Async - Biggest C# Mistake

CONTACT@ADAMFURMANEK.PL

HTTP://BLOG.ADAMFURMANEK.PL

FURMANEKADAM

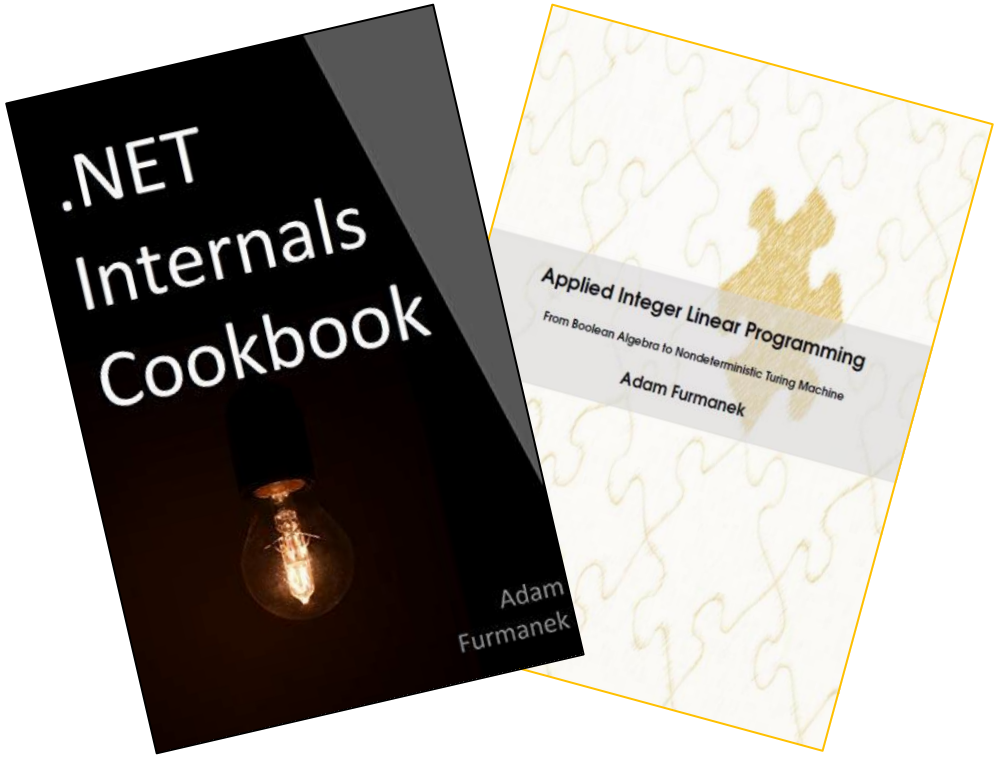
About me

Software Engineer, Blogger, Book Writer, Public Speaker.
Author of ***Applied Integer Linear Programming*** and ***.NET Internals Cookbook***.

http://blog.adamfurmanek.pl

contact@adamfurmanek.pl

furmanekadam



Random IT Utensils

IT, operating systems, maths, and more.

Agenda

It's expensive.

It uses hidden state.

It doesn't integrate with the platform.

It breaks decent programming principles.

Let's make it better!

It's expensive

Async Method

```
public static async Task OurAsyncMethod()
{
    Console.WriteLine("First part");
    await Task.FromResult(false);

    Console.WriteLine("Second part");
    await Task.Delay(200);

    Console.WriteLine("Third part");
    await Task.Yield();

    Console.WriteLine("Fourth part");
    throw new Exception();
}
```

Async Method after compilation

```
// Token: 0x06000002 RID: 2 RVA: 0x00002060 File Offset: 0x00000260
[DebuggerStepThrough]
public static Task OurAsyncMethod()
{
    Program.<OurAsyncMethod>d__1 <OurAsyncMethod>d__ = new Program.<OurAsyncMethod>d__1();
    <OurAsyncMethod>d__.<>t__builder = AsyncTaskMethodBuilder.Create();
    <OurAsyncMethod>d__.<>1__state = -1;
    AsyncTaskMethodBuilder <>t__builder = <OurAsyncMethod>d__.<>t__builder;
    <>t__builder.Start<Program.<OurAsyncMethod>d__1>(ref <OurAsyncMethod>d__);
    return <OurAsyncMethod>d__.<>t__builder.Task;
}
```

ValueTask

Task is a class so it is allocated on the heap, and needs to be collected by the GC.

To avoid explicit allocation, we can use *ValueTask* which is a struct, and is allocated on the stack.

The trick is in the second constructor parameter — the token.

```
public ValueTask(IValueTaskSource<T> source, short token);
```

See https://github.com/kkokosa/PooledValueTaskSource

Conceptually it was used in *Midori* — .NET-based operating system implemented by Microsoft Research.

„It still kills me that I can't go back in time and make .NET's task a struct" — Joe Duffy in http://joeduffyblog.com/2015/11/19/asynchronous-everything/

ValueTaskSource

```
public interface IValueTaskSource<out TResult>
{
    ValueTaskSourceStatus GetStatus(short token);
    void OnCompleted(Action<object> continuation, object state, short token,
                     ValueTaskSourceOnCompletedFlags flags);
    TResult GetResult(short token);
}
```

This can be reused! Whenever you await the task, it is allowed to reset the state.

await the task only once!

Getting result is allowed if and only if the result is available. *GetAwaiter().GetResult()* may not block, is not required to be thread-safe, may crash your application.

Results

```
BenchmarkDotNet=v0.13.1, OS=Windows 10.0.19042.1466 (20H2/October2020Update)
Intel Core i7-8565U CPU 1.80GHz (Whiskey Lake), 1 CPU, 8 logical and 4 physical cores
.NET SDK=5.0.402
  [Host]     : .NET 5.0.11 (5.0.1121.47308), X64 RyuJIT
  DefaultJob : .NET 5.0.11 (5.0.1121.47308), X64 RyuJIT


|           Method |         Mean |      Error |     StdDev | Ratio | RatioSD |        Gen 0 |      Allocated |
|----------------- |-------------:|-----------:|-----------:|------:|--------:|-------------:|---------------:|
|           NoTask |     11.43 ms |   0.113 ms |   0.106 ms |  1.00 |    0.00 |            - |              - |
|        TaskAsync | 1,109.96 ms |  15.060 ms |  13.350 ms | 97.03 |    0.95 | 191000.0000 | 799,740,720 B |
|      TaskNoAsync |     93.46 ms |   1.838 ms |   1.719 ms |  8.17 |    0.15 |  96000.0000 | 401,770,599 B |
|   ValueTaskAsync |    274.64 ms |   5.355 ms |   5.009 ms | 24.02 |    0.54 |            - |              - |
| ValueTaskNoAsync |     32.30 ms |   0.348 ms |   0.465 ms |  2.83 |    0.05 |            - |              - |
```

State machine is expensive.

Each '*await*' operation for an unfinished task takes about 4us and allocates almost 300B per invocation.

If the *async* method completes synchronously the following memory overhead will occur: for *async Task* methods there is no overhead, for *async Task<T>* methods the overhead is 88 bytes per operation (on x64 platform).

ValueTask<T> can remove the overhead mentioned above for async methods that complete synchronously.

A *ValueTask<T>*-based async method is a bit faster than a *Task<T>*-based method if the method completes synchronously, and a bit slower otherwise.

A performance overhead of *async* methods that await non-completed task is way more substantial (~300 bytes per operation on x64 platform).

https://devblogs.microsoft.com/premier-developer/the-performance-characteristics-of-async-methods/

It uses hidden state

SynchronizationContext

	Specific thread executing the code	Delegates executed serially	Delegates executed in order	Send is synchronous	Post is asynchronous
Default (Thread Pool based)	No – any thread in the thread pool	No	No	Yes	Yes
ASP.NET	No – any thread in the thread pool	Yes	No	Yes	No
WinForms, WPF, WinRT, Xamarin, Blazor	Yes – UI thread	Yes	Yes	Only if called on the UI thread	Yes
ASP.NET Core	No – any thread in the thread pool	No	No	Yes	Yes

In ASP.NET only one continuation can be executed at a time for given request - no concurrency.
In ASP.NET Core multiple **continuations can run concurrently** – we have concurrency and parallelism.

Deadlocks

Because **there is no thread** we can cause a **deadlock with just one thread**!

Depending on the application type our code may run correctly or not.

Use async all the way up!

Use *ConfigureAwait(false)* all the way down!

SynchronizationContext

```
async void DownloadButton_Click(object sender, EventArgs e)
{
    // We are on the UI thread but we don’t block it
    await ProcessDataAsync();

    // We are back on the UI thread
    resultTextBox.Text = "Done";
}

async Task ProcessDataAsync()
{
    // We are still on the UI thread
    var content = await DownloadAsync().ConfigureAwait(false);

    // Because of ConfigureAwait we are most likely *not* on the UI thread but on the thread pool
    // However, ConfigureAwait(false) *is* still required because of possible synchronous execution
    // Always use ConfigureAwait(false) unless you really want to capture the context
    await TransformAsync(content).ConfigureAwait(false);
}
```

Console

Program.cs

AsyncSyncWait

AsyncSyncWait.Program

```
1   using System;
2   using System.Threading.Tasks;
3
4   namespace AsyncSyncWait
5   {
6       class Program
7       {
8           static void Main(string[] args)
9           {
10              async Task operationOnContext()
11              {
12                  Console.WriteLine("\tWaiting");
13                  await Task.Delay(1000);
14                  Console.WriteLine("\tDone waiting");
15              }
16              Console.WriteLine("Waiting");
17              operationOnContext().Wait();
18              Console.WriteLine("Done!");
19          }
20      }
21  }
22
```

C:\WINDOWS\system32\cmd.exe

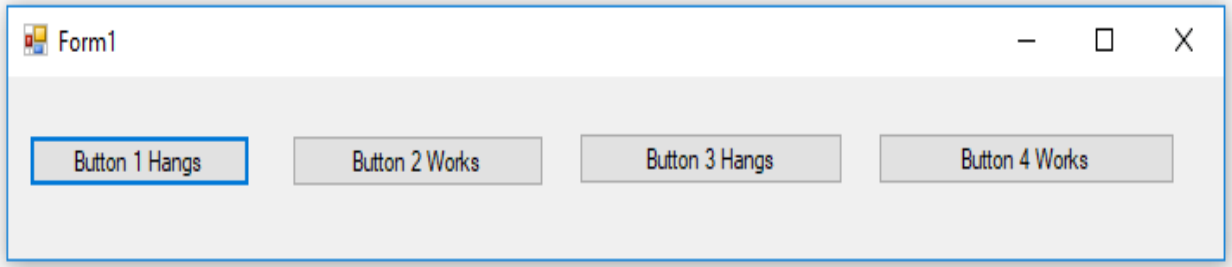
```
Waiting
        Waiting
        Done waiting
Done!
Press any key to continue . . .
```

GUI

Form1.cs

AsyncSyncWait_Forms | AsyncSyncWait_Forms.Form1 | InitializeComponent()

```
1   using System;
2   using System.Threading;
3   using System.Threading.Tasks;
4   using System.Windows.Forms;
5
6   namespace AsyncSyncWait_Forms
7   {
8       public partial class Form1 : Form
9       {
10          public Form1()
11          {
12              InitializeComponent();
13          }
14
15          private void button1_Click(object sender, EventArgs e)
16          {
17              async Task operationOnContext()
18              {
19                  await Task.Delay(1000);
20                  MessageBox.Show("Waiting on context finished!");
21              }
22              operationOnContext().Wait();
23              MessageBox.Show("Done");
24          }
25
26          private void button2_Click(object sender, EventArgs e)...
36
37          private void button3_Click(object sender, EventArgs e)...
48
49          private void button4_Click(object sender, EventArgs e)...
72      }
73  }
74
```

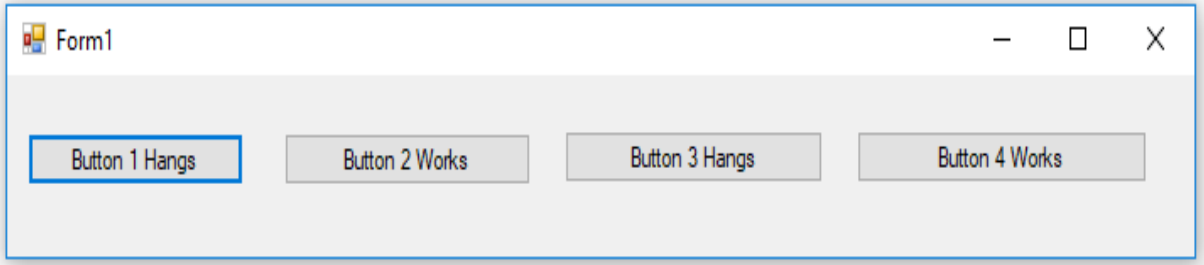


GUI

Form1.cs

AsyncSyncWait_Forms | AsyncSyncWait_Forms.Form1 | InitializeComponent()

```
1   using System;
2   using System.Threading;
3   using System.Threading.Tasks;
4   using System.Windows.Forms;
5
6   namespace AsyncSyncWait_Forms
7   {
8       public partial class Form1 : Form
9       {
10          public Form1()
11          {
12              InitializeComponent();
13          }
14
15          private void button1_Click(object sender, EventArgs e)...
25
26          private void button2_Click(object sender, EventArgs e)
27          {
28              async Task operationOnThreadPool()
29              {
30                  await Task.Delay(1000).ConfigureAwait(false);
31                  MessageBox.Show("Waiting on thread pool finished!");
32              }
33              operationOnThreadPool().Wait();
34              MessageBox.Show("Done");
35          }
36
37          private void button3_Click(object sender, EventArgs e)...
48
49          private void button4_Click(object sender, EventArgs e)...
72      }
73  }
74
```

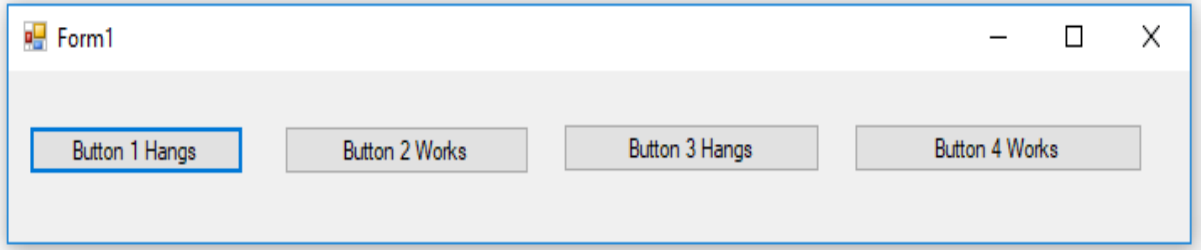


GUI

Form1.cs

AsyncSyncWait_Forms | AsyncSyncWait_Forms.Form1 | InitializeComponent()

```
1   using System;
2   using System.Threading;
3   using System.Threading.Tasks;
4   using System.Windows.Forms;
5
6   namespace AsyncSyncWait_Forms
7   {
8       public partial class Form1 : Form
9       {
10          public Form1()
11          {
12              InitializeComponent();
13          }
14
15          private void button1_Click(object sender, EventArgs e)...
25
26          private void button2_Click(object sender, EventArgs e)...
36
37          private void button3_Click(object sender, EventArgs e)
38          {
39              async Task operationOnThreadPool()
40              {
41                  await Task.Delay(1000).ConfigureAwait(false);
42                  Invoke((MethodInvoker)(() => MessageBox.Show("Posting from context")));
43                  MessageBox.Show("Waiting on context finished!");
44              }
45              operationOnThreadPool().Wait();
46              MessageBox.Show("Done");
47          }
48
49          private void button4_Click(object sender, EventArgs e)...
72      }
73  }
74
```

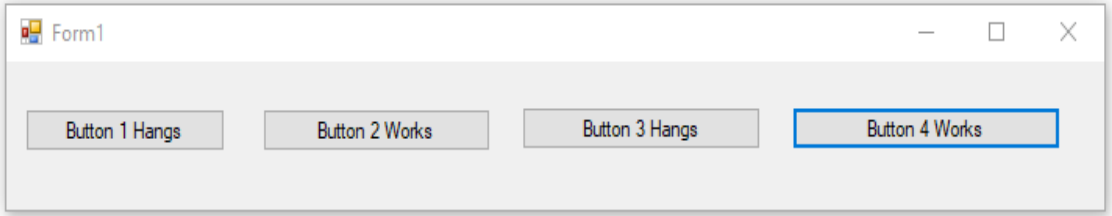


GUI

Form1.cs

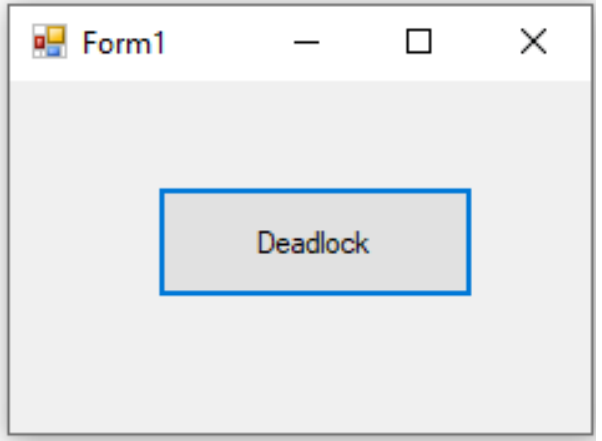
AsyncSyncWait_Forms | AsyncSyncWait_Forms.Form1 | button4_Click(object sender, EventArgs e)

```
1   using System;
2   using System.Threading;
3   using System.Threading.Tasks;
4   using System.Windows.Forms;
5
6   namespace AsyncSyncWait_Forms
7   {
8       public partial class Form1 : Form
9       {
10          public Form1()
11          {
12              InitializeComponent();
13          }
14
15          private void button1_Click(object sender, EventArgs e)...
25
26          private void button2_Click(object sender, EventArgs e)...
36
37          private void button3_Click(object sender, EventArgs e)...
48
49          private void button4_Click(object sender, EventArgs e)
50          {
51              async Task operationOnThreadPool()
52              {
53                  await Task.Delay(1000).ConfigureAwait(false);
54                  Invoke((MethodInvoker)(() => MessageBox.Show("Posting from context")));
55                  MessageBox.Show("Waiting on context finished!");
56              }
57
58              var task = operationOnThreadPool();
59
60              while (!task.IsCompleted)
61              {
62                  Application.DoEvents();
63                  Thread.Sleep(TimeSpan.FromMilliseconds(1));
64              }
65
66              MessageBox.Show("Done");
67          }
68      }
69  }
70
```



GUI

```
using System.Threading.Tasks;
using System.Windows.Forms;

namespace ConfigureAwaitFalse
{
    3 references
    public partial class Form1 : Form
    {
        1 reference
        public Form1()...

        1 reference
        private void button1_Click(object sender, EventArgs e)
        {
            Wait1().GetAwaiter().GetResult();
        }

        1 reference
        private async Task Wait1()
        {
            await Wait2().ConfigureAwait(false);
        }

        1 reference
        private async Task Wait2()
        {
            await Wait3();
        }

        1 reference
        private async Task Wait3()
        {
            await Task.Delay(1000).ConfigureAwait(false);
        }
    }
}
```



Unit test

FormsUnitTest — FormsUnitTest.Program

```
1  using NUnit.Framework;
2  using System.Threading.Tasks;
3  using System.Windows.Forms;
4
5  namespace FormsUnitTest
6  {
7      [TestFixture]
8      class Program
9      {
10
11         [Test]
12         public async Task Test()
13         {
14             // Arrange
15             new Form();
16
17             // Act
18             await Task.Delay(100);
19
20             // Assert
21             Assert.Pass();
22         }
23     }
24 }
```

Unit test

FormsUnitTest — FormsUnitTest.Program

```
1   using NUnit.Framework;
2   using System.Threading;
3   using System.Threading.Tasks;
4   using System.Windows.Forms;
5
6   namespace FormsUnitTest
7   {
8       [TestFixture]
9       class Program
10      {
11          [Test]
12          public async Task Test()
13          {
14              // Arrange
15              var context = SynchronizationContext.Current;
16              new Form();
17              SynchronizationContext.SetSynchronizationContext(context);
18
19              // Act
20              await Task.Delay(100);
21
22              // Assert
23              Assert.Pass();
24          }
25      }
26  }
```

Blazor

```
@code {
    void ShowPopup()
    {
        async Task operationOnContext()
        {
            Console.WriteLine("Calling some JS");
            await JsRuntime.InvokeVoidAsync("alert", "Waiting on context finished!");
            Console.WriteLine("Done calling JS");
        }

        operationOnContext().Wait();
        Console.WriteLine("Done done");
    }
}
```

Blazor

```
@code {
    void ShowPopup()
    {
        async Task operationOnContext()
        {
            Console.WriteLine("Calling some JS");
            await JsRuntime.InvokeVoidAsync("alert", "Waiting on context finished!");
            Console.WriteLine("Done calling JS");
        }

        var task = operationOnContext();
        while(!task.IsCompleted){
            Thread.Sleep(1000);
        }
        Console.WriteLine("Done done");
    }
}
```

Blazor

```
@code {
    void ShowPopup()
    {
        async Task operationOnContext()
        {
            Console.WriteLine("Calling some JS");
            await JsRuntime.InvokeVoidAsync("alert", "Waiting on context finished!").ConfigureAwait(false);
            Console.WriteLine("Done calling JS");
        }


        var task = operationOnContext();
        while(!task.IsCompleted){
            Thread.Sleep(1000);
        }
        Console.WriteLine("Done done");
    }
}
```

Blazor

```
@code {
    void ShowPopup()
    {
        async Task operationOnContext()
        {
            Console.WriteLine("Calling some JS");
            await JsRuntime.InvokeVoidAsync("alert", "Waiting on context finished!").ConfigureAwait(false);
            Console.WriteLine("Done calling JS");
        }


        var task = operationOnContext();
        while(!task.IsCompleted){
            Thread.Sleep(1000);
        }
        Console.WriteLine("Done done");
    }
}
```

It doesn't integrate with the platform

Stacking threads

Task can create a child. It can be either attached or detached.

Attached child task is coupled:

- Parent waits for it
- Parent propagates its exceptions.

Task can block others from attaching by specifying *DenyChildAttach*. In that case child executes normally when trying to attach to the parent.

AggregateException

Contains all *InnerExceptions* – if a *Task* has child task, the exceptions create a tree (instead of a list).

Has method *Flatten* which makes a list from the exception tree.

Even if only one exception is thrown, it is still wrapped.

Can be retrieved by waiting for the task or by checking its *Exception* property.

Contains method *Handle* which takes care of rethrowing exception if it is not of a correct type.

Works *weird* for *await* code.

AggregateException

```
1  using System;
2  using System.Threading.Tasks;
3
4  namespace AggregateException
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             CreateAndAwait().Wait();
11             //CreateAndWait().Wait();
12         }
13
14         static Task CreateAndWait()...
22
23         static async Task CreateAndAwait()
24         {
25             await CreateTask();
26         }
27
28         static Task CreateTask()
29         {
30             return Task.Factory.StartNew(() =>
31             {
32                 Task.Factory.StartNew(() => { throw new Exception("FIRST TASK EXCEPTION!!!"); }, TaskCreationOptions.AttachedToParent);
33                 Task.Factory.StartNew(() => { throw new Exception("SECOND TASK EXCEPTION!!!"); }, TaskCreationOptions.AttachedToParent);
34                 Console.WriteLine("Attached both children");
35             });
36         }
37     }
38 }
39
```

C:\WINDOWS\system32\cmd.exe

```
Attached both children

Unhandled Exception: System.AggregateException: One or more errors occurred. ---> System.AggregateException: One or more
errors occurred. ---> System.Exception: SECOND TASK EXCEPTION!!!
   at AggregateException.Program.<>c.<CreateTask>b__3_2() in C:\Users\adafurma\Desktop\msp_windowsinternals\AggregateExce
ption\Program.cs:line 33
   at System.Threading.Tasks.Task.InnerInvoke()
   at System.Threading.Tasks.Task.Execute()
   --- End of inner exception stack trace ---
   at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)
   at System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
   at System.Runtime.CompilerServices.TaskAwaiter.GetResult()
   at AggregateException.Program.<CreateAndAwait>d__2.MoveNext() in C:\Users\adafurma\Desktop\msp_windowsinternals\Aggreg
ateException\Program.cs:line 25
   --- End of inner exception stack trace ---
   at System.Threading.Tasks.Task.ThrowIfExceptional(Boolean includeTaskCanceledExceptions)
   at System.Threading.Tasks.Task.Wait(Int32 millisecondsTimeout, CancellationToken cancellationToken)
   at System.Threading.Tasks.Task.Wait()
   at AggregateException.Program.Main(String[] args) in C:\Users\adafurma\Desktop\msp_windowsinternals\AggregateException
\Program.cs:line 10
Press any key to continue . . .
```

Program.cs

AggregateException

AggregateException.Program

```
1  using System;
2  using System.Threading.Tasks;
3
4  namespace AggregateException
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             //CreateAndAwait().Wait();
11             CreateAndWait().Wait();
12         }
13
14         static Task CreateAndWait()
15         {
16             Task action = CreateTask();
17             Task faulted = action.ContinueWith(p => Console.WriteLine(p.Exception),
18                 TaskContinuationOptions.OnlyOnFaulted);
19             Task succeeded = action.ContinueWith(r => { }, TaskContinuationOptions.OnlyOnRanToCompletion);
20
21             return Task.WhenAll(faulted, succeeded);
22         }
23
24         static async Task CreateAndAwait()...
28
29         static Task CreateTask()
30         {
31             return Task.Factory.StartNew(() =>
32             {
33                 Task.Factory.StartNew(() => { throw new Exception("FIRST TASK EXCEPTION!!!"); },
34                     TaskCreationOptions.AttachedToParent);
35                 Task.Factory.StartNew(() => { throw new Exception("SECOND TASK EXCEPTION!!!"); },
36                     TaskCreationOptions.AttachedToParent);
37                 Console.WriteLine("Attached both children");
38             });
39         }
40     }
41 }
42
```

C:\WINDOWS\system32\cmd.exe

```
Attached both children
System.AggregateException: One or more errors occurred. ---> System.AggregateException: One or more errors occurred. ---
> System.Exception: SECOND TASK EXCEPTION!!!
   at AggregateException.Program.<>c.<CreateTask>b__3_2() in C:\Users\adafurma\Desktop\msp_windowsinternals\AggregateExc
eption\Program.cs:line 33
   at System.Threading.Tasks.Task.InnerInvoke()
   at System.Threading.Tasks.Task.Execute()
   --- End of inner exception stack trace ---
   --- End of inner exception stack trace ---
---> (Inner Exception #0) System.AggregateException: One or more errors occurred. ---> System.Exception: SECOND TASK EXC
EPTION!!!
   at AggregateException.Program.<>c.<CreateTask>b__3_2() in C:\Users\adafurma\Desktop\msp_windowsinternals\AggregateExc
eption\Program.cs:line 33
   at System.Threading.Tasks.Task.InnerInvoke()
   at System.Threading.Tasks.Task.Execute()
   --- End of inner exception stack trace ---
---> (Inner Exception #0) System.Exception: SECOND TASK EXCEPTION!!!
   at AggregateException.Program.<>c.<CreateTask>b__3_2() in C:\Users\adafurma\Desktop\msp_windowsinternals\AggregateExc
eption\Program.cs:line 33
   at System.Threading.Tasks.Task.InnerInvoke()
   at System.Threading.Tasks.Task.Execute()<---
<---

---> (Inner Exception #1) System.AggregateException: One or more errors occurred. ---> System.Exception: FIRST TASK EXCE
PTION!!!
   at AggregateException.Program.<>c.<CreateTask>b__3_1() in C:\Users\adafurma\Desktop\msp_windowsinternals\AggregateExc
eption\Program.cs:line 32
   at System.Threading.Tasks.Task.InnerInvoke()
   at System.Threading.Tasks.Task.Execute()
   --- End of inner exception stack trace ---
---> (Inner Exception #0) System.Exception: FIRST TASK EXCEPTION!!!
   at AggregateException.Program.<>c.<CreateTask>b__3_1() in C:\Users\adafurma\Desktop\msp_windowsinternals\AggregateExc
eption\Program.cs:line 32
   at System.Threading.Tasks.Task.InnerInvoke()
   at System.Threading.Tasks.Task.Execute()<---
<---


Unhandled Exception: System.AggregateException: One or more errors occurred. ---> System.Threading.Tasks.TaskCanceledExc
eption: A task was canceled.
   --- End of inner exception stack trace ---
   at System.Threading.Tasks.Task.ThrowIfExceptional(Boolean includeTaskCanceledExceptions)
   at System.Threading.Tasks.Task.Wait(Int32 millisecondsTimeout, CancellationToken cancellationToken)
   at System.Threading.Tasks.Task.Wait()
   at AggregateException.Program.Main(String[] args) in C:\Users\adafurma\Desktop\msp_windowsinternals\AggregateExceptio
n\Program.cs:line 11
Press any key to continue . . .
```

Exceptions in async

VOID METHOD

One cannot await *void* method (sure?).

Exception is propagated but it is not deterministic.

Use *AsyncContext* from *AsyncEx* library.

TASK METHOD

Exception is stored in the *Task*.

You can also await the method and have the exception propagated.

When chaining in parent-child hierarchy (*TaskCreationOptions.AttachedToParent*) we may miss exceptions, even in *AggregatedException*.

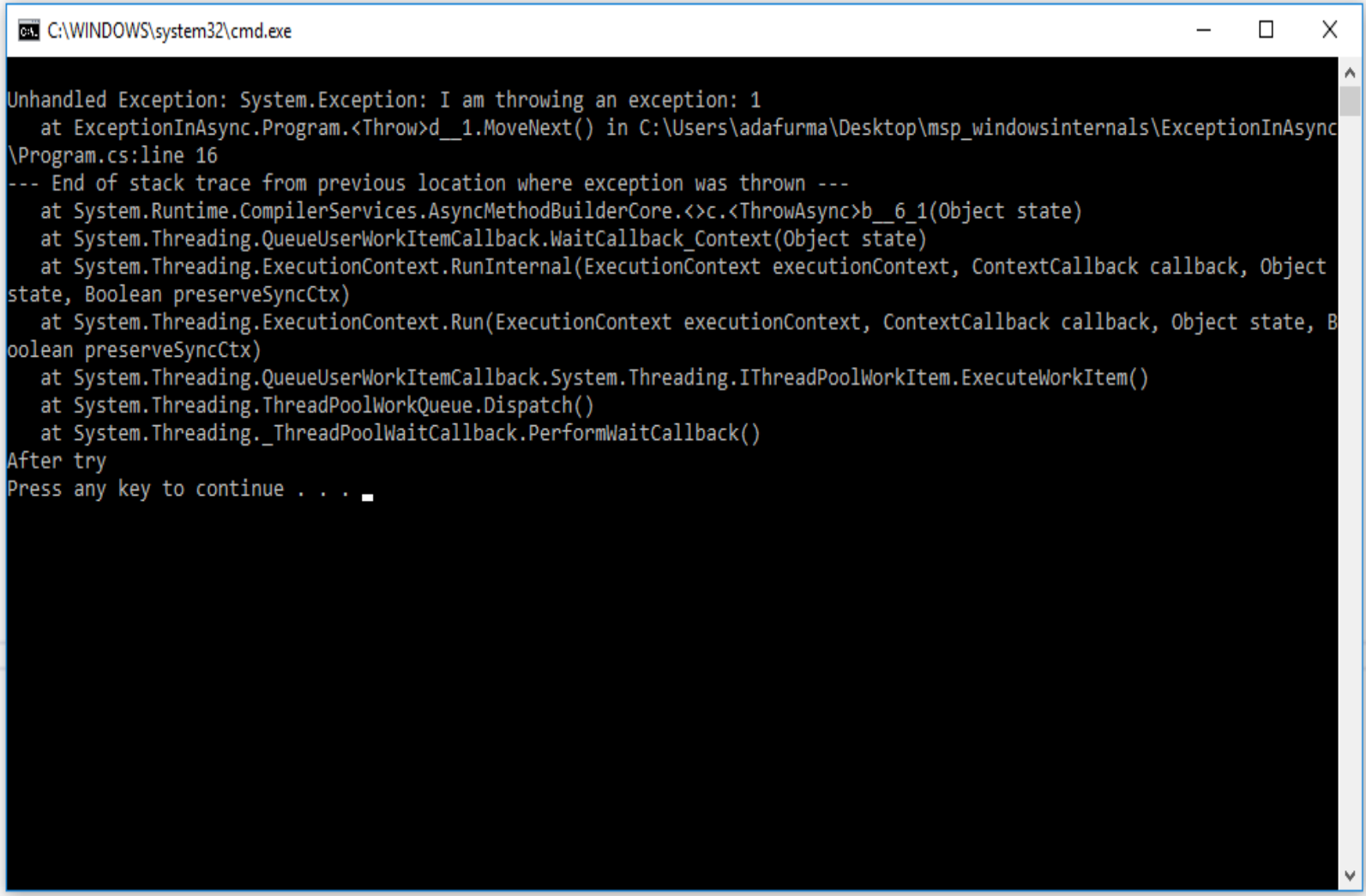
If there is an unobserved exception, it is raised by finalizer thread in *UnobservedTaskException* event where it can be cleared. If not cleared, the process dies (.NET 4) or the exception is suppressed (.NET 4.5).

```
1  using System;
2  using System.Threading;
3  using System.Threading.Tasks;
4
5  namespace ExceptionInAsync
6  {
7
8      class Program
9      {
10         static int Id = 1;
11
12         static async void Throw()
13         //static async Task Throw()
14         {
15             await Task.Delay(300);
16             throw new Exception("I am throwing an exception: " + (Id++));
17         }
18
19         static void Main(string[] args)
20         {
21             // Observe that void method propagates the exception
22             // Task method does not
23
24             try
25             {
26                 Throw();
27                 //Thread.Sleep(600);
28                 Throw();
29             }
30             catch (Exception e)
31             {
32                 Console.WriteLine("Handling " + e);
33             }
34
35             Console.WriteLine("After try");
36             Thread.Sleep(900);
37             Console.WriteLine("After sleep");
38             Console.WriteLine("Done");
39         }
40     }
41 }
42
```

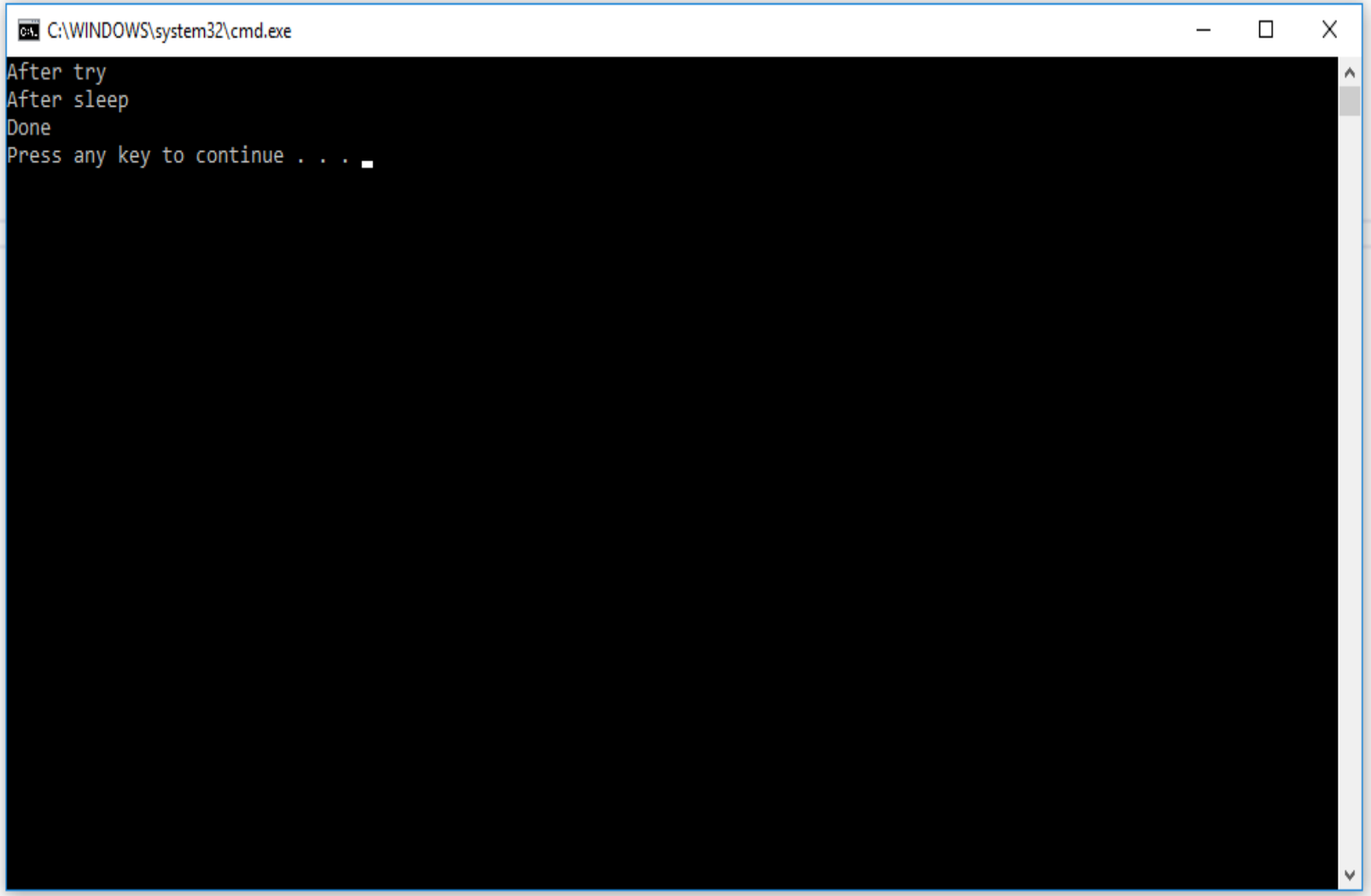
C:\WINDOWS\system32\cmd.exe

```
After try

Unhandled Exception:
Unhandled Exception: System.Exception: I am throwing an exception: 1
   at ExceptionInAsync.Program.<Throw>d__1.MoveNext() in C:\Users\adafurma\Desktop\msp_windowsinternals\ExceptionInAsync
\Program.cs:line 16
--- End of stack trace from previous location where exception was thrown ---
   at System.Runtime.CompilerServices.AsyncMethodBuilderCore.<>c.<ThrowAsync>b__6_1(Object state)
   at System.Threading.QueueUserWorkItemCallback.WaitCallback_Context(Object state)
   at System.Threading.ExecutionContext.RunInternal(ExecutionContext executionContext, ContextCallback callback, Object
state, Boolean preserveSyncCtx)
   at System.Threading.ExecutionContext.Run(ExecutionContext executionContext, ContextCallback callback, Object state, B
oolean preserveSyncCtx)
   at System.Threading.QueueUserWorkItemCallback.System.Threading.IThreadPoolWorkItem.ExecuteWorkItem()
   at System.Threading.ThreadPoolWorkQueue.Dispatch()
   at System.Threading._ThreadPoolWaitCallback.PerformWaitCallback()
System.Exception: I am throwing an exception: 2
   at ExceptionInAsync.Program.<Throw>d__1.MoveNext() in C:\Users\adafurma\Desktop\msp_windowsinternals\ExceptionInAsync
\Program.cs:line 16
--- End of stack trace from previous location where exception was thrown ---
   at System.Runtime.CompilerServices.AsyncMethodBuilderCore.<>c.<ThrowAsync>b__6_1(Object state)
   at System.Threading.QueueUserWorkItemCallback.WaitCallback_Context(Object state)
   at System.Threading.ExecutionContext.RunInternal(ExecutionContext executionContext, ContextCallback callback, Object
state, Boolean preserveSyncCtx)
   at System.Threading.ExecutionContext.Run(ExecutionContext executionContext, ContextCallback callback, Object state, B
oolean preserveSyncCtx)
   at System.Threading.QueueUserWorkItemCallback.System.Threading.IThreadPoolWorkItem.ExecuteWorkItem()
   at System.Threading.ThreadPoolWorkQueue.Dispatch()
   at System.Threading._ThreadPoolWaitCallback.PerformWaitCallback()
After sleep
Done
Press any key to continue . . .
```

```
1  using System;
2  using System.Threading;
3  using System.Threading.Tasks;
4
5  namespace ExceptionInAsync
6  {
7
8      class Program
9      {
10         static int Id = 1;
11
12         static async void Throw()
13         //static async Task Throw()
14         {
15             await Task.Delay(300);
16             throw new Exception("I am throwing an exception: " + (Id++));
17         }
18
19         static void Main(string[] args)
20         {
21             // Observe that void method propagates the exception
22             // Task method does not
23
24             try
25             {
26                 Throw();
27                 Thread.Sleep(600);
28                 Throw();
29             }
30             catch (Exception e)
31             {
32                 Console.WriteLine("Handling " + e);
33             }
34
35             Console.WriteLine("After try");
36             Thread.Sleep(900);
37             Console.WriteLine("After sleep");
38             Console.WriteLine("Done");
39         }
40     }
41 }
42
```



```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace ExceptionInAsync
{

    class Program
    {
        static int Id = 1;

        //static async void Throw()
        static async Task Throw()
        {
            await Task.Delay(300);
            throw new Exception("I am throwing an exception: " + (Id++));
        }

        static void Main(string[] args)
        {
            // Observe that void method propagates the exception
            // Task method does not

            try
            {
                Throw();
                //Thread.Sleep(600);
                Throw();
            }
            catch (Exception e)
            {
                Console.WriteLine("Handling " + e);
            }

            Console.WriteLine("After try");
            Thread.Sleep(900);
            Console.WriteLine("After sleep");
            Console.WriteLine("Done");
        }
    }
}
```



Awaiting *async void*

We cannot do it directly as method returns nothing.

We need to implement custom synchronization context.

To handle exceptions we need to write custom task scheduler.

await async void

Program.cs

AwaitVoid | AwaitVoid.MyContext | Checkpoint()

```
31              }
32          }
33
34          public override void OperationStarted()
35          {
36              _taskCount++;
37          }
38
39          public override void OperationCompleted()
40          {
41              _taskCount--;
42              SignalIfDone();
43          }
44      }
45
46      public static class DelegateHelper
47      {
48          public static Task AwaitAsynchronousHandlers(this Delegate @delegate)
49          {
50              var context = new MyContext();
51              var thread = new Thread(() => {
52                  SynchronizationContext.SetSynchronizationContext(context);
53                  @delegate.DynamicInvoke();
54                  context.Checkpoint();
55              });
56              thread.Start();
57              return context.Waiter;
58          }
59      }
60
61      delegate void Worker();
62
63      public class Program
64      {
65          static event Worker Workers;
66
67          public static void Main()
68          {
69              Workers += async () => await Task.Delay(100).ContinueWith(t => Console.WriteLine(100));
70              Workers += async () => await Task.Delay(1500).ContinueWith(t => Console.WriteLine(1500));
71              Workers += () => Console.WriteLine("No delay");
72              Workers.AwaitAsynchronousHandlers().Wait();
73          }
74      }
75  }
76
```

C:\WINDOWS\system32\cmd.exe

```
No delay
100
1500
Press any key to continue . . .
```

Solution Exp

Search Solut

Zad

Catch exceptions in *async void*

Program.cs

CatchAsyncVoid | CatchAsyncVoid.MyContext | Run(Action action)

```
82      public static Task Run(Action action)
83      {
84          return Task.Run(() =>
85          {
86              var oldContext = SynchronizationContext.Current;
87              var newContext = new MyContext();
88              try
89              {
90                  SynchronizationContext.SetSynchronizationContext(newContext);
91                  var spanningTask = newContext.factory.StartNew(action);
92                  foreach (var task in newContext.scheduler.tasks.GetConsumingEnumerable())
93                  {
94                      newContext.scheduler.TryExecuteTask(task);
95                      task.GetAwaiter().GetResult();
96                  }
97                  spanningTask.GetAwaiter().GetResult();
98              }
99              finally
100             {
101                 SynchronizationContext.SetSynchronizationContext(oldContext);
102             }
103         });
104     }
105 }
106
107 class Program
108 {
109     static void Main(string[] args)
110     {
111         Console.WriteLine("Preparing job to run");
112         var task = MyContext.Run(() => Throw());
113         Console.WriteLine("Job is scheduled, will run any second. Sleeping main thread");
114         Thread.Sleep(5000);
115         Console.WriteLine("Catching exception");
116         try
117         {
118             task.GetAwaiter().GetResult(); // Using Wait() here (or in lines 95, 97) instead would return AggregateException instead of original one
119         }
120         catch (Exception e)
121         {
122             Console.WriteLine("Swallowing exception " + e.GetType() + "\n" + e);
123         }
124
125         Thread.Sleep(1000);
126         Console.WriteLine("Done");
127     }
128
```

C:\WINDOWS\system32\cmd.exe

```
Preparing job to run
Job is scheduled, will run any second. Sleeping main thread
        Waiting in async void
        Throwing in async void
Catching exception
Swallowing exception System.Exception
System.Exception: Hahaha from async void
   at CatchAsyncVoid.Program.<Throw>d__1.MoveNext() in C:\Users\adafurma\Desktop\msp_windowsinternals\CatchAsyncVoid\Pro
gram.cs:line 134
--- End of stack trace from previous location where exception was thrown ---
   at System.Runtime.CompilerServices.AsyncMethodBuilderCore.<>c.<ThrowAsync>b__6_0(Object state)
   at CatchAsyncVoid.MyContext.<>c__DisplayClass4_0.<Post>b__0() in C:\Users\adafurma\Desktop\msp_windowsinternals\Catch
AsyncVoid\Program.cs:line 59
   at System.Threading.Tasks.Task.InnerInvoke()
   at System.Threading.Tasks.Task.Execute()
--- End of stack trace from previous location where exception was thrown ---
   at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)
   at System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
   at System.Runtime.CompilerServices.TaskAwaiter.GetResult()
   at CatchAsyncVoid.MyContext.<>c__DisplayClass8_0.<Run>b__0() in C:\Users\adafurma\Desktop\msp_windowsinternals\CatchA
syncVoid\Program.cs:line 95
   at System.Threading.Tasks.Task.InnerInvoke()
   at System.Threading.Tasks.Task.Execute()
--- End of stack trace from previous location where exception was thrown ---
   at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)
   at System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
   at System.Runtime.CompilerServices.TaskAwaiter.GetResult()
   at CatchAsyncVoid.Program.Main(String[] args) in C:\Users\adafurma\Desktop\msp_windowsinternals\CatchAsyncVoid\Progra
m.cs:line 118
Done
Press any key to continue . . .
```

Solution Explorer

Wątki

Zadania

Async Constructor Pattern

```
1  public class Program
2  {
3      async public Program(){
4      }
5  }
```

Compilation error (line 3, col 15):
The modifier 'async' is not valid for this item

```
public class Program
{
    public static async Task<Program> CreateAsync()
    {
        Program x = new Program();
        await x.InitializeAsync();
        return x;
    }
    private Program(){}

    private async Task InitializeAsync()
    {
    }
}
```

Lock in async

```
1  using System;
2  using System.Threading.Tasks;
3  using System.Threading;
4
5  public class Program
6  {
7      public static void Main()
8      {
9          Test().Wait();
10     }
11
12     public static async Task Test(){
13         var o = new object();
14         lock(o){
15             Console.WriteLine(Thread.CurrentThread.ManagedThreadId);
16             await Task.Delay(100);
17             Console.WriteLine(Thread.CurrentThread.ManagedThreadId);
18         }
19     }
20 }
21
```

```
Compilation error (line 16, col 4): The 'await' operator cannot be used in the body of a lock statement
```

```
1  using System;
2  using System.Threading.Tasks;
3  using System.Threading;
4
5  public class Program
6  {
7      public static void Main()
8      {
9          Test().Wait();
10     }
11
12     public static async Task Test(){
13         var o = new object();
14         bool taken = false;
15         try{
16             Monitor.Enter(o, ref taken);
17             Console.WriteLine(Thread.CurrentThread.ManagedThreadId);
18             await Task.Delay(100);
19             Console.WriteLine(Thread.CurrentThread.ManagedThreadId);
20         }finally{
21             if(taken){
22                 Monitor.Exit(o);
23             }
```

```
1
4
Unhandled exception. System.AggregateException: One or more errors occurred. (Object synchronization method was called from an unsynchronized block of code.)
 ---> System.Threading.SynchronizationLockException: Object synchronization method was called from an unsynchronized block of code.
   at System.Threading.Monitor.Exit(Object obj)
   at Program.Test()
   --- End of inner exception stack trace ---
   at System.Threading.Tasks.Task.Wait(Int32 millisecondsTimeout, CancellationToken cancellationToken)
   at System.Threading.Tasks.Task.Wait()
   at Program.Main()
Command terminated by signal 6
```

It breaks decent programming principles

Never wait without a timeout!

FIRST PRINCIPLE OF MULTITHREADING

How do we await with async?

```
static async Task Main(string[] args)
{
    await Do();
}

public static async Task Do()
{
    await Hang(); // How to timeout here?
}

public static async Task Hang()
{
    await Task.Delay(TimeSpan.FromDays(1));
    Console.WriteLine("Worked!");
}
```

Manually

```
public static class TimeoutManually
{
    1 reference
    public static async Task Do()
    {
        var completed = await Task.WhenAny(Program.Hang(), Program.ThrowTimeoutException<bool>()).ConfigureAwait(false);
        await completed.ConfigureAwait(false);
    }
}
```

With extension

```
public static class TimeoutWithExtension
{

    public static async Task Do()
    {
        await Program.Hang().Timeout().ConfigureAwait(false);
    }
}


static class TaskExtensions
{

    public static async Task Timeout(this Task t)
    {
        await (await Task.WhenAny(t, Program.ThrowTimeoutException<bool>()).ConfigureAwait(false)).ConfigureAwait(false);
    }

    0 references
    public static async Task<T> Timeout<T>(this Task<T> t)
    {
        return await (await Task.WhenAny(t, Program.ThrowTimeoutException<T>()).ConfigureAwait(false)).ConfigureAwait(false);
    }
}
```

With synchronization context

```
public static Task Run(Action action)
{
    TaskCompletionSource<bool> taskCompletionSource = new TaskCompletionSource<bool>(TaskContinuationOptions.RunContinuationsAsynchronously);
    Program.ThrowTimeoutException<bool>().ContinueWith(t => taskCompletionSource.SetException(t.Exception));

    Task.Run(() =>
    {
        var oldContext = SynchronizationContext.Current;
        var newContext = new MyContext();
        try
        {
            SynchronizationContext.SetSynchronizationContext(newContext);
            var spanningTask = newContext.factory.StartNew(action);
            foreach (var task in newContext.scheduler.tasks.GetConsumingEnumerable())
            {
                newContext.scheduler.TryExecuteTask(task);
                task.GetAwaiter().GetResult();
            }
            spanningTask.GetAwaiter().GetResult();
        }
        finally
        {
            SynchronizationContext.SetSynchronizationContext(oldContext);
        }
    }).ContinueWith(t => taskCompletionSource.SetException(t.Exception), TaskContinuationOptions.OnlyOnFaulted).ContinueWith(_ => taskCompletionSource.SetResult(true));

    return taskCompletionSource.Task;
}
```

With custom task type

```
public static class TimeoutWithCustomTask
{
    0 references
    public static async TimeoutableTask Do()
    {
        await Program.Hang().ConfigureAwait(false);
    }
}

[AsyncMethodBuilder(typeof(TimeoutableTaskMethodBuilder))]
4 references
public class TimeoutableTask
{
    5 references
    public TaskCompletionSource<object> Promise { get; } = new TaskCompletionSource<object>();

    1 reference
    public Task AsTask() => Promise.Task;

    1 reference
    public TaskAwaiter<object> GetAwaiter()
    {
        return Promise.Task.GetAwaiter();
    }

    public static implicit operator Task(TimeoutableTask task) => task.AsTask();
}
```

Do not repeat yourself (DRY)

```
private static string InternalReadAllText(string path, Encoding encoding)
{
    Debug.Assert(path != null);
    Debug.Assert(encoding != null);
    Debug.Assert(path.Length > 0);

    using (StreamReader sr = new StreamReader(path, encoding, detectEncodingFromByteOrderMarks: true))
        return sr.ReadToEnd();
}
```

```
private static async Task InternalReadAllTextAsync(string path, Encoding encoding, CancellationToken cancellationToken)
{
    Debug.Assert(!string.IsNullOrEmpty(path));
    Debug.Assert(encoding != null);

    char[] buffer = null;
    StreamReader sr = AsyncStreamReader(path, encoding);
    try
    {
        cancellationToken.ThrowIfCancellationRequested();
        buffer = ArrayPool.Shared.Rent(sr.CurrentEncoding.GetMaxCharCount(DefaultBufferSize));
        StringBuilder sb = new StringBuilder();
        while (true)
        {
#if MS_IO_REDIST
            int read = await sr.ReadAsync(buffer, 0, buffer.Length).ConfigureAwait(false);
#else
            int read = await sr.ReadAsync(new Memory(buffer), cancellationToken).ConfigureAwait(false);
#endif
            if (read == 0)
            {
                return sb.ToString();
            }

            sb.Append(buffer, 0, read);
        }
    }
    finally
    {
        sr.Dispose();
        if (buffer != null)
        {
            ArrayPool.Shared.Return(buffer);
        }
    }
}
```

Dependency Inversion

HOW DO YOU REPLACE A STRING?

String improvements in Java

Internal structure:

- Originally String was implemented using char array under the hood.
- Java 9 changed it to byte array to allocate 1 byte if string has no unicode characters.

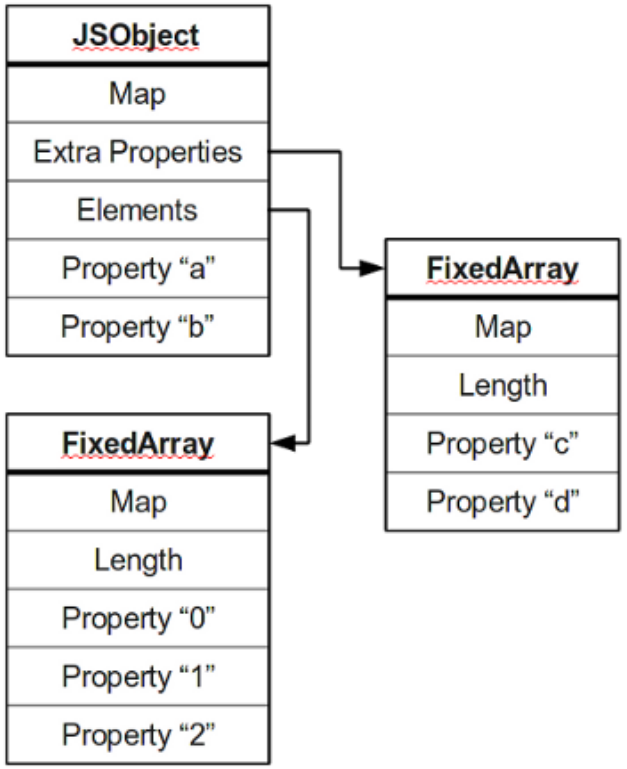
Concatenation performance:

- Before Java 9 concatenations were translated to StringBuilder.append.
- Starting in Java 9 they are translated to invokedynamic and reuse multiple strategies.

String at Facebook – 1% performance win

CppCon 2016: Nicholas Ormrod “The strange details of std::string at Facebook"

Objects in V8



Object is a dictionary — use hash map.

Use maps to optimize access by offset.

Reserve more memory than needed to have room for new properties.

Property names are strings but for arrays we can use… well, arrays.

V8 switches implementation depending on the usage.

How do you replace a string?

You cannot!

String is:

- A class – not an interface
- A sealed class – no inheritance
- Exposed on IL level via string literals
- Highly coupled with managed and unmanaged code (native part relies on memory structure)

Colorful functions

	Green	Red
Green		
Red		

Colorful functions

	Green	Red
Green		
Red		

	void Foo()	async Task Foo()
void Foo()		
async Task Foo()		

Colorful functions

	Green	Red
Green		
Red		

	void Foo()	async Task Foo()
void Foo()		
async Task Foo()		

	void Foo()	async ValueTask Foo()	async Task Foo()
void Foo()			
async ValueTask Foo()			
async Task Foo()			

Let's make it better!

Asynchronous code **does not block** the operating system level thread.

async in C#

async in C# is implemented as:

- coroutine compiler level transformation with
- service locator for promise orchestration and
- statically bound promise factories

Coroutines

COROUTINE COMPILER LEVEL TRANSFORMATION

Fibers

DO NOT BLOCK THE OPERATING SYSTEM LEVEL THREAD

Monads

STATICALLY BOUND PROMISE FACTORIES

Generics

SERVICE LOCATOR FOR PROMISE ORCHESTRATION

But why?


Tweet
David Fowler @davidfowl · 4 cze
Today we had a meeting about experimenting with a green threads implementation in .NET. More info to come soon, exciting times! #dotnet
48
106
729
Adam Furmanek
@furmanekadam
W odpowiedzi do @thylux @shanselman i@davidfowl
Thanks I have a post about async on Fibers blog.adamfurmanek.pl/2020/10/03/asy... and a talk blog.adamfurmanek.pl/videos-talks-p...
Loom in JVM does that, so it's definitely worth a shot.
Przetłumacz Tweeta
1:56 PM · 4 cze 2022 · Twitter Web App

Key Challenges

Green threads introduce a completely new async programming model. The interaction between green threads and the existing async model is quite complex for .NET developers. For example, invoking async methods from green thread code requires a sync-over-async code pattern that is a very poor choice if the code is executed on a regular thread.

Interop with native code in green threads model is complex and comparatively slow . With a benchmark of a minimal P/Invoke, the cost of making 100,000,000 P/Invoke calls changed from 300ms to about 1800ms when running on a green thread. This was expected as similar issues impact other languages implementing green threads. We found that there are surprising functional issues in interactions with code which uses thread-local static variables or exposes native thread state.
Interactions with security mitigations such as shadow stacks intended to protect against return-oriented programming would be quite challenging.

It is possible or even likely that we could make the green threads model (a bit) faster than async in important scenarios. The key challenge is that this capability would come with a cost of it being significantly slower in other scenarios and having to give up compatibility and other characteristics.

It is less clear that we could make green threads faster than async if we put significant effort into improving async.

Conclusions and next steps

We have chosen to place the green threads experiment on hold and instead keep improving the existing (async/await) model for developing asynchronous code in .NET. This decision is primarily due to concerns about introducing a new programming model. We can likely provide more value to our users by improving the async model we already have. We will continue to monitor industry trends in this field.

Summary

Know your synchronization context — and don’t abuse it!

Do not use *async void* methods if you don’t have to.

Have *async* all the way up.

Don’t wait for asynchronous methods in synchronous code if you don’t have to.

Avoid creating threads if you can.

Always await tasks, handle all the exceptions.

Always add handlers to unobserved exceptions and unhandled exceptions.

Q&A

References

Jeffrey Richter - „CLR via C#”

Jeffrey Richter, Christophe Nasarre - „Windows via C/C++”

Mark Russinovich, David A. Solomon, Alex Ionescu - „Windows Internals”

Penny Orwick – „Developing drivers with the Microsoft Windows Driver Foundation”

Mario Hewardt, Daniel Pravat - „Advanced Windows Debugging”

Mario Hewardt - „Advanced .NET Debugging”

Steven Pratschner - „Customizing the Microsoft .NET Framework Common Language Runtime”

Serge Lidin - „Expert .NET 2.0 IL Assembler”

Joel Pobar, Ted Neward — „Shared Source CLI 2.0 Internals”

Adam Furmanek – „.NET Internals Cookbook”

https://github.com/dotnet/coreclr/blob/master/Documentation/botr/README.md — „Book of the Runtime”

https://blogs.msdn.microsoft.com/oldnewthing/ — Raymond Chen „The Old New Thing”

References

https://blog.adamfurmanek.pl/blog/2016/10/08/async-wandering-part-1/ — async in unit tests

https://blog.adamfurmanek.pl/blog/2017/01/07/async-wandering-part-3/ — WinForms

https://blog.adamfurmanek.pl/blog/2017/06/03/capturing-thread-creation-to-catch-exceptions/ — overriding Thread constructor to handle exceptions

https://blog.adamfurmanek.pl/blog/2017/01/14/async-wandering-part-4-awaiting-for-void-methods/ — awaiting *async void*

https://blog.adamfurmanek.pl/blog/2018/10/06/async-wandering-part-5-catching-exceptions-from-async-void/ — catching exceptions in *async void*

References

https://www.codeproject.com/Articles/662735/Internals-of-Windows-Thread - Windows threads

http://aviadezra.blogspot.com/2009/06/net-clr-thread-pool-work.html - .NET ThreadPool

https://mattwarren.org/2017/04/13/The-CLR-Thread-Pool-Thread-Injection-Algorithm/ — ThreadPool injection algorithm

http://www.microsoft.com/download/en/details.aspx?id=19957 — TAP

https://msdn.microsoft.com/en-us/magazine/gg598924.aspx?f=255&MSPPError=-2147217396 – It's all about the synchronization context

https://blogs.msdn.microsoft.com/seteplia/2018/10/01/the-danger-of-taskcompletionsourcet-class/ - TaskCompletionSource

https://blog.stephencleary.com/2014/04/a-tour-of-task-part-0-overview.html - Task internals

https://blog.stephencleary.com/2017/03/aspnetcore-synchronization-context.html — ASP.NET Core sychronization context

https://blogs.msdn.microsoft.com/pfxteam/2012/06/15/executioncontext-vs-synchronizationcontext/ — ExecutionContext internals

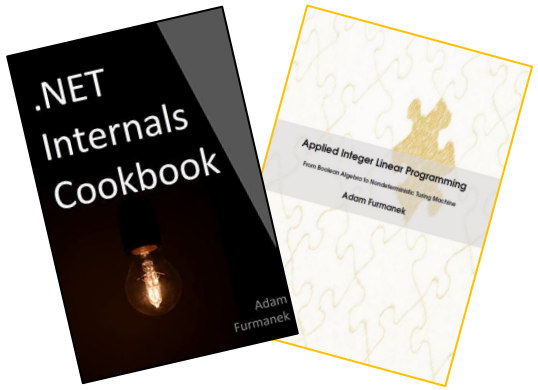
https://weblogs.asp.net/dixin/understanding-c-sharp-async-await-3-runtime-context — ExecutionContext internals

https://blogs.msdn.microsoft.com/seteplia/2017/11/30/dissecting-the-async-methods-in-c/ - State machine

https://weblogs.asp.net/dixin/understanding-c-sharp-async-await-1-compilation - State machine

https://github.com/dotnet/runtimelab/issues/2398 - .NET Green Thread experimentations

https://github.com/dotnet/runtimelab/blob/feature/green-threads/docs/design/features/greenthreads.md - .NET Green Thread Reports



Random IT Utensils

IT, operating systems, maths, and more.

Thanks!

CONTACT@ADAMFURMANEK.PL

HTTP://BLOG.ADAMFURMANEK.PL

FURMANEKADAM