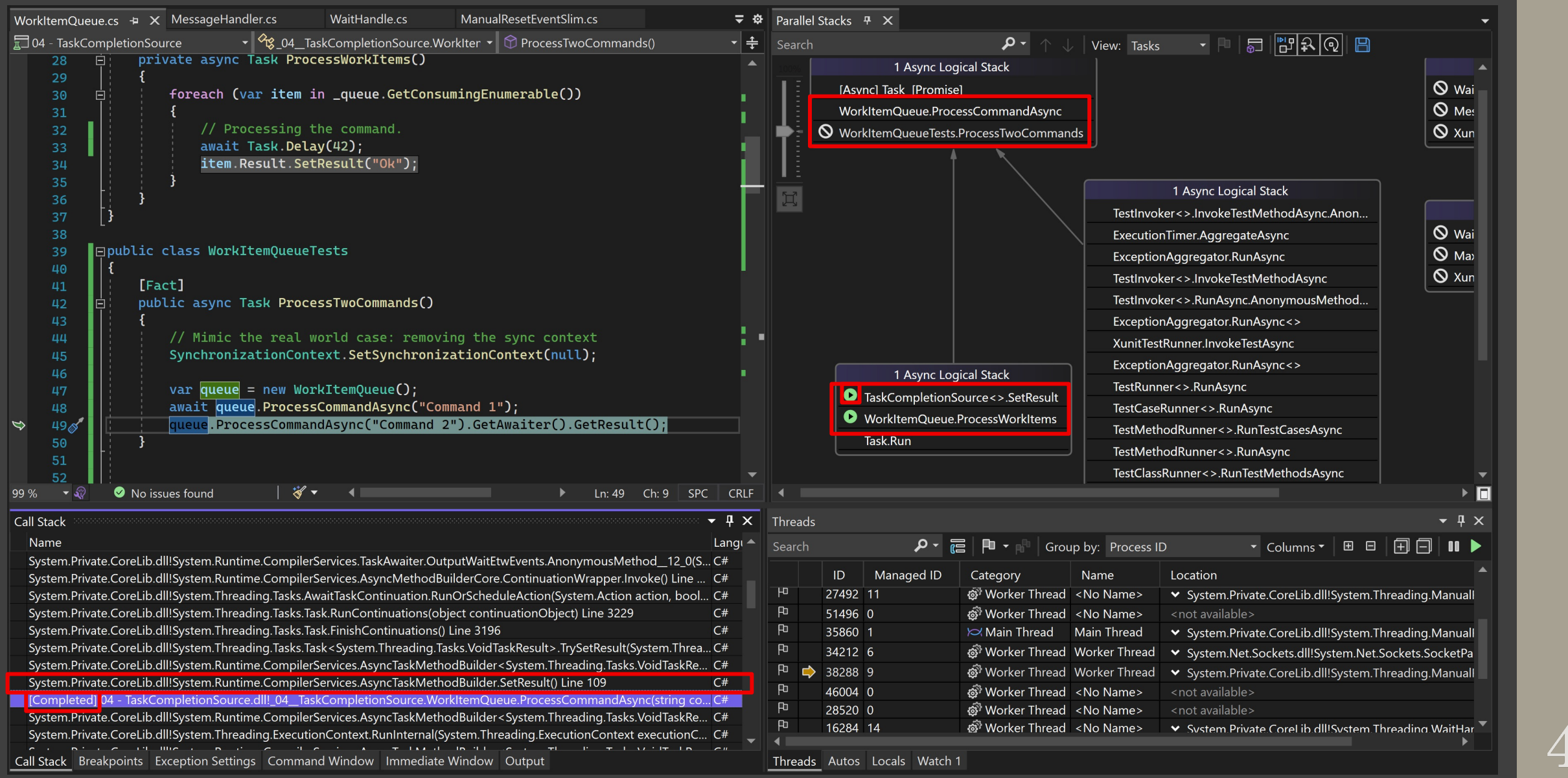

SYNC OVER ASYNC STRIKES AGAIN
38

What is going on at runtime?

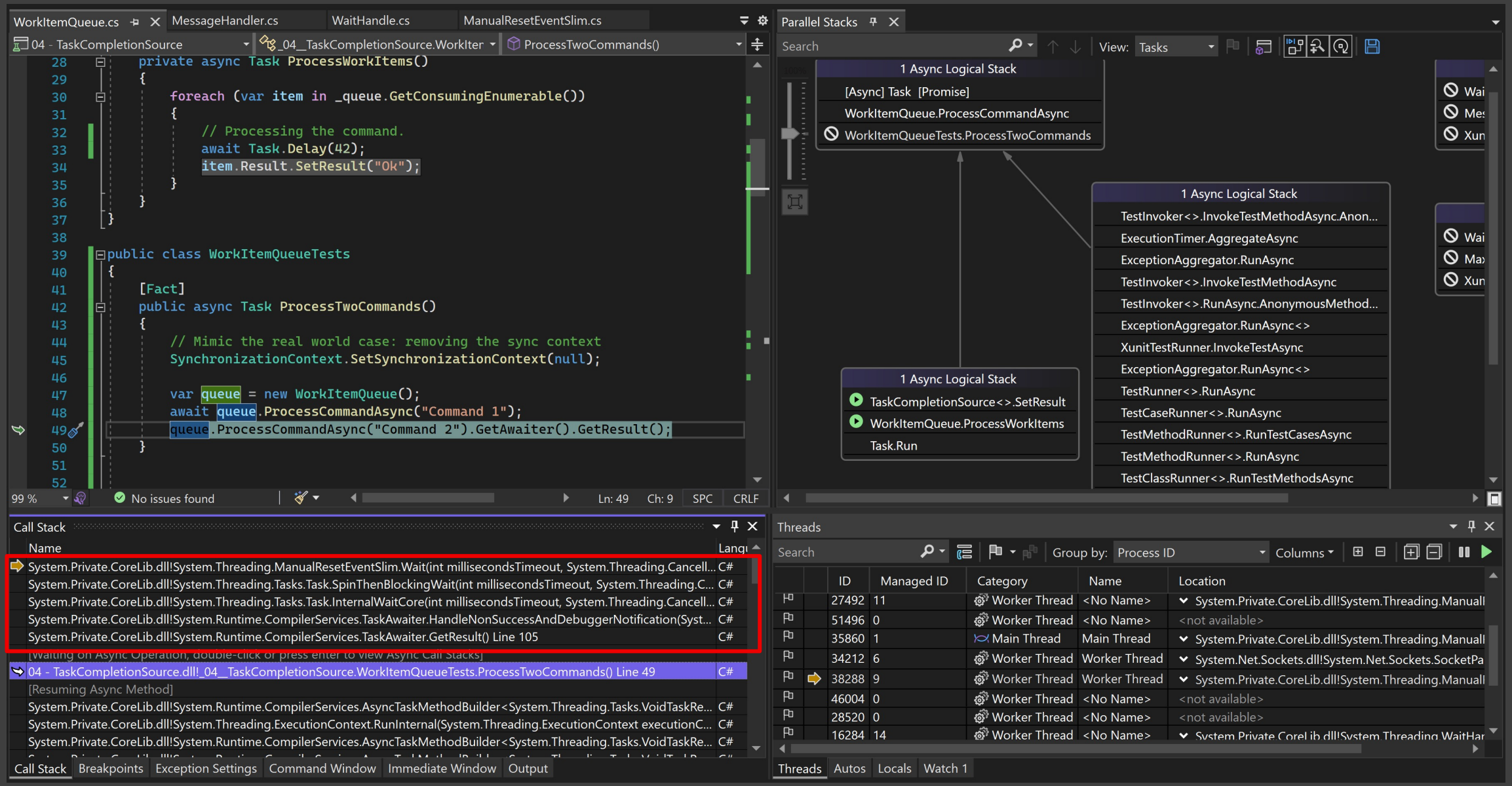
```
    public async Task ProcessCommandAsync(string command)
    {
        var tcs = new TaskCompletionSource<string>();
        _queue.Add(new WorkItem(command, tcs));
        await tcs.Task;
    }

    private async Task ProcessWorkItems()
    {
        foreach (var item in _queue.GetConsumingEnumerable())
        {
            // Processing the command.
            await Task.Delay(42);
            item.Result.SetResult("OK");
        }
    }
}

public class WorkItemQueueTests
{
    [Fact]
    public async Task ProcessTwoCommands()
    {
        // Mimic the real world case: removing the sync context
        SynchronizationContext.SetSynchronizationContext(null);

        var queue = new WorkItemQueue();
        await queue.ProcessCommandAsync("Command 1");
        queue.ProcessCommandAsync("Command 2").GetAwaiter().GetResult();
    }
```



Parallel Stacks to the rescue!



Parallel Stacks to the rescue!



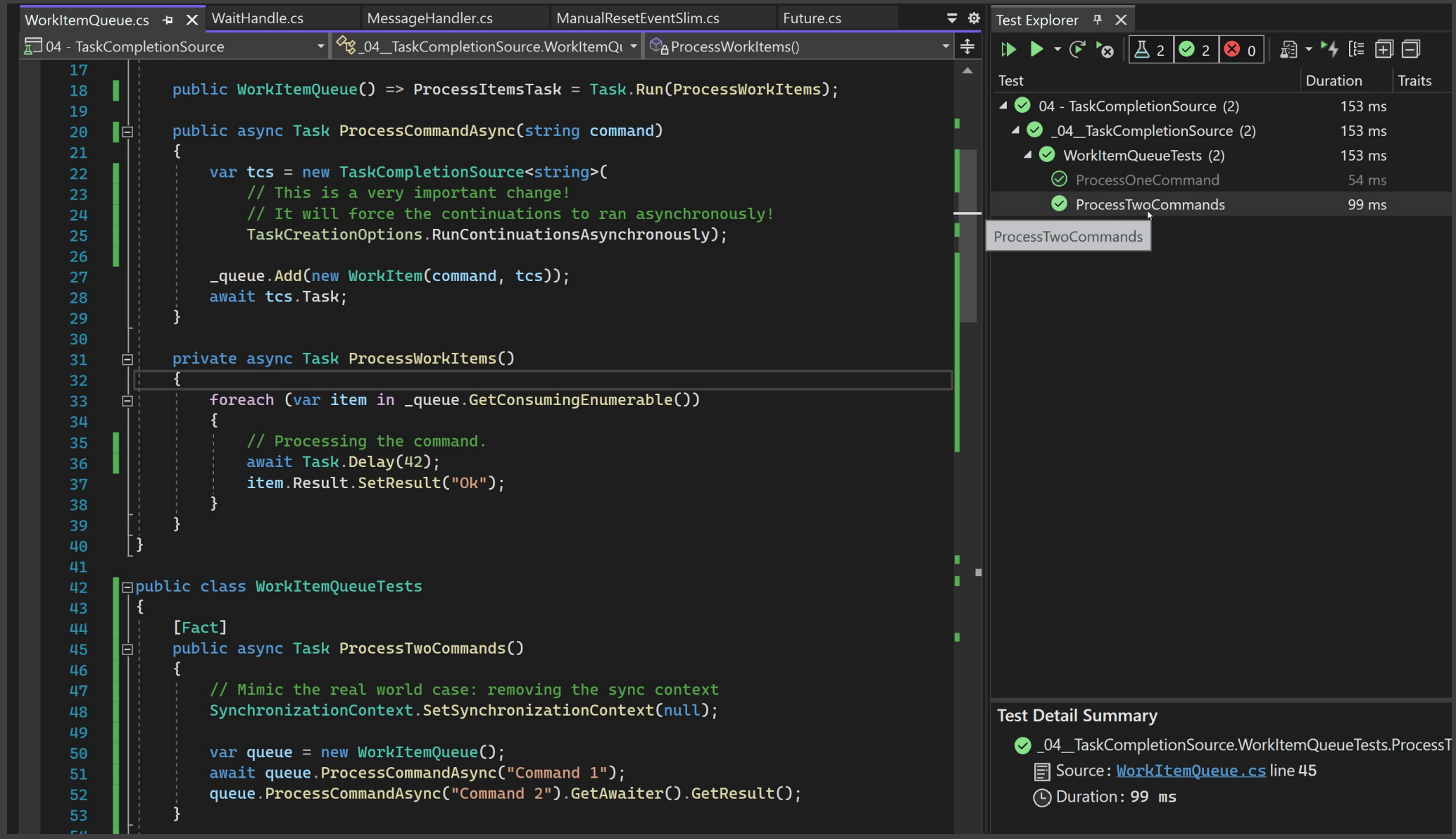
How to solve the problem?

- Don’t block async code (maybe easier said then done!)
- Run TaskCompletionSource’s continuations asynchronously (.NET Framework 4.6+)!
- Consider using a wrapper or a helper for running continuations asynchronously

```
public readonly struct TaskSourceSlim<TResult>
{
    private readonly TaskCompletionSource<TResult> m_tcs;

    internal TaskSourceSlim(bool dummy)
        : this()
    {
        m_tcs = new TaskCompletionSource<TResult>(
            TaskCreationOptions.RunContinuationsAsynchronously);
    }

    public Task<TResult> Task => m_tcs.Task;

    // SetResult, SetException, SetCanceled and other members
}

public static class TaskSourceSlim
{
    public static TaskSourceSlim<TResult> Create<TResult>()
        => new TaskSourceSlim<TResult>(dummy: true);
}
```

```
public class WorkItemQueue
{
    private record WorkItem(string Command, TaskCompletionSource<string> Result);
    private readonly BlockingCollection<WorkItem> _queue = new ();

    public Task ProcessItemsTask { get; }

    public WorkItemQueue() => ProcessItemsTask = Task.Run(ProcessWorkItems);

    public async Task ProcessCommandAsync(string command)
    {
        var tcs = new TaskCompletionSource<string>(
            // This is a very important change!
            // It will force the continuations to ran asynchronously!
            TaskCreationOptions.RunContinuationsAsynchronously);

        _queue.Add(new WorkItem(command, tcs));
        await tcs.Task;
    }

    private async Task ProcessWorkItems()
    {
        foreach (var item in _queue.GetConsumingEnumerable())
        {
            // Processing the command.
            await Task.Delay(42);
            item.Result.SetResult("Ok");
        }
    }
}
```

Let's check it!

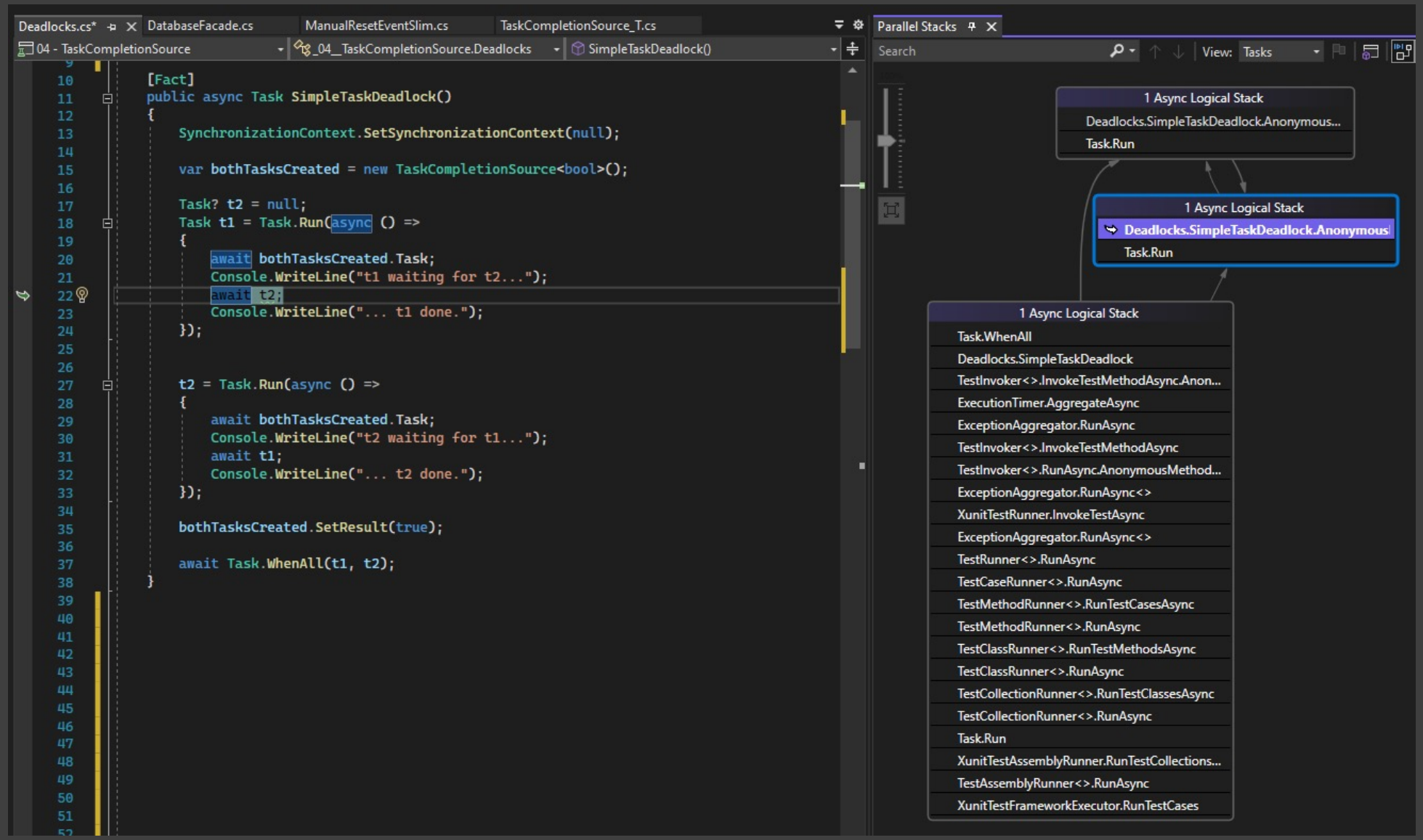


Lessons learned

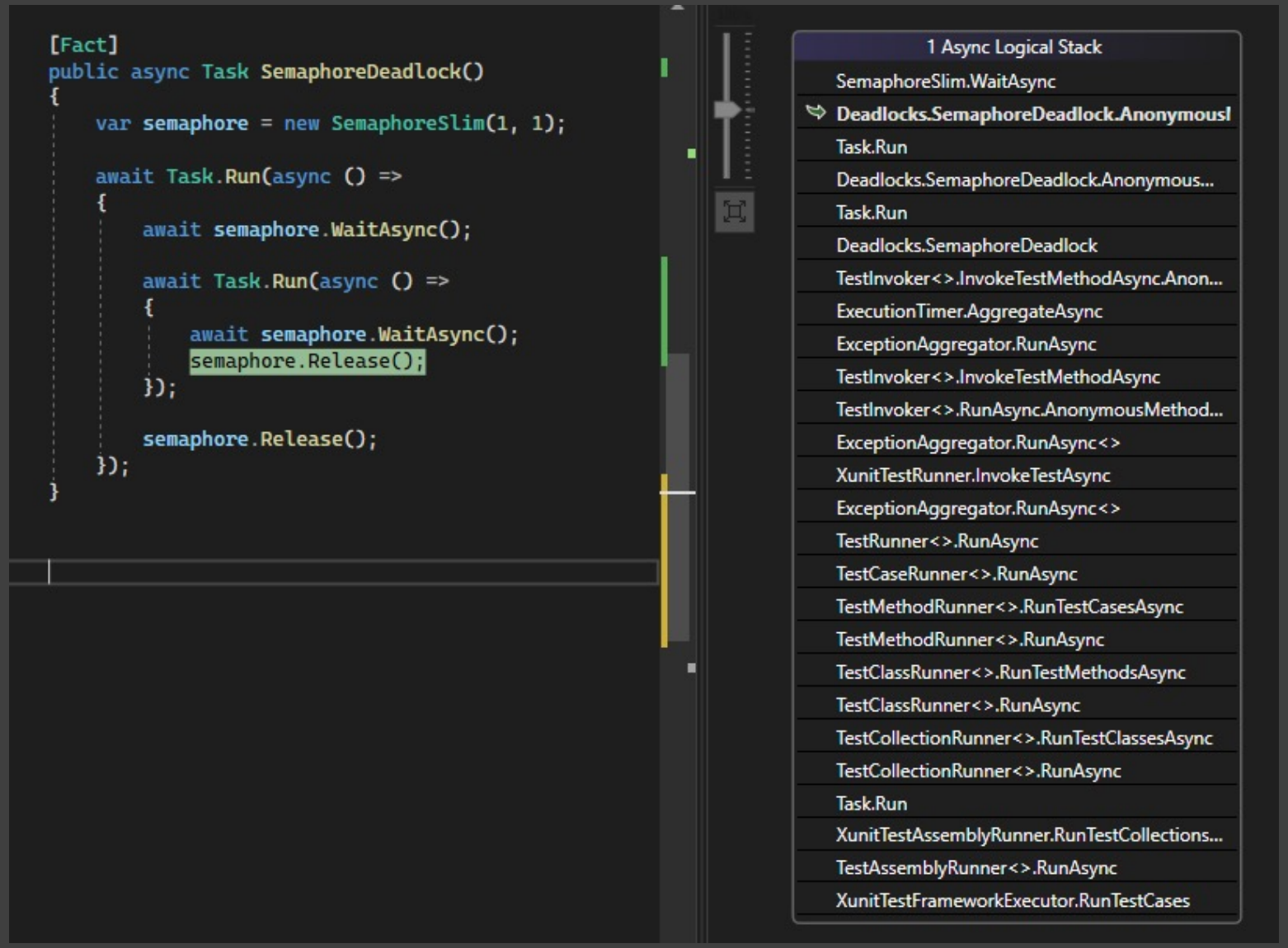
- Synchronous behavior of TaskCompletionSource.SetResult is quite dangerous.
- Blocking can be hidden. ‘GetConsumingEnumerable’ is a blocking call.
- Always force async continuations when using TaskCompletionSource.
  - This will detach the caller from synchronously calling unknown code.
- Consider using a helper or a wrapper for that.

Thanks a lot, everyone!

[Bonus] Deadlock with two tasks



[Bonus] Deadlock a semaphore



[Bonus] Deadlock with yield return and lock

```
[Fact]
public async Task YieldReturnDeadlock()
{
    object _syncObj = new object();
    foreach (var i in GetEnum())
    {
        await UseAsync(i).ConfigureAwait(false);
    }

    IEnumerable<int> GetEnum()
    {
        lock (_syncObj)
        {
            yield return GetValue(0);
            yield return GetValue(1);
        }
    }

    int GetValue(int value)
    {
        lock (_syncObj)
        {
            return value + 1;
        }
    }

    Task UseAsync(int i) => Task.Delay(i + 100);
}
```