Throughout the entire process, a key takeaway is that **no thread is dedicated to running the task**.

HTTPS://DOCS.MICROSOFT.COM/EN-US/DOTNET/STANDARD/ASYNC-IN-DEPTH

Internals of Async

CONTACT@ADAMFURMANEK.PL

HTTP://BLOG.ADAMFURMANEK.PL

FURMANEKADAM

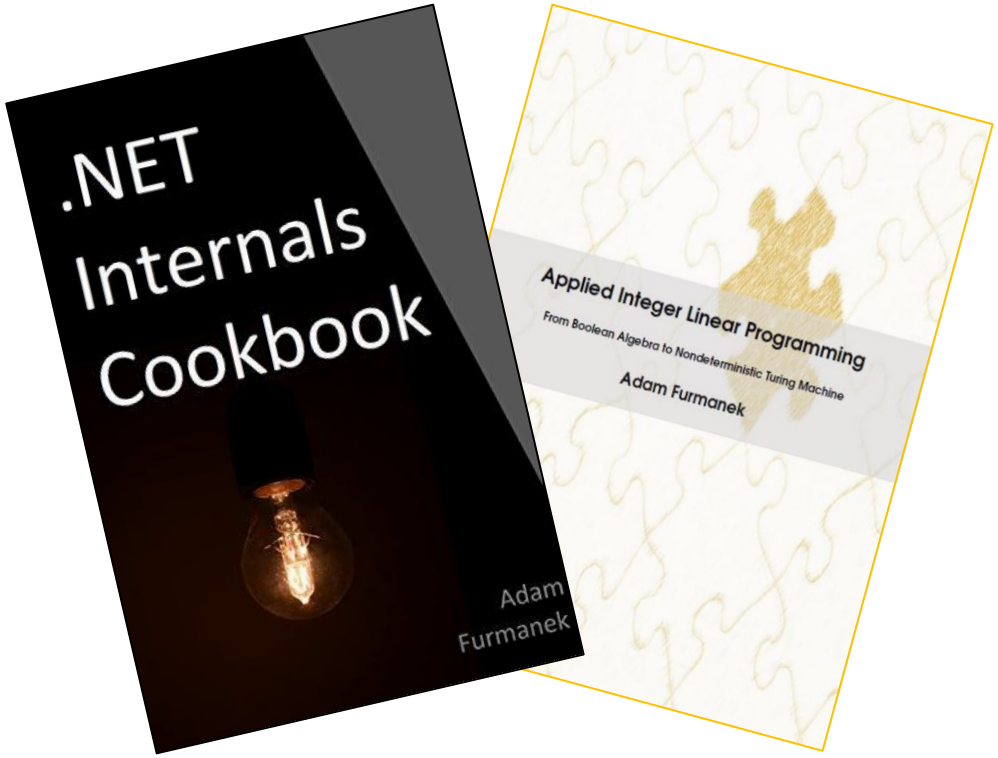
About me

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

Primitves under the hood.

Task detail.

SynchronizationContext internals.

State machine.

Waiting for async void and handling exceptions

Primitives under the hood

Native thread

Two types: foreground and background (don’t stop application from terminating).

Consists of *Thread Kernel Object*, two stacks (user mode and kernel mode) and *Thread Environment Block* (TEB).

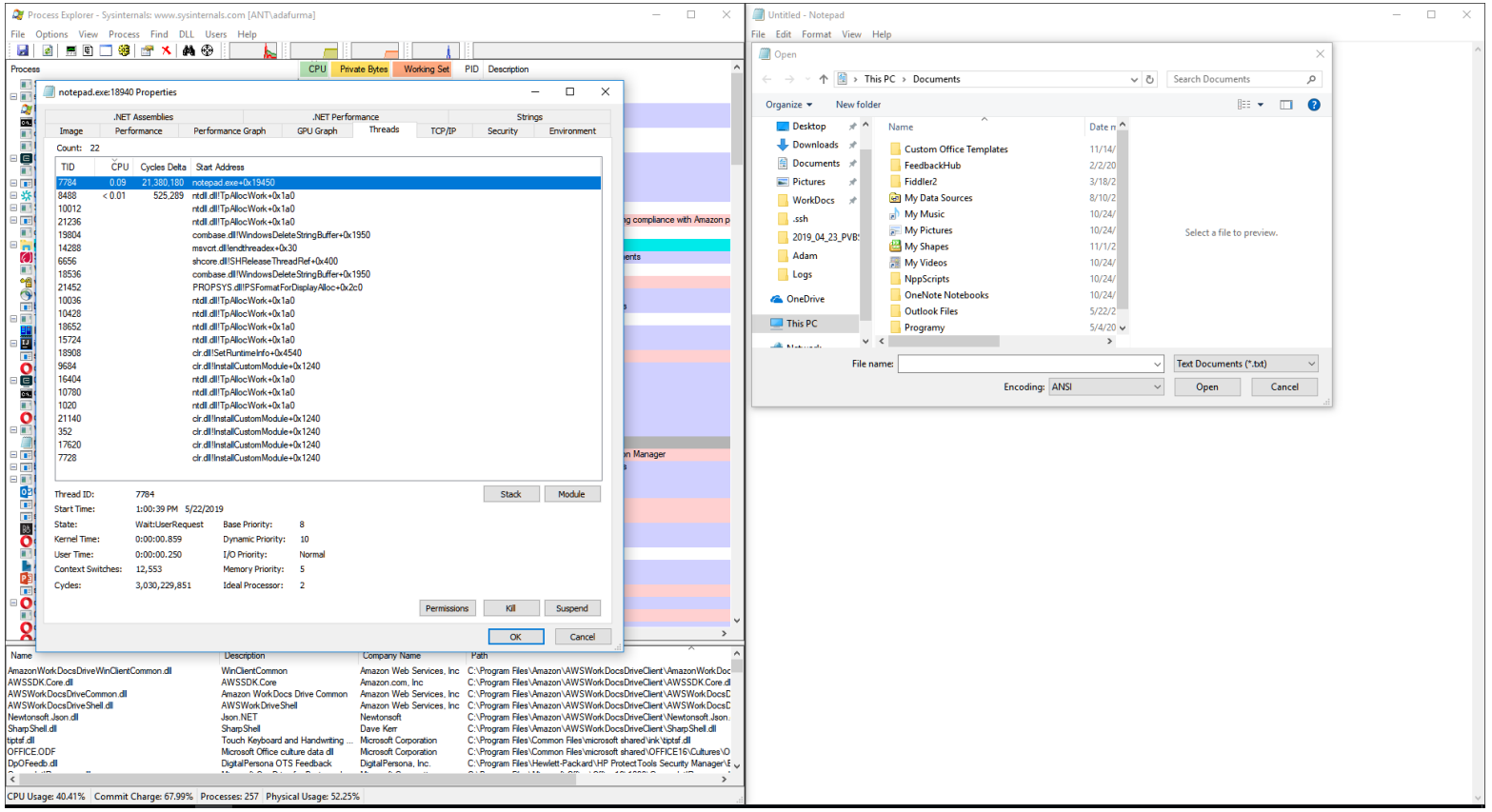
User mode stack by default has 1 MB, kernel mode has 12/24 KB.

Has impersonation info, security context, *Thread Local Storage* (TLS).

Windows schedules threads, not processes!

How many threads does the *notepad.exe* have?

How many threads does a notepad have?



Managed thread

Has ID independent of native thread ID.

Can have name.

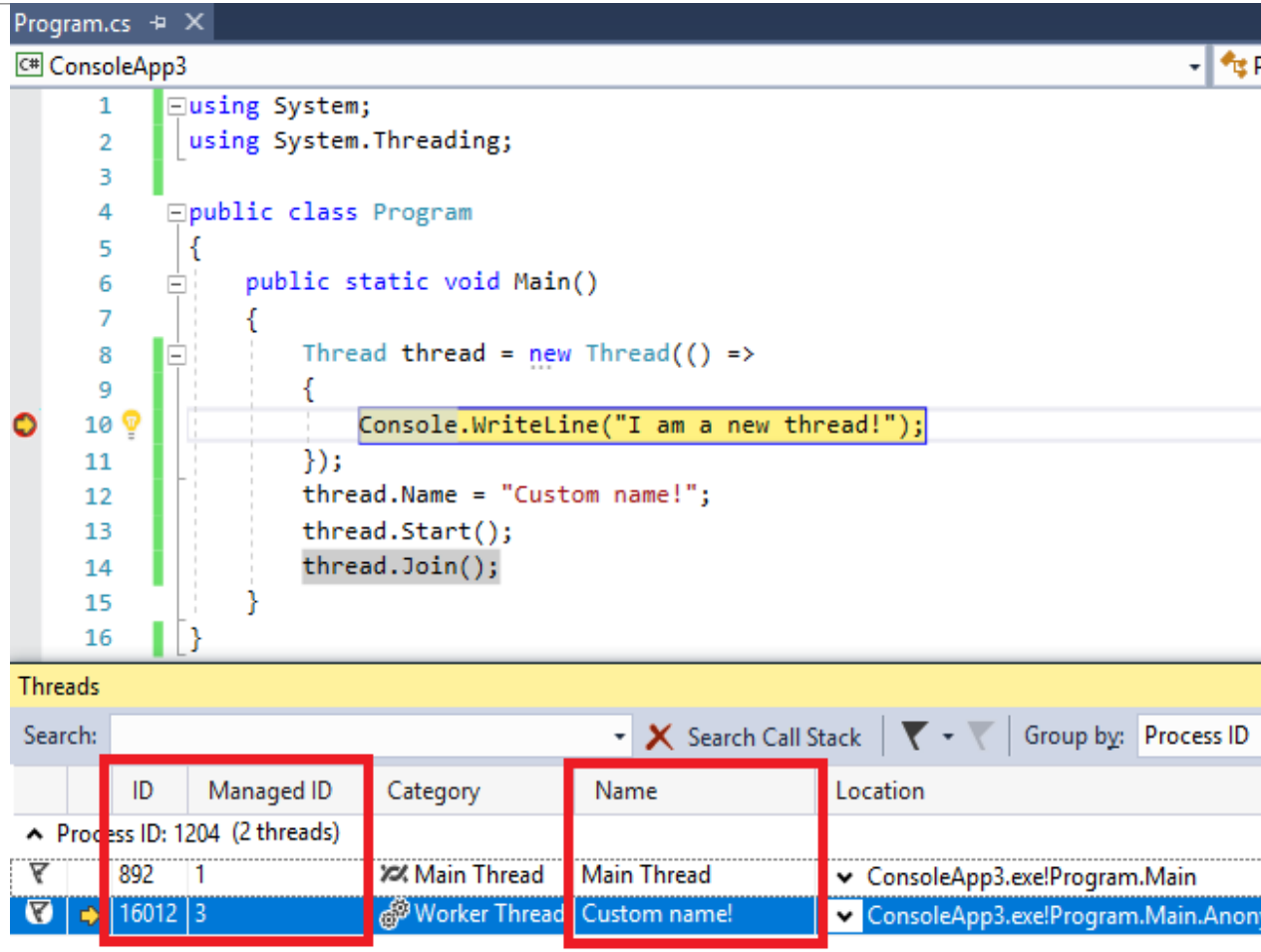
Can be suspended but this should not be done! Can be aborted by *Thread.Abort* but this doesn't guarantee anything.

Precommits stack when created.

Unhandled exception kills the application in most cases.

In .NET 1 it was different:

- Exception in other thread was printed to the console and the thread was terminated.
- Exception on the finalizer was printed to the console and finalizer was still working.
- Exception on the main thread resulted in application termination.



ThreadPool

Different from Win32 thread pool. Used by tasks, asynchronous timers, wait handles and *ThreadPool.QueueUserWorkItem*.

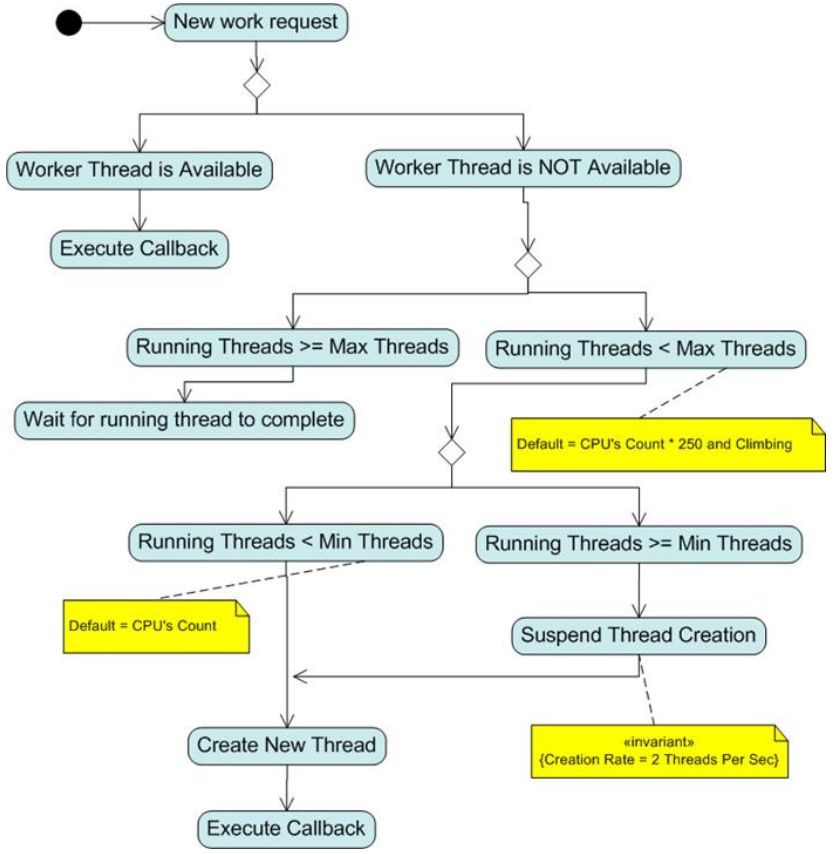
Static class – you cannot just create your own thread pool.

Threads work in background, do not clear the TLS, have default stack size and default priority.

One pool per process, its size depends on the size of the virtual address space. Threads are created and destroyed as needed using hill climbing algorithm.

Two types of threads: for ordinary callbacks and for I/O operations.

Thrown exception is held until awaiting and then propagated if possible (thrown out of band for *async void*). In .NET 1 it was different - exception on a thread pool was printed to the console and the thread was returned to the pool.



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

ThreadPool implementation

```
public static class SimpleThreadPool
{
    private static BlockingCollection<Action> _work = new BlockingCollection<Action>();
    static SimpleThreadPool() {
        for (int i = 0; i < Environment.ProcessorCount; i++) {
            new Thread(() => {
              foreach (var action in _work.GetConsumingEnumerable()) {
                  action();
              }
            }) { IsBackground = true }.Start();
         }
     }
     public static void QueueWorkItem(Action workItem) { _work.Add(workItem); }
}
```

Asynchronous Programming Model (APM)

BeginOperation returns an object implementing *IAsyncResult*.

- Triggers the asynchronous calculations on different thread.
- Can also accept a callback to be called when the operation is finished.

IAsyncResult:

- Has some *AsyncState*.
- Contains *WaitHandle* which we can use to block the application.
- Has flag indicating whether the operation is completed.

EndOperation accepts *IAsyncResult* as a parameter and returns the same as synchronous counterpart.

- Throws all exceptions if needed.
- If the operation hasn't finished, blocks the thread.

```
var fs = new FileStream(@"C:\file.txt");
byte[] data = new byte[100];
fs.BeginRead(data, 0, data.Length,
(IAsyncResult ar) =>
    {
        int bytesRead = fs.EndRead(ar);
        fs.Close();
    }, null
);
```

Event-based Asynchronous Pattern (EAP)

MethodNameAsync.

- Triggers the operation on a separate thread.

MethodNameCompleted.

- Event fired when the operation finishes.
- Passes parameter *AsyncCompletedEventArgs*.

AsyncCompletedEventArgs:

- Contains flag if the job was cancelled.
- Contains all the errors.
- Has some *UserState*.

Can be canceled.

Can be used easily with *BackgroundWorker*.

```
backgroundWorker.DoWork += backgroundWorker_DoWork;

private void backgroundWorker_DoWork(object sender,
DoWorkEventArgs e)

{

    // ...

}

private void
backgroundWorker_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)

{

    // ...

}
```

Task-based Asynchronous Pattern (TAP)

Task.Run accepting delegate triggers the job:

- Equivalent to *Task.Factory.StartNew(job, CancellationToken.None, TaskCreationOptions.DenyChildAttach, TaskScheduler.Default);*
- Unwraps the result if needed (so we get *Task<int>* instead of *Task<Task<int>>*).

Task can be created manually via constructor and schedulled using *Start* method.

Can be joined by using *ContinueWith*.

Exceptions are caught and propagated on continuation.

Can be used with *TaskCompletionSource*.

Can be cancelled with *CancellationToken*.

Can report progress using *IProgress<T>.*

Parallel Language Integrated Queries (PLINQ)

Created when *AsParallel* called on *IEnumerable*. Can be reverted by *AsSequential*.

Operations defined in *ParallelEnumerable* class.

Can be ordered by calling *AsOrdered*.

Task merging can be configured by specifying *ParallelMergeOptions*.

Maximum number of concurrent tasks can be controlled using *WithDegreeOfParallelism*.

Parallelism is not mandatory! Can be forced with *ParallelExcecutionMode*.

Each *AsParallel* call reshuffles the tasks.

async and *await*

await can be executed on anything awaitable – not necessarily a *Task*!

- Task.Yield returns *YieldAwaitable*

Duck typing - awaitable type must be able to return *GetAwaiter()* with the following:

- Implements *INotifyCompletion* interface
- `bool IsCompleted { get; }`
- `void OnCompleted(Action continuation);`
- `TResult GetResult(); // or void`

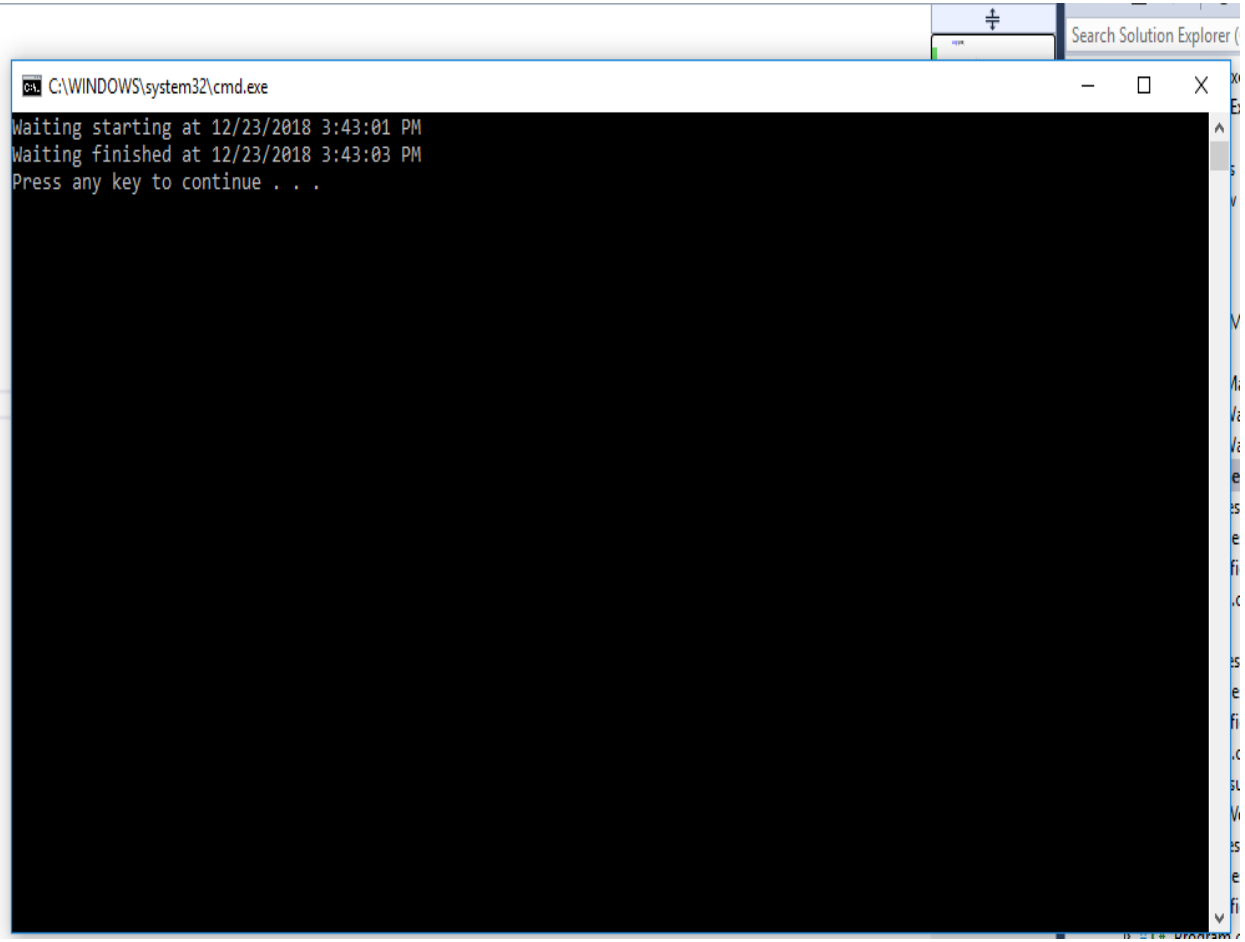
async means nothing — it only instructs the compiler to create a state machine.

We can make any type awaitable using extension methods!

Very similar to *foreach*.

Awaiting on integer

```
1  using System;
2  using System.Runtime.CompilerServices;
3  using System.Threading.Tasks;
4
5  namespace AwaitOnInteger
6  {
7      class Program
8      {
9          static void Main(string[] args)
10         {
11             WaitForInt().Wait();
12         }
13
14         static async Task WaitForInt()
15         {
16             Console.WriteLine($"Waiting starting at {DateTime.Now}");
17             await 2000;
18             Console.WriteLine($"Waiting finished at {DateTime.Now}");
19
20         }
21     }
22
23     public static class AwaitableInt
24     {
25         public static TaskAwaiter GetAwaiter(this int miliseconds)
26         {
27             return Task.Delay(TimeSpan.FromMilliseconds(miliseconds)).GetAwaiter();
28         }
29     }
30 }
31
```



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

Task details

STATICALLY BOUND PROMISE FACTORIES

Two types of tasks

DELEGATE TASK – CPU-BOUND

Has some code to run.

Mostly in *TPL* world.

Created by *TaskFactory* or by constructor.

Used in *PLINQ*.

Can be scheduled and executed.

PROMISE TASK – I/O-BOUND

Signals completion of something.

Mostly in *async* world.

Task.FromResult

- Creates completed *Task* with result.

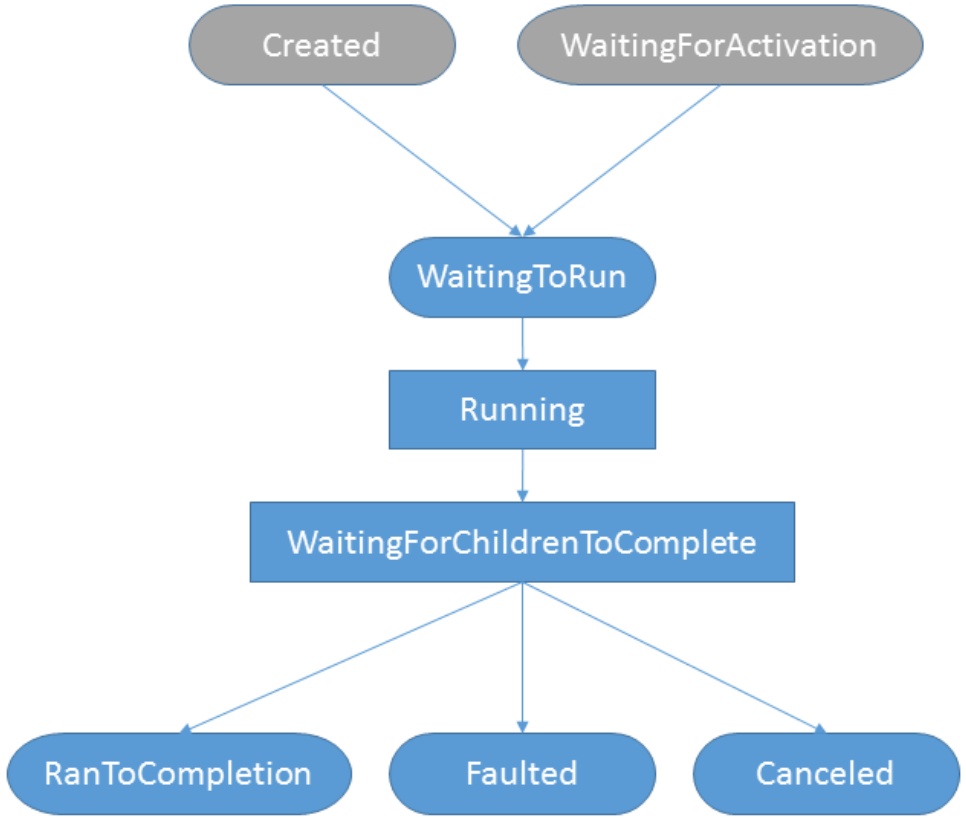
Task.Delay

- Equivalent of *Thread.Sleep.*

Task.Yield

- Returns *YieldAwaitable.*
- Schedules continuation immediately

Task state



https://blog.stephencleary.com/2014/06/a-tour-of-task-part-3-status.html

Task creation

CONSTRUCTOR

Do not use!

Creates only delegate T*ask*.

Created *Task* is not scheduled so will not start running unless asked to.

Created *Task* can be started by calling *Start* method (and optionally providing a scheduler).

FACTORY

Task.Run()

Task.Factory.StartNew()

PLINQ

Task.ScheduleAndStart

```
// Token: 0x06003F22 RID: 16162 RVA: 0x000EAA2C File Offset: 0x000E8C2C
[SecuritySafeCritical]
internal void ScheduleAndStart(bool needsProtection)
{
    if (needsProtection)
    {
        if (!this.MarkStarted())
        {
            return;
        }
    }
    else
    {
        this.m_stateFlags |= 65536;
    }
    if (Task.s_asyncDebuggingEnabled)
    {
        Task.AddToActiveTasks(this);
    }
    if (AsyncCausalityTracer.LoggingOn && (this.Options & (TaskCreationOptions)512) == TaskCreationOptions.None)
    {
        AsyncCausalityTracer.TraceOperationCreation(CausalityTraceLevel.Required, this.Id, "Task: " + ((Delegate)this.m_action).Method.Name, 0UL);
    }
    try
    {
        this.m_taskScheduler.InternalQueueTask(this);
    }
    catch (ThreadAbortException exceptionObject)
    {
        this.AddException(exceptionObject);
        this.FinishThreadAbortedTask(true, false);
    }
    catch (Exception innerException)
    {
        TaskSchedulerException ex = new TaskSchedulerException(innerException);
        this.AddException(ex);
        this.Finish(false);
        if ((this.Options & (TaskCreationOptions)512) == TaskCreationOptions.None)
        {
            this.m_contingentProperties.m_exceptionsHolder.MarkAsHandled(false);
        }
        throw ex;
    }
}
```

TaskScheduler

Schedules tasks on the threads – it makes sure that the work of a task is eventually executed.

For *TPL* and *PLINQ* is based on the thread pool.

Supports work-stealing, thread injection/retirement and fairness.

Two types of queues:

- Global – for top level tasks
- Local – for nested/child tasks, accessed in LIFO order

Long running tasks are handled separately, do not go via global/local queue.

We can implement our own schedulers.

TaskScheduler implementation

```
public class MyTaskScheduler : TaskScheduler
{
    private readonly MyContext context;
    public BlockingCollection<Task> tasks = new BlockingCollection<Task>();

    protected override IEnumerable<Task> GetScheduledTasks()
    {
        return tasks;
    }

    protected override void QueueTask(Task task)
    {
        tasks.Add(task);
    }

    protected override bool TryExecuteTaskInline(Task task, bool taskWasPreviouslyQueued)
    {
        return TryExecuteTask(task);
    }
}
```

Task.ContinueWith

Creates a continuation that executes asynchronously when the target task complets

- Can specify *CancellationToken*
- Can specify *TaskScheduler*
- Can specify *TaskContinuationOptions*

Options:

- OnlyOnCompletion, OnlyOnCanceled, OnlyOnFaulted, NotOnCanceled, NotOnFaulted, NotOnCompletion – to choose when it is supposed to run
- *AttachedToParent* – to create hierarchy of tasks
- *ExecuteSynchronously, RunContinuationAsynchronously* – to choose the thread running it
- *HideScheduler* – to run using the default scheduler instead of the current one
- *LongRunning* – more or less to run on dedicated thread
- Prefer fairness – to run in order

Task.ContinueWith

```
// Token: 0x06003F72 RID: 16242 RVA: 0x000EC0CC File Offset: 0x000EA2CC
internal void ContinueWithCore(Task continuationTask, TaskScheduler scheduler, CancellationToken cancellationToken, TaskContinuationOptions options)
{
    TaskContinuation taskContinuation = new StandardTaskContinuation(continuationTask, options, scheduler);
    if (cancellationToken.CanBeCanceled)
    {
        if (this.IsCompleted || cancellationToken.IsCancellationRequested)
        {
            continuationTask.AssignCancellationToken(cancellationToken, null, null);
        }
        else
        {
            continuationTask.AssignCancellationToken(cancellationToken, this, taskContinuation);
        }
    }
    if (!continuationTask.IsCompleted)
    {
        if ((this.Options & (TaskCreationOptions)1024) != TaskCreationOptions.None && !(this is ITaskCompletionAction))
        {
            TplEtwProvider log = TplEtwProvider.Log;
            if (log.IsEnabled())
            {
                log.AwaitTaskContinuationScheduled(TaskScheduler.Current.Id, Task.CurrentId ?? 0, continuationTask.Id);
            }
        }
        if (!this.AddTaskContinuation(taskContinuation, false))
        {
            taskContinuation.Run(this, true);
        }
    }
}
```

Task.Complete

```
// Token: 0x06006DE2 RID: 28130 RVA: 0x00179FE8 File Offset: 0x001781E8
internal void Complete()
{
    bool flag;
    if (this.Token.IsCancellationRequested)
    {
        flag = base.TrySetCanceled(this.Token);
    }
    else
    {
        if (AsyncCausalityTracer.LoggingOn)
        {
            AsyncCausalityTracer.TraceOperationCompletion(CausalityTraceLevel.Required, base.Id, AsyncCausalityStatus.Completed);
        }
        if (Task.s_asyncDebuggingEnabled)
        {
            Task.RemoveFromActiveTasks(base.Id);
        }
        flag = base.TrySetResult(default(VoidTaskResult));
    }
    if (flag)
    {
        if (this.Timer != null)
        {
            this.Timer.Dispose();
        }
        this.Registration.Dispose();
    }
}
```

Task.TrySetResult

```
// Token: 0x06003E09 RID: 15881 RVA: 0x000E5F34 File Offset: 0x000E4134
internal bool TrySetResult(TResult result)
{
    if (base.IsCompleted)
    {
        return false;
    }
    if (base.AtomicStateUpdate(67108864, 90177536))
    {
        this.m_result = result;
        Interlocked.Exchange(ref this.m_stateFlags, this.m_stateFlags | 16777216);
        Task.ContingentProperties contingentProperties = this.m_contingentProperties;
        if (contingentProperties != null)
        {
            contingentProperties.SetCompleted();
        }
        base.FinishStageThree();
        return true;
    }
    return false;
}
```

Task.FinishContinuations

```
internal void FinishContinuations()
{
    object obj = Interlocked.Exchange(ref this.m_continuationObject, Task.s_taskCompletionSentinel);
    TplEtwProvider.Log.RunningContinuation(this.Id, obj);
    if (obj != null)
    {
        if (AsyncCausalityTracer.LoggingOn)
        {
            AsyncCausalityTracer.TraceSynchronousWorkStart(CausalityTraceLevel.Required, this.Id, CausalitySynchronousWork.CompletionNotification);
        }
        bool flag = (this.m_stateFlags & 134217728) == 0 && Thread.CurrentThread.ThreadState != ThreadState.AbortRequested && (this.m_stateFlags & 64) == 0;
        Action action = obj as Action;
        if (action != null)
        {
            AwaitTaskContinuation.RunOrScheduleAction(action, flag, ref Task.t_currentTask);
            this.LogFinishCompletionNotification();
            return;
        }
        ITaskCompletionAction taskCompletionAction = obj as ITaskCompletionAction;
        if (taskCompletionAction != null)
        {
            if (flag)
            {
                taskCompletionAction.Invoke(this);
            }
            else
            {
                ThreadPool.UnsafeQueueCustomWorkItem(new CompletionActionInvoker(taskCompletionAction, this), false);
            }
            this.LogFinishCompletionNotification();
            return;
        }
        TaskContinuation taskContinuation = obj as TaskContinuation;
        if (taskContinuation != null)
        {
            taskContinuation.Run(this, flag);
            this.LogFinishCompletionNotification();
            return;
        }
```

StandardTaskContinuation.Run

```
// Token: 0x06003FD7 RID: 16343 RVA: 0x000EDB00 File Offset: 0x000EBD00
internal override void Run(Task completedTask, bool bCanInlineContinuationTask)
{
    TaskContinuationOptions options = this.m_options;
    bool flag = completedTask.IsRanToCompletion ? ((options & TaskContinuationOptions.NotOnRanToCompletion) == TaskContinuationOptions.None) : (completedTask.IsCanceled ? ((options &
      TaskContinuationOptions.NotOnCanceled) == TaskContinuationOptions.None) : ((options & TaskContinuationOptions.NotOnFaulted) == TaskContinuationOptions.None));
    Task task = this.m_task;
    if (flag)
    {
        if (!task.IsCanceled && AsyncCausalityTracer.LoggingOn)
        {
            AsyncCausalityTracer.TraceOperationRelation(CausalityTraceLevel.Important, task.Id, CausalityRelation.AssignDelegate);
        }
        task.m_taskScheduler = this.m_taskScheduler;
        if (bCanInlineContinuationTask && (options & TaskContinuationOptions.ExecuteSynchronously) != TaskContinuationOptions.None)
        {
            TaskContinuation.InlineIfPossibleOrElseQueue(task, true);
            return;
        }
        try
        {
            task.ScheduleAndStart(true);
            return;
        }
        catch (TaskSchedulerException)
        {
            return;
        }
    }
    task.InternalCancel(false);
}
```

Disposing a *Task*

Task **may** allocated *WaitHandle* which implements *IDisposable*.

Disposing a *Task* in .NET 4 was making it unusable — we couldn't even schedule continuation.

In .NET 4.5 this was changed, *Task* is still usable, only *WaitHandle* is not.

WaitHandle was created when *Task.WaitAny* or *Task.WaitAll* was called, this is no longer true.

Starting in .NET 4.5 *WaitHandle* is allocated only when it is explicitly accessed.

Summary:

- .**NET 4** — **don't dispose** unless you have to. Do so only if you are sure that the *Task* will never be used again.
- .**NET 4.5** — it shouldn't make a difference so probably **don't bother.**

Task.Status

State	IsCompleted	IsCanceled	IsFaulted
RanToCompletion	**True**	False	False
Canceled	**True**	**True**	False
Faulted	**True**	False	**True**
Other	False	False	False

Task.Id

Generated on demand.

Can be reused — you can generate collision!

Independent from *TaskScheduler.Id*.

0 is not a valid identifier.

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 |           - |              - |
```

SynchronizationContext internals

SERVICE LOCATOR FOR PROMISE ORCHESTRATION

ISynchronizeInvoke – life before *SynchronizationContext*

Provides a way to synchronously or asynchronously execute a delegate:

- *InvokeRequired* – checks if invoking is requred, effectively if we are running on the same thread
- *Invoke* – synchronous invocation
- *BeginInvoke*, *EndInvoke* – asynchronous invocation

It ties communication and threads.

If we don't need specific thread – as in ASP.NET – we should not use *ISynchronizeInvoke*.

This is how *SynchronizationContext* emerged.

ExecutionContext and other

Bag holding logical context of the execution.

Contains *SynchronizationContext*, *LogicalCallContext*, *SecurityContext*, *HostExecutionContext*, *CallContext* etc.

Does not need to rely on *Thread Local Storage* (TLS) and is passed correctly through asynchronous points — will follow to the other thread.

Before .NET 4.5 *LogicalCallContext* was performing shadow copies and couldn't be used between asynchronous points of invocation.

Starting in .NET 4.6 there is an *AsyncLocal<T>* class working as *TLS* variables for tasks.

Methods with *Unsafe** do not propagate the context — for instance *ThreadPool.UnsafeQueueUserWorkItem*.

AsyncLocal<T>

Program.cs

AsyncLocal

AsyncLocal.Program

```
1  using System;
2  using System.Threading;
3  using System.Threading.Tasks;
4
5  namespace AsyncLocal
6  {
7      class Program
8      {
9          static AsyncLocal<string> asyncLocal = new AsyncLocal<string>();
10
11         static ThreadLocal<string> threadLocal = new ThreadLocal<string>();
12
13         static void Main(string[] args)
14         {
15             AsyncMethodA().Wait();
16         }
17
18         static async Task AsyncMethodA()
19         {
20             asyncLocal.Value = "Value 1";
21             threadLocal.Value = "Value 1";
22             var task1 = AsyncMethodB("Value 1");
23
24             asyncLocal.Value = "Value 2";
25             threadLocal.Value = "Value 2";
26             var task2 = AsyncMethodB("Value 2");
27
28             await task1;
29             await task2;
30         }
31
32         static async Task AsyncMethodB(string expectedValue)
33         {
34             Console.WriteLine("Entering AsyncMethodB.");
35             Console.WriteLine("\tExpected '{0}', AsyncLocal value is '{1}', ThreadLocal value is '{2}'",
36                               expectedValue, asyncLocal.Value, threadLocal.Value);
37             await Task.Delay(100);
38             Console.WriteLine("Exiting AsyncMethodB.");
39             Console.WriteLine("\tExpected '{0}', got '{1}', ThreadLocal value is '{2}'",
40                               expectedValue, asyncLocal.Value, threadLocal.Value);
41         }
42     }
43 }
```

C:\WINDOWS\system32\cmd.exe

```
Entering AsyncMethodB.
        Expected 'Value 1', AsyncLocal value is 'Value 1', ThreadLocal value is 'Value 1'
Entering AsyncMethodB.
        Expected 'Value 2', AsyncLocal value is 'Value 2', ThreadLocal value is 'Value 2'
Exiting AsyncMethodB.
        Expected 'Value 2', got 'Value 2', ThreadLocal value is ''
Exiting AsyncMethodB.
        Expected 'Value 1', got 'Value 1', ThreadLocal value is ''
Press any key to continue . . .
```

Solution Explore

Wątki

Zadani

SynchronizationContext

The *SynchronizationContext* class is a base class that provides a free-threaded context with no synchronization:

- *OperationStarted* and *OperationCompleted* – handles notifications
- *Send* – synchronous message
- *Post* – asynchronous message
- *Current* – gets synchronization context for the thread

The purpose of the synchronization model implemented by this class is to allow the internal asynchronous/synchronous operations of the common language runtime to behave properly with different synchronization models

SynchronizationContext

When awaiting the awaitable type the current context is captured. Later, the rest of the method is posted on the context.

We can use *ConfigureAwait(false)* to avoid capturing the context. Rule of thumb — always use it unless you are sure that you need a context.

SynchronizationContext

Synchronization context is a **global variable**.

If *SynchronizationContext.Current* is not null then this context is captured:

- For UI thread it is UI context – *WindowsFormsSynchronizationContext*, *DispatcherSynchronizationContext*, *WinRTSynchronizationContext, WinRTCoreDispatcherBasedSynchronizationContext*
  - Implemented via event loop, for instance.
- For ASP.NET request it is ASP.NET context — *AspNetSynchronizationContext*
  - This can be **different** thread than original one, but still the request context is the same.

Otherwise it is current *TaskScheduler*:

- *TaskScheduler.Default* is the thread pool context.
- ASP.NET Core doesn't have spearate context – no risk of deadlock, no need to use *ConfigureAwait(false)*

Each method can have its own context.

SynchronizationContext

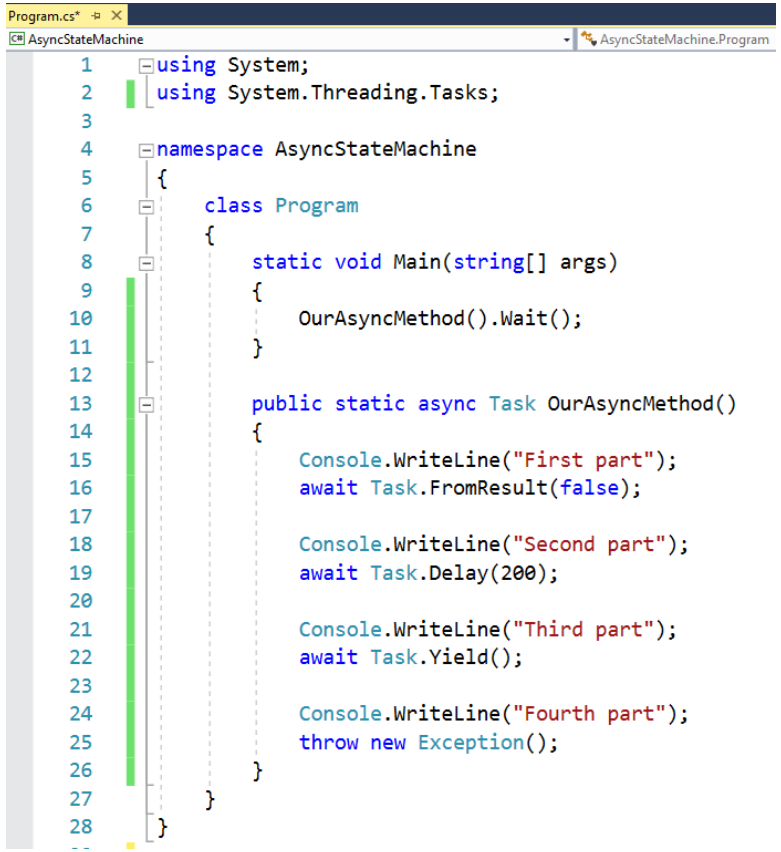
	Specific thread executing the code	Delegates executed serially	Delegates executed in order	Send is synchronous	Post is asynchronous
Default (Thread Pool based)	No – any thread in the thread pool	No	No	Yes	Yes
ASP.NET	No – any thread in the thread pool	Yes	No	Yes	No
WinForms, WPF, WinRT, Xamarin, Blazor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

State machine

COROUTINE COMPILER LEVEL TRANSFORMATION

State machine 1 – before compilation




```
1   using System;
2   using System.Threading.Tasks;
3
4   namespace AsyncStateMachine
5   {
6       class Program
7       {
8           static void Main(string[] args)
9           {
10              OurAsyncMethod().Wait();
11          }
12
13          public static async Task OurAsyncMethod()
14          {
15              Console.WriteLine("First part");
16              await Task.FromResult(false);
17
18              Console.WriteLine("Second part");
19              await Task.Delay(200);
20
21              Console.WriteLine("Third part");
22              await Task.Yield();
23
24              Console.WriteLine("Fourth part");
25              throw new Exception();
26          }
27      }
28  }
```

OurAsyncMethod has four parts

- First part will run synchronously as the *Task.FromResult* is already resolved
- Second part will eventually block because of the delay
- Third part will block and explicitly create continuation
- Fourth part with just throw exception

State machine 2 – method after compilation

Program

```
1  using System;
2  using System.Diagnostics;
3  using System.Runtime.CompilerServices;
4  using System.Threading.Tasks;
5
6  namespace AsyncStateMachine
7  {
8      // Token: 0x02000002 RID: 2
9      internal class Program
10     {
11         // Token: 0x06000001 RID: 1 RVA: 0x00002050 File Offset: 0x00000250
12         private static void Main(string[] args)
13         {
14             Program.OurAsyncMethod().Wait();
15         }
16
17         // Token: 0x06000002 RID: 2 RVA: 0x00002060 File Offset: 0x00000260
18         [DebuggerStepThrough]
19         public static Task OurAsyncMethod()
20         {
21             Program.<OurAsyncMethod>d__1 <OurAsyncMethod>d__ = new Program.<OurAsyncMethod>d__1();
22             <OurAsyncMethod>d__.<>t__builder = AsyncTaskMethodBuilder.Create();
23             <OurAsyncMethod>d__.<>1__state = -1;
24             AsyncTaskMethodBuilder <>t__builder = <OurAsyncMethod>d__.<>t__builder;
25             <>t__builder.Start<Program.<OurAsyncMethod>d__1>(ref <OurAsyncMethod>d__);
26             return <OurAsyncMethod>d__.<>t__builder.Task;
27         }
```

async is no longer there — it is only on C# level.

DebuggerStepThroughAttribute tells the debugger to step through (ignore) the method.

Program.<OurAsyncMethod>d__1 *<OurAsyncMethod>d__* type created to encapsulate state machine pieces.

State is initialized to -1 meaning „ready to do some work”.

AsyncTaskMethodBuilder is a .NET class capable of executing the state machine.

Effectively we prepare the machine, **start it** and return the *Task* object with the result.

State machine 3 – fields

```
// Token: 0x02000003 RID: 3
[CompilerGenerated]
private sealed class <OurAsyncMethod>d__1 : IAsyncStateMachine
{
    // Token: 0x06000006 RID: 6 RVA: 0x00002270 File Offset: 0x00000470
    [DebuggerHidden]
    void IAsyncStateMachine.SetStateMachine(IAsyncStateMachine stateMachine)
    {
    }

    // Token: 0x04000001 RID: 1
    public int <>1__state;

    // Token: 0x04000002 RID: 2
    public AsyncTaskMethodBuilder <>t__builder;

    // Token: 0x04000003 RID: 3
    private TaskAwaiter<bool> <>u__1;

    // Token: 0x04000004 RID: 4
    private TaskAwaiter <>u__2;

    // Token: 0x04000005 RID: 5
    private YieldAwaitable.YieldAwaiter <>u__3;
}
```

Program.<OurAsyncMethod>d__1 *<OurAsyncMethod>d__* type created to encapsulate state machine pieces.

<OurAsyncMethod>d__.<>1__state variable maintains the state

- Initially it is set to -1 meaning „not started”
- -2 means „done”
- Non-negative states indicate different pieces of the state machine

Three different awaiters as we have *await* three times in the original method.

State machine 4 – *Start* method

```
25      /// <summary>Begins running the builder with the associated state machine.</summary>
26      /// <param name="stateMachine">The state machine instance, passed by reference.</param>
27      /// <typeparam name="TStateMachine">The type of the state machine.</typeparam>
28      /// <exception cref="T:System.ArgumentNullException">
29      ///   <paramref name="stateMachine" /> is null.</exception>
30      // Token: 0x06005CB9 RID: 23737 RVA: 0x00144E38 File Offset: 0x00143038
31      [SecuritySafeCritical]
32      [DebuggerStepThrough]
33      [__DynamicallyInvokable]
34      public void Start<TStateMachine>(ref TStateMachine stateMachine) where TStateMachine : IAsyncStateMachine
35      {
36          if (stateMachine == null)
37          {
38              throw new ArgumentNullException("stateMachine");
39          }
40          ExecutionContextSwitcher executionContextSwitcher = default(ExecutionContextSwitcher);
41          RuntimeHelpers.PrepareConstrainedRegions();
42          try
43          {
44              ExecutionContext.EstablishCopyOnWriteScope(ref executionContextSwitcher);
45              stateMachine.MoveNext();
46          }
47          finally
48          {
49              executionContextSwitcher.Undo();
50          }
51      }
```

State machine 5 – exception handling

```
29          // Token: 0x02000003 RID: 3
30          [CompilerGenerated]
31          private sealed class <OurAsyncMethod>d__1 : IAsyncStateMachine
32          {
33              // Token: 0x06000005 RID: 5 RVA: 0x000020AC File Offset: 0x000002AC
34              void IAsyncStateMachine.MoveNext()
35              {
36                  int num = this.<>1__state;
37                  try
38                  {
39                      // ...
99                  }
100                 catch (Exception exception)
101                 {
102                     this.<>1__state = -2;
103                     this.<>t__builder.SetException(exception);
104                 }
105             }
```

We capture the state to local variable.

We handle all exceptions and terminate the machine if needed.

State machine 6 – states

```
37          try
38          {
39              TaskAwaiter<bool> awaiter;
40              TaskAwaiter awaiter2;
41              YieldAwaitable.YieldAwaiter awaiter3;
42              switch (num)
43              {
44              case 0:                          1
45                  // ...
48                  break;
49              case 1:                          2
50                  // ...
53                  goto IL_FA;
54              case 2:                          3
55                  // ...
58                  goto IL_168;
59              default:                         1
60                  // ...
70                  break;
71              }
72              // ...                           2
83              IL_FA:
84              // ...                           3
95              IL_168:
96              // ...                           4
99          }
```

Awaiter at the beginning for different result types.

Four different branches as we have *await* three times generating four blocks.

State machine 7 – num = -1, state = -1

`await Task.FromResult(false);`

```
42          switch (num)
43          {
44          case 0:
45              awaiter = this.<>u__1;
46              this.<>u__1 = default(TaskAwaiter<bool>);
47              this.<>1__state = -1;
48              break;
49          // ...
59          default:
60              Console.WriteLine("First part");
61              awaiter = Task.FromResult<bool>(false).GetAwaiter();
62              if (!awaiter.IsCompleted)
63              {
64                  this.<>1__state = 0;
65                  this.<>u__1 = awaiter;
66                  Program.<OurAsyncMethod>d__1 <OurAsyncMethod>d__ = this;
67                  this.<>t__builder.AwaitUnsafeOnCompleted<TaskAwaiter<bool>, Program.<OurAsyncMethod>d__1>
                      (ref awaiter, ref <OurAsyncMethod>d__);
68                  return;
69              }
70              break;
71          }
72          awaiter.GetResult();
```

It starts in *default*.

We print to the console and get awaiter for the result.

If it is completed (as this is the case now)

- We call *GetResult* which returns the value immediately
- We end in line 72
- num = -1, state = -1

State machine 8 – num = -1, state = -1

```
await Task.Delay(200);
```

```
42          switch (num)
43          {
44          // ...
49          case 1:
50              awaiter2 = this.<>u__2;
51              this.<>u__2 = default(TaskAwaiter);
52              this.<>1__state = -1;
53              goto IL_FA;
54          // ...
71          }
72          // ...
73          Console.WriteLine("Second part");
74          awaiter2 = Task.Delay(200).GetAwaiter();
75          if (!awaiter2.IsCompleted)
76          {
77              this.<>1__state = 1;
78              this.<>u__2 = awaiter2;
79              Program.<OurAsyncMethod>d__1 <OurAsyncMethod>d__ = this;
80              this.<>t__builder.AwaitUnsafeOnCompleted<TaskAwaiter, Program.<OurAsyncMethod>d__1>(ref
                  awaiter2, ref <OurAsyncMethod>d__);
81              return;
82          }
83          IL_FA:
84          awaiter2.GetResult();
```

It starts in line 73.

We print to the console and get the awaiter.

If it is not completed

- We change the state (so we know where to come back)
- We call *AwaitUnsafeOnCompleted* (see in a bit)
- And then we **return**

Later we continue in case 1

- We jump to the label IL_FA and end in line 84
- num = 1, state = -1

AsyncTaskMethodBuilder.AwaitUnsafeOnCompleted

```
[__DynamicallyInvokable]
public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine stateMachine) where TAwaiter : ICriticalNotifyCompletion where TStateMachine : IAsyncStateMachine
{
  this.m_builder.AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(ref awaiter, ref stateMachine);
}
```

```
[__DynamicallyInvokable]
public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine stateMachine) where TAwaiter : ICriticalNotifyCompletion where TStateMachine :
  IAsyncStateMachine
{
    try
    {
        AsyncMethodBuilderCore.MoveNextRunner runner = null;
        Action completionAction = this.m_coreState.GetCompletionAction(AsyncCausalityTracer.LoggingOn ? this.Task : null, ref runner);
        if (this.m_coreState.m_stateMachine == null)
        {
            Task<TResult> task = this.Task;
            this.m_coreState.PostBoxInitialization(stateMachine, runner, task);
        }
        awaiter.UnsafeOnCompleted(completionAction);
    }
    catch (Exception exception)
    {
        AsyncMethodBuilderCore.ThrowAsync(exception, null);
    }
}
```

AwaitUnsafeOnCompleted — GetCompletionAction

```
// Token: 0x06005C7A RID: 23674 RVA: 0x00143828 File Offset: 0x00141A28
[SecuritySafeCritical]
internal Action GetCompletionAction(Task taskForTracing, ref AsyncMethodBuilderCore.MoveNextRunner runnerToInitialize)
{
    Debugger.NotifyOfCrossThreadDependency();
    ExecutionContext executionContext = ExecutionContext.FastCapture();
    Action action;
    AsyncMethodBuilderCore.MoveNextRunner moveNextRunner;
    if (executionContext != null && executionContext.IsPreAllocatedDefault)
    {
        action = this.m_defaultContextAction;
        if (action != null)
        {
            return action;
        }
        moveNextRunner = new AsyncMethodBuilderCore.MoveNextRunner(executionContext, this.m_stateMachine);
        action = new Action(moveNextRunner.Run);
        if (taskForTracing != null)
        {
            action = (this.m_defaultContextAction = this.OutputAsyncCausalityEvents(taskForTracing, action));
        }
        else
        {
            this.m_defaultContextAction = action;
        }
    }
    else
    {
        moveNextRunner = new AsyncMethodBuilderCore.MoveNextRunner(executionContext, this.m_stateMachine);
        action = new Action(moveNextRunner.Run);
        if (taskForTracing != null)
        {
            action = this.OutputAsyncCausalityEvents(taskForTracing, action);
        }
    }
    if (this.m_stateMachine == null)
    {
        runnerToInitialize = moveNextRunner;
    }
    return action;
}
```

MoveNextRunner

```
// Token: 0x06006EF5 RID: 28405 RVA: 0x0017D154 File Offset: 0x0017B354
[SecuritySafeCritical]
internal void Run()
{
    if (this.m_context != null)
    {
        try
        {
            ContextCallback contextCallback = AsyncMethodBuilderCore.MoveNextRunner.s_invokeMoveNext;
            if (contextCallback == null)
            {
                contextCallback = (AsyncMethodBuilderCore.MoveNextRunner.s_invokeMoveNext = new ContextCallback(AsyncMethodBuilderCore.MoveNextRunner.InvokeMoveNext));
            }
            ExecutionContext.Run(this.m_context, contextCallback, this.m_stateMachine, true);
            return;
        }
        finally
        {
            this.m_context.Dispose();
        }
    }
    this.m_stateMachine.MoveNext();
}
```

```
// Token: 0x06006EF6 RID: 28406 RVA: 0x0017D1C4 File Offset: 0x0017B3C4
[SecurityCritical]
private static void InvokeMoveNext(object stateMachine)
{
    ((IAsyncStateMachine)stateMachine).MoveNext();
}
```

TaskAwaiter.AwaitUnsafeOnCompleted

```
[SecurityCritical]
[__DynamicallyInvokable]
public void UnsafeOnCompleted(Action continuation)
{
  TaskAwaiter.OnCompletedInternal((Task) this.m_task, continuation, true, false);
}

[SecurityCritical]
[MethodImpl(MethodImplOptions.NoInlining)]
internal static void OnCompletedInternal(Task task, Action continuation, bool continueOnCapturedContext, bool flowExecutionContext)
{
  if (continuation == null)
    throw new ArgumentNullException("continuation");
  StackCrawlMark stackMark = StackCrawlMark.LookForMyCaller;
  if (TplEtwProvider.Log.IsEnabled() || Task.s_asyncDebuggingEnabled)
    continuation = TaskAwaiter.OutputWaitEtwEvents(task, continuation);
  task.SetContinuationForAwait(continuation, continueOnCapturedContext, flowExecutionContext, ref stackMark);
}
```

Task.SetContinuationForAwait

```
// Token: 0x06003F46 RID: 16198 RVA: 0x000EB46C File Offset: 0x000E966C
[SecurityCritical]
internal void SetContinuationForAwait(Action continuationAction, bool continueOnCapturedContext, bool flowExecutionContext, ref StackCrawlMark stackMark)
{
    TaskContinuation taskContinuation = null;
    if (continueOnCapturedContext)
    {
        SynchronizationContext currentNoFlow = SynchronizationContext.CurrentNoFlow;
        if (currentNoFlow != null && currentNoFlow.GetType() != typeof(SynchronizationContext))
        {
            taskContinuation = new SynchronizationContextAwaitTaskContinuation(currentNoFlow, continuationAction, flowExecutionContext, ref stackMark);
        }
        else
        {
            TaskScheduler internalCurrent = TaskScheduler.InternalCurrent;
            if (internalCurrent != null && internalCurrent != TaskScheduler.Default)
            {
                taskContinuation = new TaskSchedulerAwaitTaskContinuation(internalCurrent, continuationAction, flowExecutionContext, ref stackMark);
            }
        }
    }
    if (taskContinuation == null && flowExecutionContext)
    {
        taskContinuation = new AwaitTaskContinuation(continuationAction, true, ref stackMark);
    }
    if (taskContinuation != null)
    {
        if (!this.AddTaskContinuation(taskContinuation, false))
        {
            taskContinuation.Run(this, false);
            return;
        }
    }
    else if (!this.AddTaskContinuation(continuationAction, false))
    {
        AwaitTaskContinuation.UnsafeScheduleAction(continuationAction, this);
    }
}
```

SynchronizationContextAwaitTaskContinuation.Run

```
// Token: 0x06003FDA RID: 16346 RVA: 0x000EDBFC File Offset: 0x000EBDFC
[SecuritySafeCritical]
internal sealed override void Run(Task task, bool canInlineContinuationTask)
{
    if (canInlineContinuationTask && this.m_syncContext == SynchronizationContext.CurrentNoFlow)
    {
        base.RunCallback(AwaitTaskContinuation.GetInvokeActionCallback(), this.m_action, ref Task.t_currentTask);
        return;
    }
    TplEtwProvider log = TplEtwProvider.Log;
    if (log.IsEnabled())
    {
        this.m_continuationId = Task.NewId();
        log.AwaitTaskContinuationScheduled((task.ExecutingTaskScheduler ?? TaskScheduler.Default).Id, task.Id, this.m_continuationId);
    }
    base.RunCallback(SynchronizationContextAwaitTaskContinuation.GetPostActionCallback(), this, ref Task.t_currentTask);
}
```

SynchronizationContextAwaitTaskContinuation.PostAction

```
// Token: 0x06003FDB RID: 16347 RVA: 0x000EDC80 File Offset: 0x000EBE80
[SecurityCritical]
private static void PostAction(object state)
{
    SynchronizationContextAwaitTaskContinuation synchronizationContextAwaitTaskContinuation = (SynchronizationContextAwaitTaskContinuation)state;
    TplEtwProvider log = TplEtwProvider.Log;
    if (log.TasksSetActivityIds && synchronizationContextAwaitTaskContinuation.m_continuationId != 0)
    {
        synchronizationContextAwaitTaskContinuation.m_syncContext.Post(SynchronizationContextAwaitTaskContinuation.s_postCallback,
          SynchronizationContextAwaitTaskContinuation.GetActionLogDelegate(synchronizationContextAwaitTaskContinuation.m_continuationId, synchronizationContextAwaitTaskContinuation.m_action));
        return;
    }
    synchronizationContextAwaitTaskContinuation.m_syncContext.Post(SynchronizationContextAwaitTaskContinuation.s_postCallback, synchronizationContextAwaitTaskContinuation.m_action);
}
```

State machine 9 – num = 1, state = -1

`await Task.Yield();`

```
42          switch (num)
43          {
44          // ...
54          case 2:
55              awaiter3 = this.<>u__3;
56              this.<>u__3 = default(YieldAwaitable.YieldAwaiter);
57              this.<>1__state = -1;
58              goto IL_168;
59          // ...
71          }
72          // ...
85          Console.WriteLine("Third part");
86          awaiter3 = Task.Yield().GetAwaiter();
87          if (!awaiter3.IsCompleted)
88          {
89              this.<>1__state = 2;
90              this.<>u__3 = awaiter3;
91              Program.<OurAsyncMethod>d__1 <OurAsyncMethod>d__ = this;
92              this.<>t__builder.AwaitUnsafeOnCompleted<YieldAwaitable.YieldAwaiter,
                    Program.<OurAsyncMethod>d__1>(ref awaiter3, ref <OurAsyncMethod>d__);
93              return;
94          }
95          IL_168:
96          awaiter3.GetResult();
```

We start in line 85

We print to the console, get awaiter and check if it is completed.

It is not:

- We set the state
- Wait for the task
- And **return**

Later we continue in case 2, jump to the label IL_168 and end in line 96.

num = 2, state = -1

YieldAwaiter.AwaitUnsafeOnCompleted

```
[SecurityCritical]
private static void QueueContinuation(Action continuation, bool flowContext)
{
    if (continuation == null)
    {
        throw new ArgumentNullException("continuation");
    }
    if (TplEtwProvider.Log.IsEnabled())
    {
        continuation = YieldAwaitable.YieldAwaiter.OutputCorrelationEtwEvent(continuation);
    }
    SynchronizationContext currentNoFlow = SynchronizationContext.CurrentNoFlow;
    if (currentNoFlow != null && currentNoFlow.GetType() != typeof(SynchronizationContext))
    {
        currentNoFlow.Post(YieldAwaitable.YieldAwaiter.s_sendOrPostCallbackRunAction, continuation);
        return;
    }
    TaskScheduler taskScheduler = TaskScheduler.Current;
    if (taskScheduler != TaskScheduler.Default)
    {
        Task.Factory.StartNew(continuation, default(CancellationToken), TaskCreationOptions.PreferFairness, taskScheduler);
        return;
    }
    if (flowContext)
    {
        ThreadPool.QueueUserWorkItem(YieldAwaitable.YieldAwaiter.s_waitCallbackRunAction, continuation);
        return;
    }
    ThreadPool.UnsafeQueueUserWorkItem(YieldAwaitable.YieldAwaiter.s_waitCallbackRunAction, continuation);
}
```

State machine 10 – num = 2, state = -1 After last `await`

```
37          try
38          {
39              // ...
97              Console.WriteLine("Fourth part");
98              throw new Exception();
99          }
100         catch (Exception exception)
101         {
102             // ...
104         }
```

We start in line 97.

This part had no *await* in it so we just execute the code.

We print to the console and throw the exception.

State machine in Debug vs Release

DEBUG = CLASS

```
// Token: 0x02000003 RID: 3
[CompilerGenerated]
private sealed class <OurAsyncMethod>d__1
```

RELEASE = STRUCT

```
// Token: 0x02000003 RID: 3
[CompilerGenerated]
[StructLayout(LayoutKind.Auto)]
private struct <OurAsyncMethod>d__1
```

Task vs *void*

ASYNC TASK

```
// Token: 0x04000002 RID: 2
public AsyncTaskMethodBuilder <>t__builder;
```

ASYNC VOID

```
// Token: 0x04000002 RID: 2
public AsyncVoidMethodBuilder <>t__builder;
```

They capture the context in the same way.

If exception is thrown in *async Task*, it is then remembered in the context of *Task* object and propagated when awaited or cleaned up.

In *async void* methods the exception is propagated immediately. This results in throwing unhandled exception on the thread pool which kills the application.

AsyncTaskMethodBuilder.SetException

```
public void SetException(Exception exception)
{
    if (exception == null)
    {
        throw new ArgumentNullException("exception");
    }
    Task<TResult> task = this.m_task;
    if (task == null)
    {
        task = this.Task;
    }
    OperationCanceledException ex = exception as OperationCanceledException;
    if (!((ex != null) ? task.TrySetCanceled(ex.CancellationToken, ex) : task.TrySetException(exception)))
    {
        throw new InvalidOperationException(Environment.GetResourceString("TaskT_TransitionToFinal_AlreadyCompleted"));
    }
}
```

AsyncVoidMethodBuilder.SetException

```
public void SetException(Exception exception)
{
    if (exception == null)
    {
        throw new ArgumentNullException("exception");
    }
    if (AsyncCausalityTracer.LoggingOn)
    {
        AsyncCausalityTracer.TraceOperationCompletion(CausalityTraceLevel.Required, this.Task.Id, AsyncCausalityStatus.Error);
    }
    if (this.m_synchronizationContext != null)
    {
        try
        {
            AsyncMethodBuilderCore.ThrowAsync(exception, this.m_synchronizationContext);
            return;
        }
        finally
        {
            this.NotifySynchronizationContextOfCompletion();
        }
    }
    AsyncMethodBuilderCore.ThrowAsync(exception, null);
}
```

AsyncVoidMethodBuilder.SetException

```
internal static void ThrowAsync(Exception exception, SynchronizationContext targetContext)
{
    ExceptionDispatchInfo exceptionDispatchInfo = ExceptionDispatchInfo.Capture(exception);
    if (targetContext != null)
    {
        try
        {
            targetContext.Post(delegate(object state)
            {
                ((ExceptionDispatchInfo)state).Throw();
            }, exceptionDispatchInfo);
            return;
        }
        catch (Exception ex)
        {
            exceptionDispatchInfo = ExceptionDispatchInfo.Capture(new AggregateException(new Exception[]
            {
                exception,
                ex
            }));
        }
    }
    if (!WindowsRuntimeMarshal.ReportUnhandledError(exceptionDispatchInfo.SourceException))
    {
        ThreadPool.QueueUserWorkItem(delegate(object state)
        {
            ((ExceptionDispatchInfo)state).Throw();
        }, exceptionDispatchInfo);
    }
}
```

Deadlocks

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

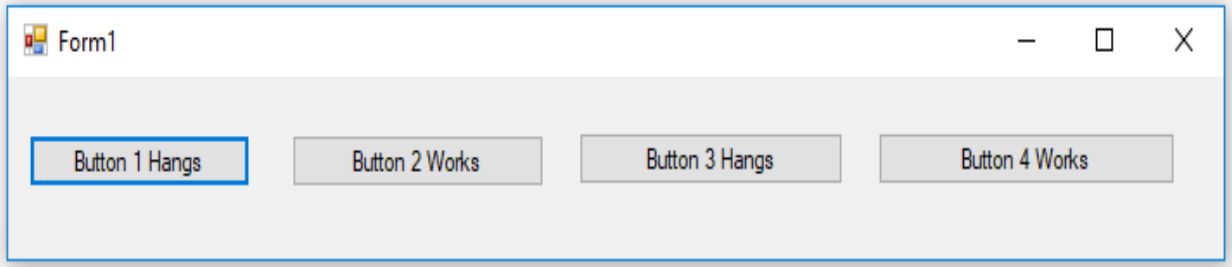
```
Waiting
        Waiting
        Done waiting
Done!
Press any key to continue . . .
```

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
```

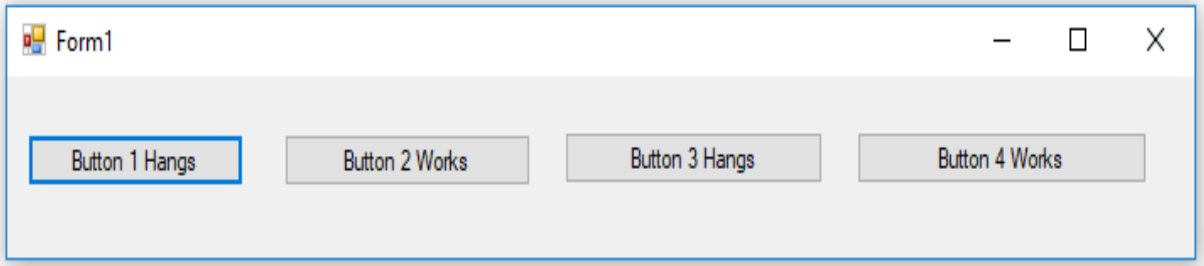


GUI

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
```

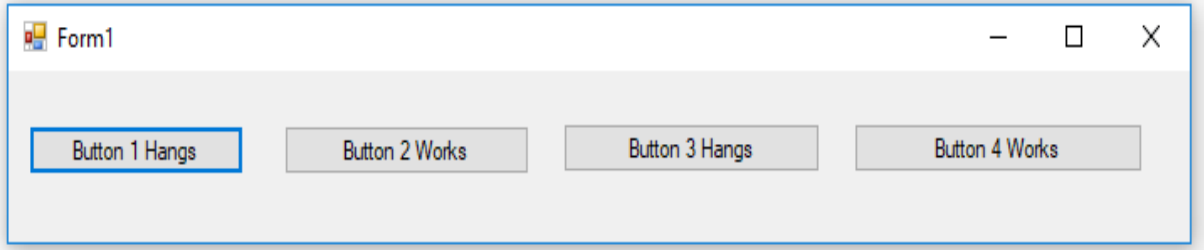


GUI

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
```

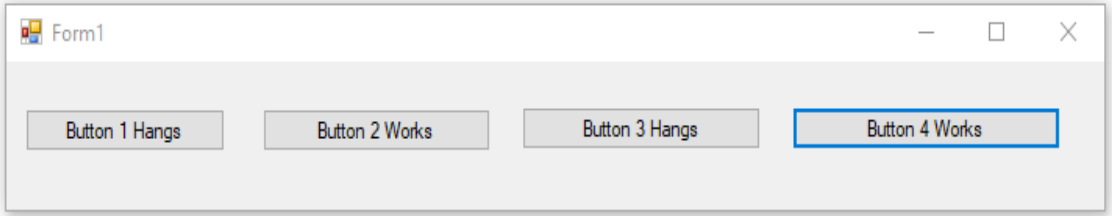


GUI

Form1.cs

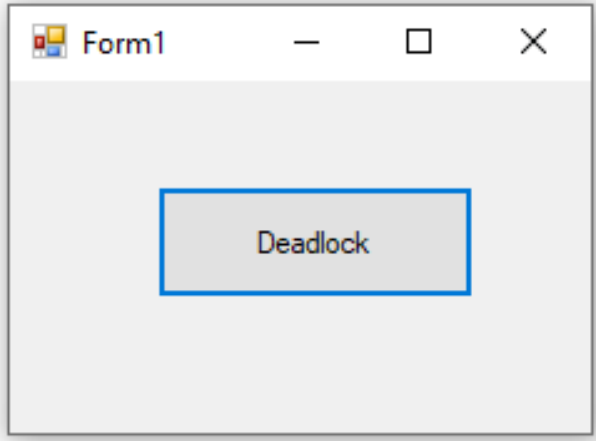
AsyncSyncWait_Forms | AsyncSyncWait_Forms.Form1 | button4_Click(object sender, EventArgs e)

```
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");
    }
}
```

TaskCompletionSource

By default *TaskCompletionSource* runs continuations synchronously when setting the result.

Continuations work differently for *async* and *ContinueWith*:

- For *await* they run synchronously (almost always).
- For *ContinueWith* they run asynchronously (almost always).

We can modify *TaskCompletionSource* behavior by passing continuation creation flags.

As a workaround we can explicitly force the application to yield the continuation.

Use *TaskCreationOptions.RunContinuationsAsynchronously* where possible.

TaskCompletionSource

Program.cs

TaskCompletionSource | TaskCompletionSource.Program | Main(string[] args)

```
using System;
using System.Collections.Concurrent;
using System.Threading.Tasks;

namespace TaskCompletionSource
{
    class Program
    {
        static void Main(string[] args)
        {
            Task.Run(async () =>
            {
                using (var database = new Database())
                using (var logger = new Logger(database))
                {
                    logger.WriteLine("Message1");
                    await Task.Delay(200);

                    Console.WriteLine("Saving something to database...");

                    await database.SaveAsync("Message to database");
                    Console.WriteLine("We are done");
                }
            }).Wait();
        }
    }

    class Logger : IDisposable
    {
        private readonly Database database;
        private readonly BlockingCollection<string> queue = new BlockingCollection<string>();
        private readonly Task worker;

        public Logger(Database facade)
        {
            database = facade;
            worker = Task.Run(SaveMessage);
        }

        public void Dispose()
        {
            queue.CompleteAdding();
        }
```

C:\WINDOWS\system32\cmd.exe

```
Logger: Message1
Database: executing 'Message1'...
Saving something to database...
```

Solution Explorer

Search Solution Explorer (Ctrl+;)

Wątki

Zadania

ThreadPool starvation

ThreadPool supports global queue. Each thread has local queue as well.

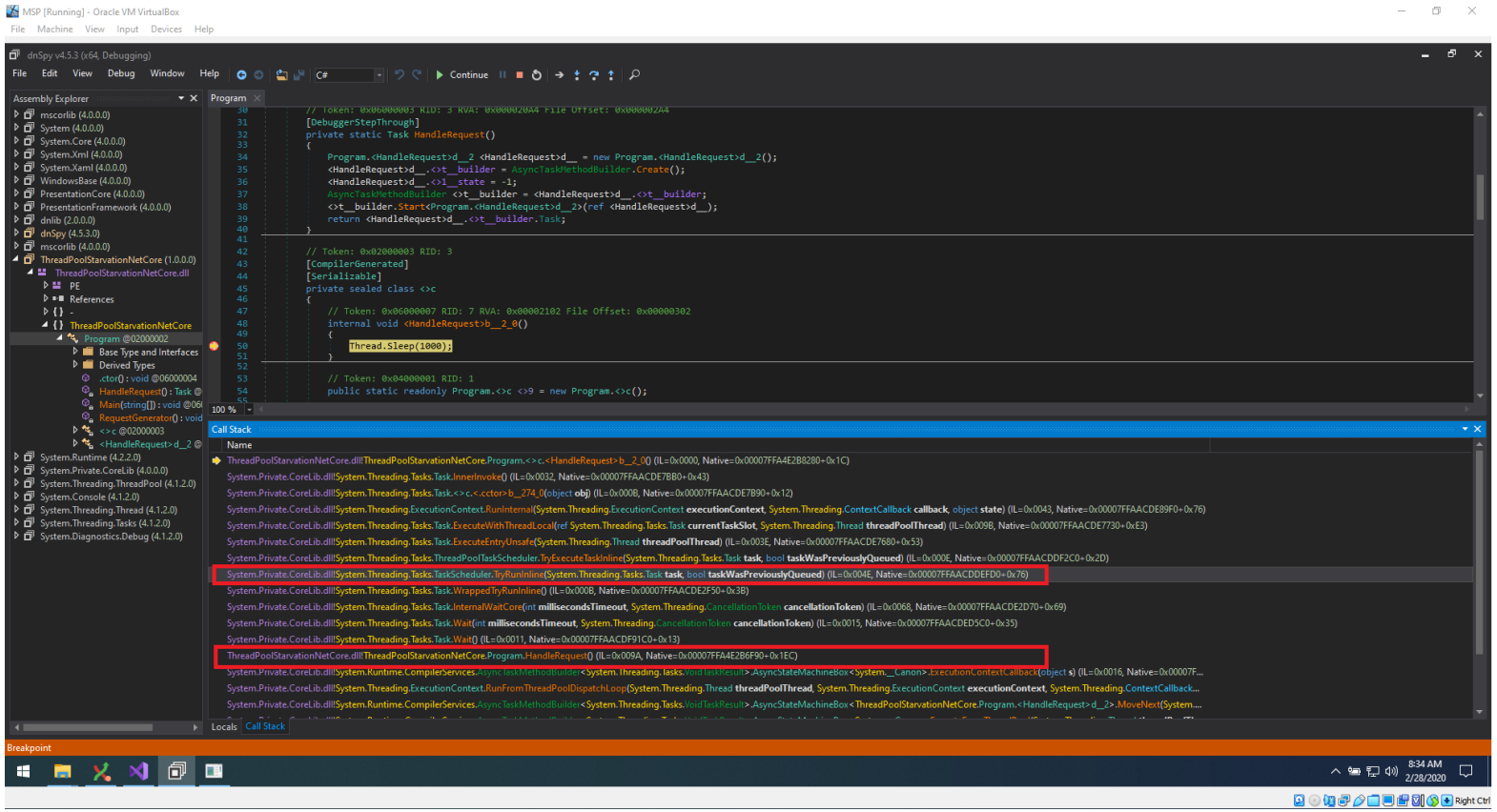
Tasks are scheduled to global queue when:

- Thread enqueing item is not a thread pool thread
- *ThreadPool.QueueUserWorkItem* is used
- *Task.Factory.StartNew* with PreferFariness is used
- *Task.Yield* is used

Otherwise items are scheduled to local queue.

Tasks can be also awaited inline.

ThreadPool starvation



Exception handling

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*.

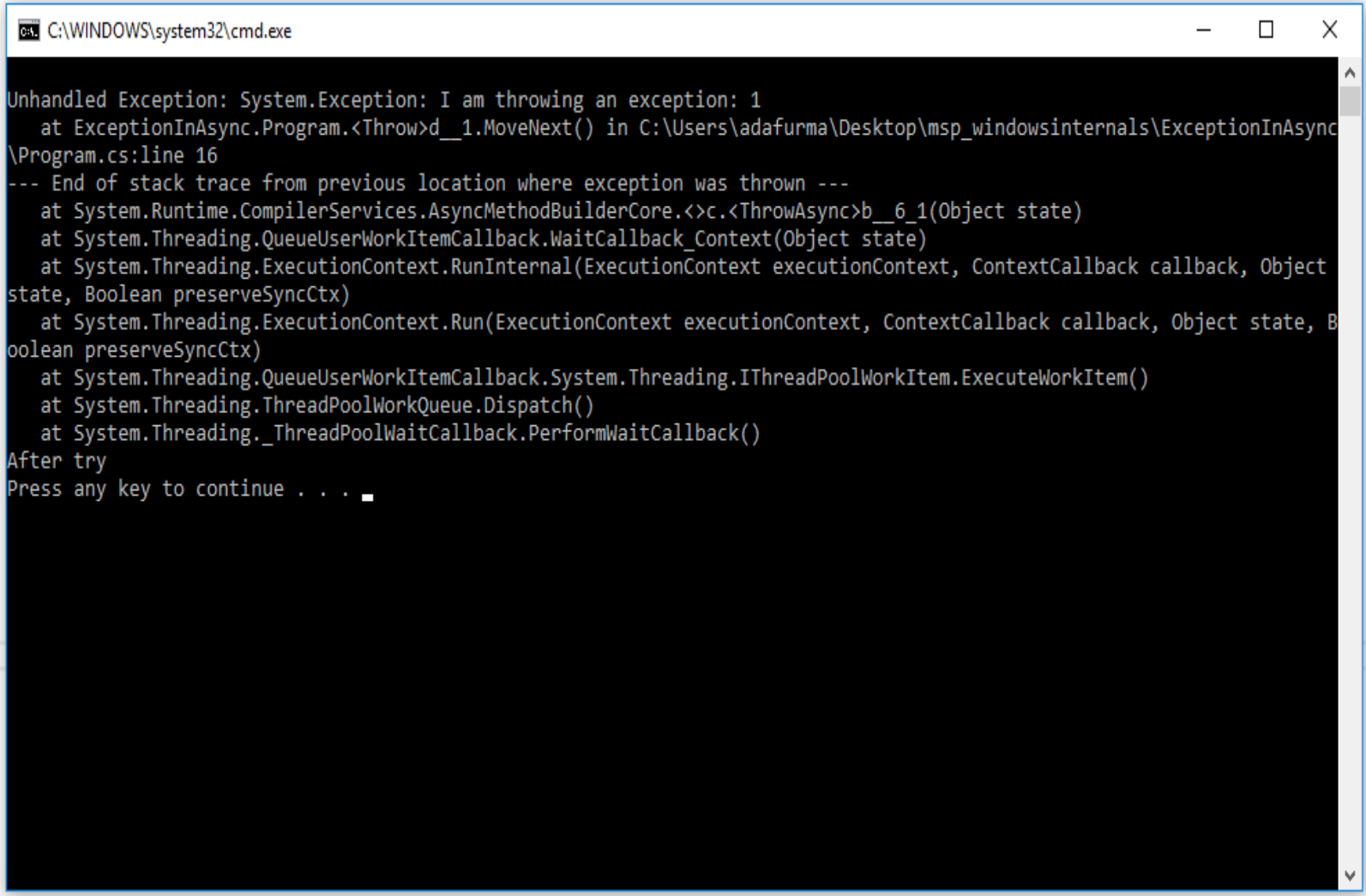
If there is an unobserved exception, it is raised by finalizer thread in *UnobservedTaskException* event where it can be cleared. If not cleared, the process dies (.NET 4) or the exception is suppressed (.NET 4.5).

```
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
```

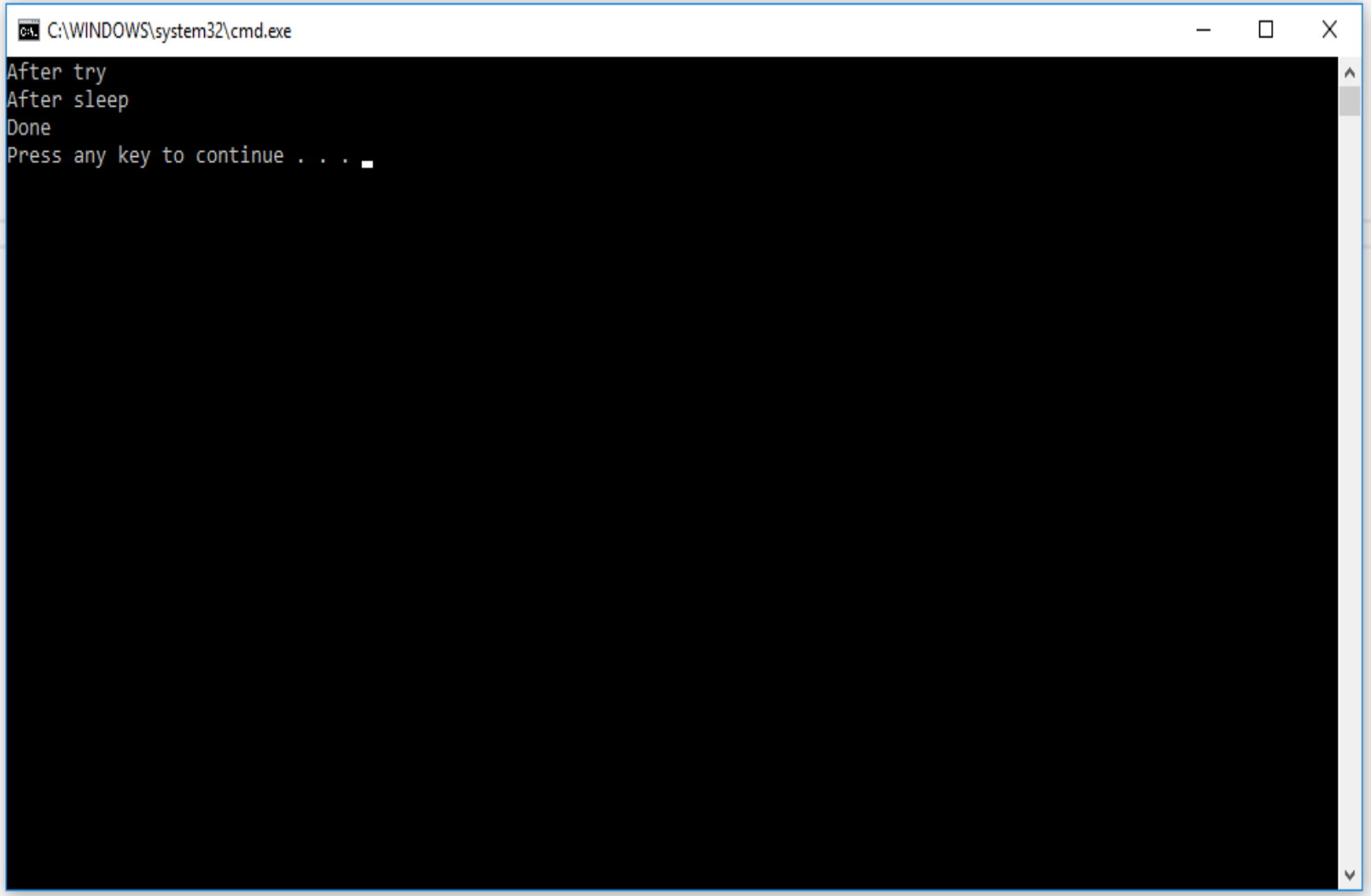
C:\WINDOWS\system32\cmd.exe

```
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");
        }
    }
}
```



Exceptions in other threads

Unhandled exception kills the application in most cases.

If it happens on a thread pool it is held until awaiting and then propagated if possible (thrown out of band for *async void*).

Catching unhandled exception with *AppDomain.CurrentDomain.UnhandledException* doesn't stop the application from terminating.

ThreadAbortException or *AppDomainUnloadedException* **do not** kill the application.

In .NET 1 it was different:

- Exception on a thread pool was printed to the console and the thread was returned to the pool.
- Exception in other thread was printed to the console and the thread was terminated.
- Exception on the finalizer was printed to the console and finalizer was still working.
- Exception on the main thread resulted in application termination.

Hijacking thread creation

Program.cs

ThreadExceptionHandler | ThreadExceptionHandler.Program | Main(string[] args)

```
102             }
103         }
104
105         public static ThreadStart ModifyHandler(object _, ThreadStart threadStart)
106         {
107             return () =>
108             {
109                 try
110                 {
111                     threadStart();
112                 }
113                 catch (Exception e)
114                 {
115                     Console.WriteLine("Handling!");
116                     Console.WriteLine(e);
117                 }
118             };
119         }
120     }
121
122     class Program
123     {
124         static void Main(string[] args)
125         {
126             ThreadHandler.EnableHandling();
127
128             MakeUnhandled();
129
130             Thread.Sleep(TimeSpan.FromSeconds(3));
131
132             Console.WriteLine("All done");
133         }
134
135         private static void MakeUnhandled()
136         {
137             ThreadStart lambda = () =>
138             {
139                 Console.WriteLine("Running in new thread");
140                 throw new Exception("This is unhandled!");
141             };
142             var thread = new Thread(lambda);
143             thread.Start();
144         }
145     }
146 }
147
```

C:\WINDOWS\system32\cmd.exe

```
Callback modifier caller: 30B24C0
Callback modifier: 1770888
Constructor: 72991A34
Running in new thread
Handling!
System.Exception: This is unhandled!
   at ThreadExceptionHandler.Program.<>c.<MakeUnhandled>b__1_0() in C:\Users\adafurma\Desktop\msp_windowsinternals\Threa
dExceptionHandler\Program.cs:line 140
   at ThreadExceptionHandler.ThreadHandler.<>c__DisplayClass6_0.<ModifyHandler>b__0() in C:\Users\adafurma\Desktop\msp_w
indowsinternals\ThreadExceptionHandler\Program.cs:line 111
All done
Press any key to continue . . .
```

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 . . .
```

Program.cs

GetAwaiterVsWait | GetAwaiterVsWait.Program | Handle()

```
1  using System;
2  using System.Threading.Tasks;
3
4  namespace GetAwaiterVsWait
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             Task.Run(async () => await Handle()).Wait();
11         }
12
13         static async Task Handle()
14         {
15             try
16             {
17                 Console.WriteLine("Using await Throw()");
18                 await Throw();
19                 Console.WriteLine("Using Throw().Wait();");
20                 Throw().Wait();
21                 Console.WriteLine("Using Throw().GetAwaiter().GetResult();");
22                 Throw().GetAwaiter().GetResult();
23             }
24             catch (Exception e)
25             {
26                 Console.WriteLine(e);
27             }
28         }
29
30         static async Task Throw()
31         {
32             await Task.Delay(100).ConfigureAwait(false);
33             throw new Exception("Throwing exception!");
34         }
35     }
36 }
37
```

Solution Explorer

Search Solution Explorer (Ctrl+;)

Solution 'WindowsInternals' (75 projects)

CatchAsyncVoid

Wątki

C:\WINDOWS\system32\cmd.exe

```
Using await Throw()
System.Exception: Throwing exception!
   at GetAwaiterVsWait.Program.<Throw>d__2.MoveNext() in C:\Users\adafurma\Desktop\msp_windowsinternals\GetAwaiterVsWait\Program.cs:line 33
--- 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 GetAwaiterVsWait.Program.<Handle>d__1.MoveNext() in C:\Users\adafurma\Desktop\msp_windowsinternals\GetAwaiterVsWait\Program.cs:line 18
Press any key to continue . . .
```

C:\WINDOWS\system32\cmd.exe

```
Using Throw().Wait();
System.AggregateException: One or more errors occurred. ---> System.Exception: Throwing exception!
   at GetAwaiterVsWait.Program.<Throw>d__2.MoveNext() in C:\Users\adafurma\Desktop\msp_windowsinternals\GetAwaiterVsWait\Program.cs:line 31
   --- 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 GetAwaiterVsWait.Program.<Handle>d__1.MoveNext() in C:\Users\adafurma\Desktop\msp_windowsinternals\GetAwaiterVsWait\Program.cs:line 18
---> (Inner Exception #0) System.Exception: Throwing exception!
   at GetAwaiterVsWait.Program.<Throw>d__2.MoveNext() in C:\Users\adafurma\Desktop\msp_windowsinternals\GetAwaiterVsWait\Program.cs:line 31<---

Press any key to continue . . .
```

C:\WINDOWS\system32\cmd.exe

```
Using Throw().GetAwaiter().GetResult();
System.Exception: Throwing exception!
   at GetAwaiterVsWait.Program.<Throw>d__2.MoveNext() in C:\Users\adafurma\Desktop\msp_windowsinternals\GetAwaiterVsWait\Program.cs:line 29
--- 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 GetAwaiterVsWait.Program.<Handle>d__1.MoveNext() in C:\Users\adafurma\Desktop\msp_windowsinternals\GetAwaiterVsWait\Program.cs:line 18
Press any key to continue . . .
```

UnobservedTaskException

UnobservedException | UnobservedException.Program | Test()

```
1  using System;
2  using System.Threading.Tasks;
3
4  namespace UnobservedException
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             TaskScheduler.UnobservedTaskException += TaskScheduler_UnobservedTaskException;
11             Test();
12
13             Console.ReadLine();
14
15             GC.Collect();
16             GC.WaitForPendingFinalizers();
17         }
18
19         static void TaskScheduler_UnobservedTaskException(object sender, UnobservedTaskExceptionEventArgs e)
20         {
21             Console.WriteLine("Unobserved exception!");
22             Console.WriteLine(e.Exception);
23         }
24
25         public static async Task Test()
26         {
27             throw new Exception();
28         }
29     }
30 }
```

C:\WINDOWS\system32\cmd.exe

```
Unobserved exception!
System.AggregateException: A Task's exception(s) were not observed either by Waiting on the Task or accessing its Except
ion property. As a result, the unobserved exception was rethrown by the finalizer thread. ---> System.Exception: Excepti
on of type 'System.Exception' was thrown.
   at UnobservedException.Program.<Test>d__2.MoveNext() in C:\Users\adafurma\Desktop\msp_windowsinternals\UnobservedExce
ption\Program.cs:line 27
   --- End of inner exception stack trace ---
---> (Inner Exception #0) System.Exception: Exception of type 'System.Exception' was thrown.
   at UnobservedException.Program.<Test>d__2.MoveNext() in C:\Users\adafurma\Desktop\msp_windowsinternals\UnobservedExce
ption\Program.cs:line 27<---

Press any key to continue . . .
```

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

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

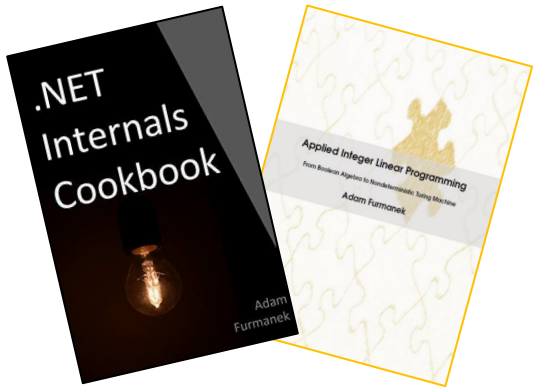
https://weblogs.asp.net/dixin/understanding-c-sharp-async-await-3-runtime-context — ExecutionContext internals

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