The State Machine

**I read somewhere that you can think of it like this:**

* Call "before" code
* Call await code as task
* Store "after" code in a continuation task

But this isn’t really the case.

**What really happens is actually something like this:**

* **On compile time:**
  + A structcalled StateMachine is generated
  + Contains fields to save function local state
  + A moveNext function is created which holds the entire code
  + The code is fragmented by await calls into several cases (machine states)
  + A calling code which creates and initializes this machine replaces our async function code.
* **On Runtime:**
  + A task is created to run the machine code:
  + Local variables are “lifted” into the state machine as fields.
  + Code is run until await
  + A awaited function's task is run
  + Machine state is set to next state so next code fragment will run on wakeup
  + A wake-up event is scheduled
  + MoveNext function returns (Thread is released to do other stuff (update UI))
* **When wakeup call is issued by OS:**
  + Thread which handles the await continuation is called
  + CurrentSyncContext is used to pick the correct thread to run it on.
    - This behavior can be changed by using: await task.ConfigureAwait(false);
  + Next code segment is run since next state was set before yielding control
  + Another await is scheduled, [etc.]

See it written in the code

Two async functions decoded

Hide   Copy Code

*// AsyncAwait.Engine.Downloader*

public static async Task<string> DownloadHtmlAsyncTask(string url)

{

HttpClient httpClient = new HttpClient();

Debug.WriteLine("before await");

string result = await httpClient.GetStringAsync(url);

Debug.WriteLine("after await");

return result;

}

*// AsyncAwait.ViewModel.MainWndViewModel*

private async Task<string> DownloadWithUrlTrackingTaskAsync(string url)

{

Debug.WriteLine("before await1");

string Data = await DownloadHtmlAsyncTask(url);

Debug.WriteLine("before await2");

string Data1 = await DownloadHtmlAsyncTask(url);

Debug.WriteLine("the end.");

return Data;

}

The decompilation was done (using ilspy) with "decompile async methods" turned off.

First function

DownloadHtmlAsyncTask uses only one await call.

This is the calling code which initializes the state machine:

Hide   Copy Code

[DebuggerStepThrough, AsyncStateMachine(typeof(AsyncMethods.<DownloadHtmlAsyncTask>d\_\_0))]

public static Task<string> DownloadHtmlAsyncTask(string url)

{

AsyncMethods.<DownloadHtmlAsyncTask>d\_\_0 <DownloadHtmlAsyncTask>d\_\_;

<DownloadHtmlAsyncTask>d\_\_.url = url;

<DownloadHtmlAsyncTask>d\_\_.<>t\_\_builder = AsyncTaskMethodBuilder<string>.Create();

*//set initial machine state to -1*

<DownloadHtmlAsyncTask>d\_\_.<>1\_\_state = -1;

AsyncTaskMethodBuilder<string> <>t\_\_builder = <DownloadHtmlAsyncTask>d\_\_.<>t\_\_builder;

<>t\_\_builder.Start<AsyncMethods.<DownloadHtmlAsyncTask>d\_\_0>(ref <DownloadHtmlAsyncTask>d\_\_);

return <DownloadHtmlAsyncTask>d\_\_.<>t\_\_builder.get\_Task();

}

The state machine definition: (**my comments inline**)

Hide   Shrink http://www.codeproject.com/images/arrow-up-16.png   Copy Code

using ConsoleApplication1;

using System;

using System.Diagnostics;

using System.Net.Http;

using System.Runtime.CompilerServices;

using System.Runtime.InteropServices;

[CompilerGenerated]

[StructLayout(LayoutKind.Auto)]

private struct <DownloadHtmlAsyncTask>d\_\_0 : IAsyncStateMachine

{

*//initial state is set to -1 by the machine's creation code*

public int <>1\_\_state;

public AsyncTaskMethodBuilder<string> <>t\_\_builder;

public string url;

public HttpClient <httpClient>5\_\_1;

public string <result>5\_\_2;

private TaskAwaiter<string> <>u\_\_$awaiter3;

private object <>t\_\_stack;

void IAsyncStateMachine.MoveNext()

{

string result;

try

{

int num = this.<>1\_\_state;

*//if (!Stop)*

if (num != -3)

{

TaskAwaiter<string> taskAwaiter;

*//machine starts with num=-1 so we enter*

if (num != 0)

{

*//first (+ initial) state code, run code before await is invoked*

this.<httpClient>5\_\_1 = new HttpClient();

Debug.WriteLine("before await");

*//a task is invoked*

taskAwaiter = this.<httpClient>5\_\_1.GetStringAsync(this.url).GetAwaiter();

*//[performance] check if this task has completed already,*

*//if it did, skip scheduling and boxing*

if (!taskAwaiter.get\_IsCompleted())

{

this.<>1\_\_state = 0;

this.<>u\_\_$awaiter3 = taskAwaiter;

*//Schedules the state machine to proceed*

*//to the next action when the specified awaiter completes.*

*//Also: sending this state machine here will trigger it's boxing into heap.*

this.<>t\_\_builder.AwaitUnsafeOnCompleted<TaskAwaiter<string>,

AsyncMethods.<DownloadHtmlAsyncTask>d\_\_0>(ref taskAwaiter, ref this);

*//release cpu knowing our next step will be called by Framework.*

return;

}

}

else

{

*//set awaiter to null*

taskAwaiter = this.<>u\_\_$awaiter3;

this.<>u\_\_$awaiter3 = default(TaskAwaiter<string>);

*//set state to initial state (temporarily)*

this.<>1\_\_state = -1;

}

*//second (+ final) state code (state=0): set result to member, printout*

string arg\_A5\_0 = taskAwaiter.GetResult();

*//set awaiter to null*

taskAwaiter = default(TaskAwaiter<string>);

*//set StateMachine's result field, and end code (print out)*

string text = arg\_A5\_0;

this.<result>5\_\_2 = text;

Debug.WriteLine("after await");

*//set return task's result*

result = this.<result>5\_\_2;

}

}

*//exception handling is done here*

catch (Exception exception)

{

*//set machine state to final state*

this.<>1\_\_state = -2;

this.<>t\_\_builder.SetException(exception);

return;

}

this.<>1\_\_state = -2;

this.<>t\_\_builder.SetResult(result);

}

[DebuggerHidden]

void IAsyncStateMachine.SetStateMachine(IAsyncStateMachine param0)

{

this.<>t\_\_builder.SetStateMachine(param0);

}

}

* State machine is created as a struct, which means it's on stack
* It is moved to heap inside the AwaitUnsafeOnCompleted function (deeper into the mechanism it is cast to IAsyncStateMachine which triggers the boxing).
* This call may be skipped if the task we await ends before the taskAwaiter.IsCoplete is checked.

This function has only two code fragments (two states) since there is only one await dividing the code, it may be a standard scenario but not an interesting one.

What happens when the StateMachine has more states? this brings us to the second example...

Second function

**DownloadHtmlAsyncTask - using two await calls:**

The calling code:

Hide   Copy Code

[DebuggerStepThrough, AsyncStateMachine(typeof(AsyncMethods.<DownloadWithUrlTrackingTaskAsync>d\_\_5))]

private Task<string> DownloadWithUrlTrackingTaskAsync(string url)

{

AsyncMethods.<DownloadWithUrlTrackingTaskAsync>d\_\_5 <DownloadWithUrlTrackingTaskAsync>d\_\_;

<DownloadWithUrlTrackingTaskAsync>d\_\_.<>4\_\_this = this;

<DownloadWithUrlTrackingTaskAsync>d\_\_.url = url;

<DownloadWithUrlTrackingTaskAsync>d\_\_.<>t\_\_builder = AsyncTaskMethodBuilder<string>.Create();

*//set initial machine state to -1*

<DownloadWithUrlTrackingTaskAsync>d\_\_.<>1\_\_state = -1;

AsyncTaskMethodBuilder<string> <>t\_\_builder =

<DownloadWithUrlTrackingTaskAsync>d\_\_.<>t\_\_builder;

<>t\_\_builder.Start<AsyncMethods.<DownloadWithUrlTrackingTaskAsync>d\_\_5>(ref <DownloadWithUrlTrackingTaskAsync>d\_\_);

return <DownloadWithUrlTrackingTaskAsync>d\_\_.<>t\_\_builder.get\_Task();

}

The state machine definition: (**read the comments in the code**)

Hide   Shrink http://www.codeproject.com/images/arrow-up-16.png   Copy Code

using ConsoleApplication1;

using System;

using System.Diagnostics;

using System.Runtime.CompilerServices;

using System.Runtime.InteropServices;

[CompilerGenerated]

[StructLayout(LayoutKind.Auto)]

private struct <DownloadWithUrlTrackingTaskAsync>d\_\_5 : IAsyncStateMachine

{

public int <>1\_\_state;

public AsyncTaskMethodBuilder<string> <>t\_\_builder;

public AsyncMethods <>4\_\_this;

public string url;

public string <Data>5\_\_6;

public string <Data1>5\_\_7;

private TaskAwaiter<string> <>u\_\_$awaiter8;

private object <>t\_\_stack;

void IAsyncStateMachine.MoveNext()

{

string result;

try

{

TaskAwaiter<string> taskAwaiter;

*//initialy state = -1, so we skip this at start*

switch (this.<>1\_\_state)

{

case -3:

*//request machine to stop!*

*//machine will go to end state and set result if one exists.*

goto IL\_168;

case 0:

taskAwaiter = this.<>u\_\_$awaiter8;

this.<>u\_\_$awaiter8 = default(TaskAwaiter<string>);

*//set state to initial state (temporarily)*

this.<>1\_\_state = -1;

goto IL\_A1;

case 1:

taskAwaiter = this.<>u\_\_$awaiter8;

this.<>u\_\_$awaiter8 = default(TaskAwaiter<string>);

*//set state to initial state (temporarily)*

this.<>1\_\_state = -1;

goto IL\_121;

}

*// first state code state=-1,*

*//printout, and await, then return control to scheduler*

Debug.WriteLine("before await1");

taskAwaiter = AsyncMethods.DownloadHtmlAsyncTask(this.url).GetAwaiter();

if (!taskAwaiter.get\_IsCompleted())

{

*//set state to next step (0)*

this.<>1\_\_state = 0;

this.<>u\_\_$awaiter8 = taskAwaiter;

this.<>t\_\_builder.AwaitUnsafeOnCompleted<TaskAwaiter<string>,

AsyncMethods.<DownloadWithUrlTrackingTaskAsync>d\_\_5>(ref taskAwaiter, ref this);

return;

}

IL\_A1:

*//second state code (state = 0), set the result of the first call to member,*

*//printout, schedule next await, yield control*

string arg\_B0\_0 = taskAwaiter.GetResult();

taskAwaiter = default(TaskAwaiter<string>);

string text = arg\_B0\_0;

this.<Data>5\_\_6 = text;

Debug.WriteLine("before await2");

taskAwaiter = AsyncMethods.DownloadHtmlAsyncTask(this.url).GetAwaiter();

if (!taskAwaiter.get\_IsCompleted())

{

*//set state to next step (1)*

this.<>1\_\_state = 1;

this.<>u\_\_$awaiter8 = taskAwaiter;

this.<>t\_\_builder.AwaitUnsafeOnCompleted<TaskAwaiter<string>,

AsyncMethods.<DownloadWithUrlTrackingTaskAsync>d\_\_5>(ref taskAwaiter, ref this);

return;

}

IL\_121:

*//third state code (state = 1),*

*//set the result of the first call to member, printout,*

*//set the (function's) return task's result.*

string arg\_130\_0 = taskAwaiter.GetResult();

taskAwaiter = default(TaskAwaiter<string>);

text = arg\_130\_0;

this.<Data1>5\_\_7 = text;

Debug.WriteLine("the end.");

result = this.<Data>5\_\_6;

}

catch (Exception exception)

{

*//some exception handling: set end state (-2)*

this.<>1\_\_state = -2;

*//set the exception in the builder*

this.<>t\_\_builder.SetException(exception);

return;

}

IL\_168:

*//if no exception set end state and result*

this.<>1\_\_state = -2;

this.<>t\_\_builder.SetResult(result);

}

[DebuggerHidden]

void IAsyncStateMachine.SetStateMachine(IAsyncStateMachine param0)

{

this.<>t\_\_builder.SetStateMachine(param0);

}

}

Points of Interest

* The overhead:
  + Anyone can see that there are "several" extra code lines added.
  + But according to Microsoft:
    - About 40 operations are used to create async state machine (microseconds).
    - This overhead is similar or less than the other older async mechanisms.
    - This can be a problem only when calling many awaits in a tight loop.
  + Besides that, with today's CPUs who will notice the overhead when it's run on a non-GUI thread? Or even better - on the HDD controller...
* Exception handling
  + Task.WhenAll will throw any exception from any of the tasks it aggregates.
  + This means that the first exception will be thrown out.
  + The task.Exception property will contain all the exceptions inside an AggregateException(you can use Task.Wait() instead of await)

**Reference :** <http://www.codeproject.com/Articles/535635/Async-Await-and-the-Generated-StateMachine>