

ASYNC/AWAIT

MODULE 2

October 2015

Life is Asynchronous!

- Fill pot with water
- Put pot on stove
- Start boiling water (`async`)
- Do dishes
- When water is boiling... (`await`)
- Put spaghetti in boiling water (`async`)
- Start warming up pasta sauce (`async`)
- Make salad
- Pour drinks
- When spaghetti and sauce finished... (`await`)
- Make plates of spaghetti with sauce and side salad



Synchronous Cooking!

- Fill pot with water
- Put pot on stove
- Wait for water to boil (just stare at it)
- Put spaghetti in pot
- Wait for spaghetti to cook (stare harder)
- Strain spaghetti
- Fill another pan with pasta sauce
- Put pasta sauce on stove
- Wait for pasta sauce to warm up (keep staring...maybe it helps?)
- Make salad
- Pour drinks
- Make plates of spaghetti with sauce and side salad



Task Parallel Library

Highlights

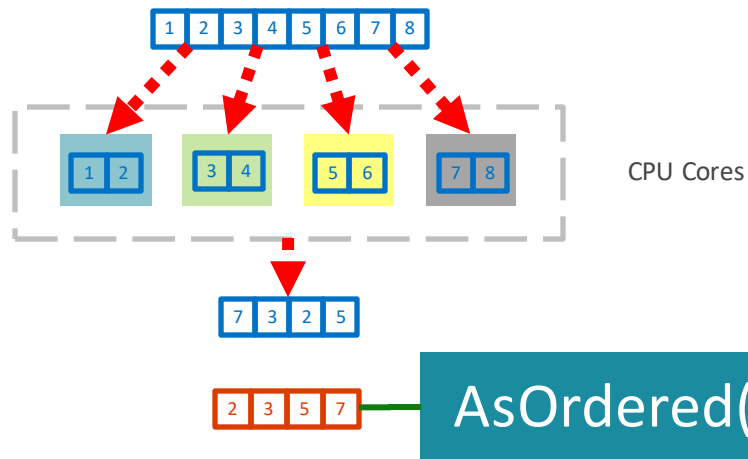
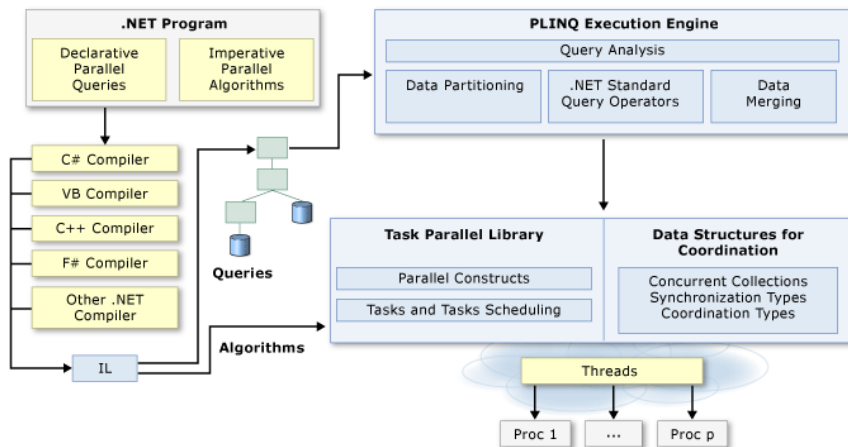
- New in .NET 4.0 (Visual Studio 2010)
- High level: We talk about Tasks, not Threads.
- New mechanisms for CPU-bound and IO-bound code (PLINQ, Parallel class, Task class)
- Cancellations with token, Continuations and Synchronization between contents

Task class

- Code that can be executed asynchronously
- In another thread? It doesn't matter...

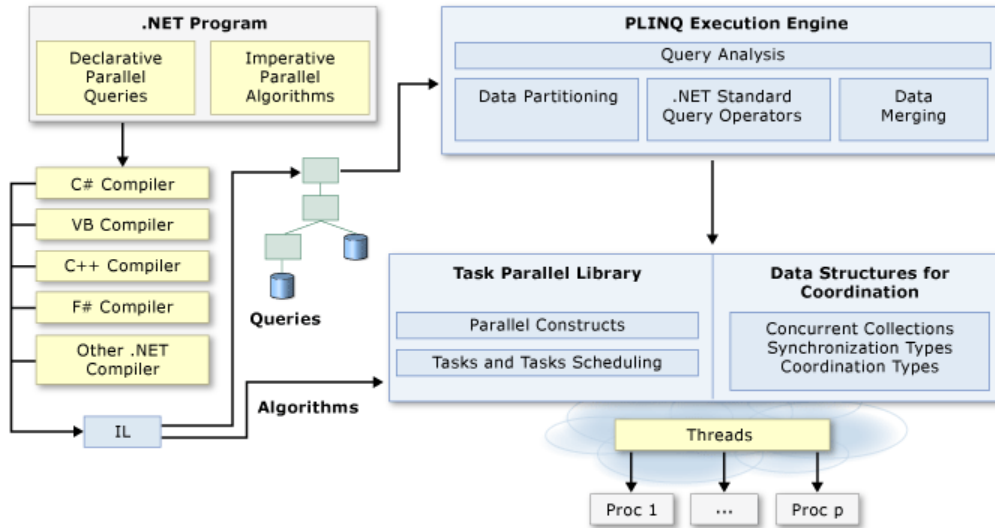
Task Parallel Library - PLINQ

```
var numbers = Enumerable.Range(1, 10000000);  
var query = numbers.AsParallel().Where(n => n.IsPrime());  
var primes = query.ToArray();
```



Task Parallel Library – Parallel class

```
var customers = Customer.GetSampleCustomers();  
Parallel.ForEach(customers, c => {  
    if(!c.IsActive) c.Balance = 0;  
});
```



PLINQ and Parallel are not Async!

Task Parallel Library - Task class

```
public static List<string> GetNetworkSQLServerInstances() {  
    //Get local network servers  
    return servers;  
}  
  
private void updateServersList(List<string> servers) {  
    listBox1.Items.AddRange(servers.ToArray());  
}  
  
//SYNC version  
private void Button1_Click(object sender, EventArgs e) {  
    var servers = GetNetworkSQLServerInstances();  
    updateServersList(servers);  
}
```

Task Parallel Library - Task class

```
public static List<string> GetNetworkSQLServerInstances() {  
    //Get local network servers  
    return servers;  
}  
  
private void updateServersList(List<string> servers) {  
    listBox1.Items.AddRange(servers.ToArray());  
}  
  
//ASYNC version  
private void Button1_Click(object sender, EventArgs e) {  
    var serversTask = Task.Factory.StartNew(() => GetNetworkSQLServerInstances());  
    serversTask.ContinueWith(t => updateServersList(serversTask.Result));  
}
```


Task Parallel Library - Task class

```
public static List<string> GetNetworkSQLServerInstances() {  
    //Get local network servers  
    return servers;  
}  
  
private void updateServersList(List<string> servers) {  
    listBox1.Items.AddRange(servers.ToArray());  
}  
  
//ASYNc version + context synchronization  
private void Button1_Click(object sender, EventArgs e) {  
    var serversTask = Task.Factory.StartNew(() => GetNetworkSQLServerInstances());  
    serversTask.ContinueWith(t => updateServersList(serversTask.Result),  
        TaskScheduler.FromCurrentSynchronizationContext());  
}
```

async/await

```
public static List<string> GetNetworkSQLServerInstances() {  
    //Get local network servers  
    return servers;  
}  
  
private void updateServersList(List<string> servers) {  
    listBox1.Items.AddRange(servers.ToArray());  
}  
  
//ASYNC/AWAIT version  
private async void Button1_Click(object sender, EventArgs e) {  
    var servers = await Task.Run(() => GetNetworkSQLServerInstances());  
    updateServersList(servers);  
}
```

Highlights

- Syntax sugar for invoking and chaining Tasks.

Pros

- Mega-Easy syntax
- Without callbacks
- Allows update UI

Cons

- More overhead than sync methods
- Typically the overhead is negligible...

async/await

```
//ASYNC/AWAIT version
private async void Button1_Click(object sender, EventArgs e) {
    var servers = await GetNetworkSQLServerInstancesAsync();
    updateServersList(servers);
}
```

async



FALSE

‘This method is asynchronous’



TRUE

‘In this method we will call asynchronous methods’

await



FALSE

‘Call this asynchronous method and wait until the method ends’



TRUE

‘Call this method and return control immediately to the caller,
when the method ends, continue the execution from that point’

What methods can be async?

Methods with the following return types can be made async

Task

Task<T>

void //but avoid it!

async Task

async Task<T>

async void

Why async/await?

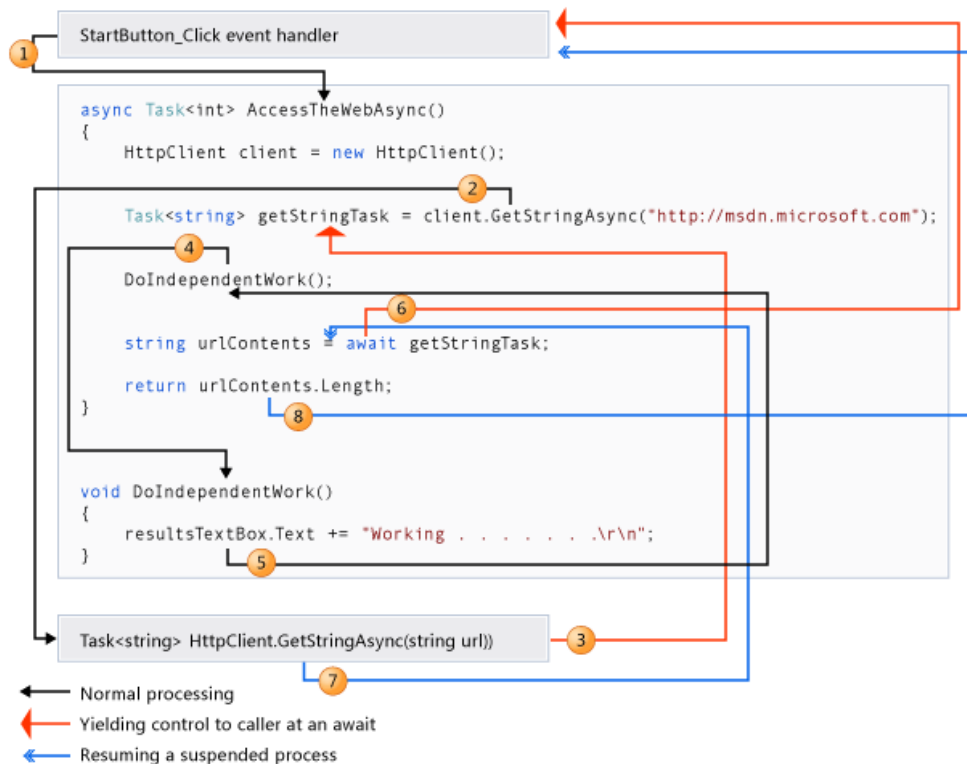
For network I/O

- Web service calls
- Database calls
- Cache calls
- Any call to any other server
 - Something else doing the work

Computationally intensive work using Task.Run (avoid in ASP.NET)

- It is doing the work

Control Flow



Async void is only for event handlers

Principles

- Async void is a “fire-and-forget” mechanism...
- The caller is *unable* to know when an async void has finished
- The caller is *unable* to catch exceptions thrown from an async void
 - (instead they get posted to the UI message-loop)

Guidance

- Use async void methods only for top-level event handlers (and their like)
- Use async Task-returning methods everywhere else
- If you need fire-and-forget elsewhere, indicate it explicitly e.g. “FredAsync().FireAndForget()”
- When you see an async lambda, verify it

SynchronizationContext

.NET applications have a synchronization context

It's different for each type of app, but fall into buckets

ASP.NET

- MVC
- WebAPI
- WebForms

UI

- WPF
- WinForms
- Windows Store app

Neither

- Console app

SynchronizationContext suggestions

If it's on the UI or in ASP.NET and you don't need the context...
don't continue on captured context

```
public async Task<string> GetUrlAsync(string url)
{
    string result = await GetUrlAsync(url,
CancellationToken.None).ConfigureAwait(false);
    return result;
}
```

SynchronizationContext suggestions

Don't use `.ConfigureAwait(false)` on endpoints and ASP.NET pipeline

- MVC controller actions
- WebAPI actions
- Filters
- HttpHandlers
- Http Message Handlers like DelegatingHandler

Use `.ConfigureAwait(false)` basically everywhere else

The Deadlock

```
async void button1_Click(...)
{
    DoWorkAsync().Wait();
}
```

1. DoWorkAsync invoked on UI thread

4. UI blocks waiting for DoWorkAsync-returned Task to complete

6. UI thread still blocked waiting for async operation to complete. Deadlock!

```
async Task DoWorkAsync()
{
    await Task.Run(...);
    Console.WriteLine("Done task");
}
```

3. Await captures SynchronizationContext and hooks up a continuation to run when task completes

2. Task.Run schedules work to run on thread pool

5. Task.Run task completes on pool & invokes continuation which Posts back to UI thread


All async tasks run **HOT**

- As soon as task created, method is started (like regular method)

Exceptions

```
private async Task GetMultipleErrorsAsync()
{
    var t1 = ThrowErrorCoreAsync();
    var t2 = ThrowErrorCoreAsync2();
    var t3 = ThrowErrorCoreAsync3();

    var tAll = Task.WhenAll(t1, t2, t3);
    try
    {
        await tAll;
    }
    catch (Exception ex)
    {
        Debugger.Break();
    }
}
```



Exception thrown is error from t1, but tAll has AggregateException of all 3 errors

Summary

- Long process
- Instead of Task
- Cache Tasks - NOT Data
- Performance - await state
- NOT Task.Wait()
 - SynchronizationContext
- DLL -> ConfigureAwait(false) ?
- Exceptions

EXAMPLE