ASYNCHRONOUS PROGRAMMING

.NET WAY

ASYNCHRONY
I Am Bit Tricky To Catch

- Trendy
- Examples: AJAX, AMD etc. and numerous language-level/addon implementations
- Languages supporting async patterns:

  C#: Tasks,

  Java7-8: Future and CompletableFuture

  Go: Routines and channels,

  Erlang: Processes and spawn

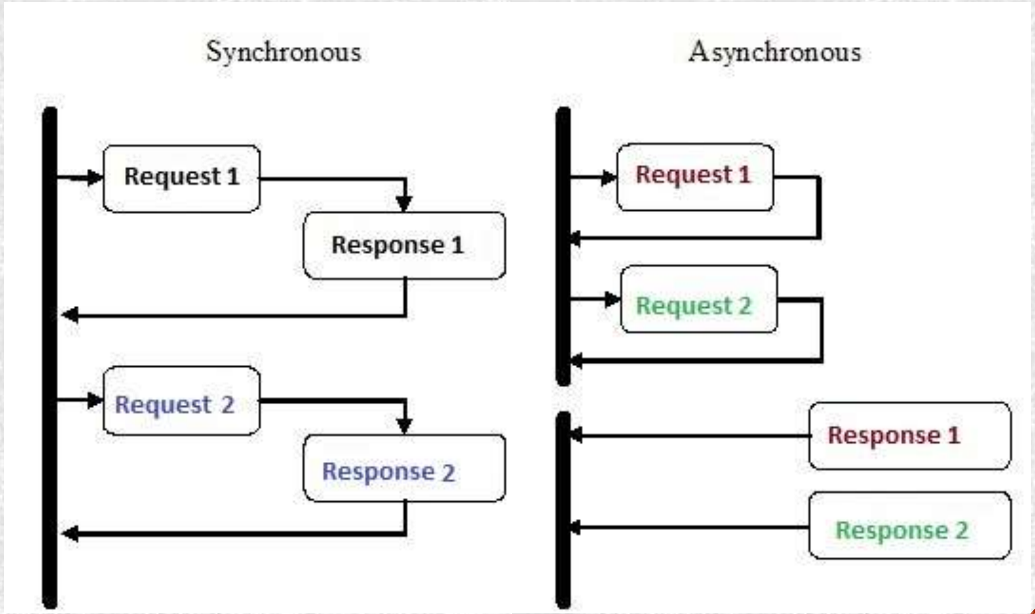
  Python: tasks, coroutines (asyncio)

  JavaScript (ES2017-ES8): Promises

  and list goes on.

ASYNCHRONY
I'll Call You Back

- Occurrence of events independent of the **main program** flow
- Enables this "main program" continue its work, not **blocking to wait** for event results
- Implicit degree of parallelism



BENEFITS
I Am Not Just For UI Apps

1. Scalability and
2. Offloading (e.g. responsiveness, parallelism)

- Most client apps care about asynchrony for "Offloading" reasons
  - For store, desktop and phone apps: primary benefit is **responsiveness**
- Most server apps care about asynchrony for "Scalability" reasons
  - Node.js, ASP.NET Core are inherently asynchronous, hence **scaling** is their key.

Common Question: Does async have a place on the server?

GOAL
I Am About To Discussed More?

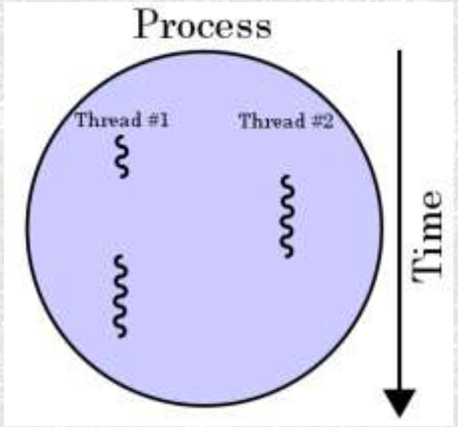
- History of Asynchronous APIs on .NET
- Thread, Threadpool and Task
- Conceptual overview of request handling on ASP.NET
- I won’t be covering C# async-await API syntax

HISTORY
.Net And .Net Languages have Always Been Loving Async

- APM (.NET1.1) : Socket class
  - **Begin<MethodName>** and **End<MethodName>** methods
- EAP (.NET2) : e.g. BackgroundWorker class
  - ***<MethodName>Async*** methods
- Task and TPL (.NET4)
- Task-based Asynchronous Pattern (TAP) (.NET4.5)
  - Async and await
- Language level support on C# and VB (2012), F# (2010)
- TAP: Recommended asynchronous design pattern

THREAD
I Am An Action Lady

- First appears in IBM OS/360 in 1967
- Its actual OS-level thread
- Gives life to a **process**
- Each process has at least one thread (Main thread)
- Threads share their address space



CLR THREAD
You should hate new Thread()

- Freaking expensive: Memory and time overhead associated with them

  Thread =

  (Thread Kernel Object) x86-**700BYTE**, x64-**1240BYTE**

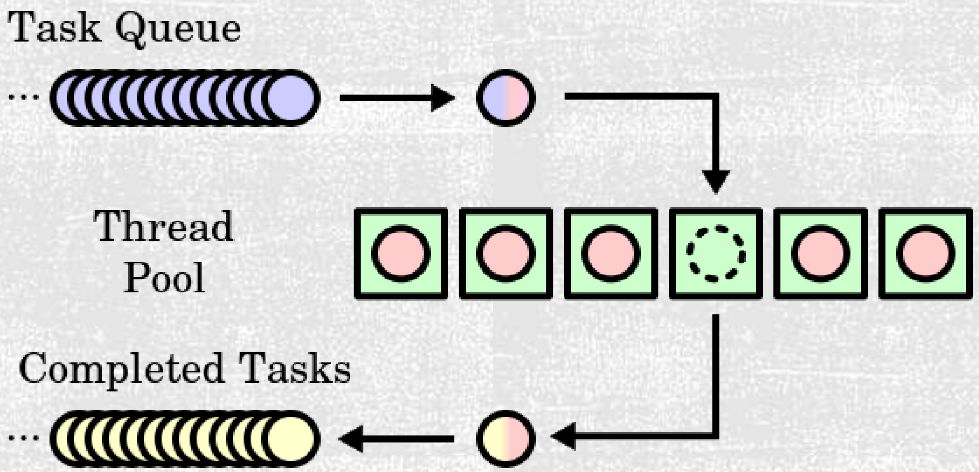
  + (Thread Environment Block) x86-**4KB**, x64-**8KB**
  + (User Mode Stack) x86, x64-**1MB**

  + (Kernel Mode Stack) x86-**12KB**, x64-**24KB**

- CLR thread directly maps to Windows thread
- Highest degree of control that programmers don't want ☺
- Only spun up new if heavy computations on multiple CPUs needs to be done.
- Foreground and background threads

THREADPOOL
I manage threads for you, sir!

- Thread pool is a set of threads available readily and maintained by the CLR.
- No control but pool size: Submit work to execute, wait for good to happen ☺
- Best for large no. of short operations where the caller does not need the result.



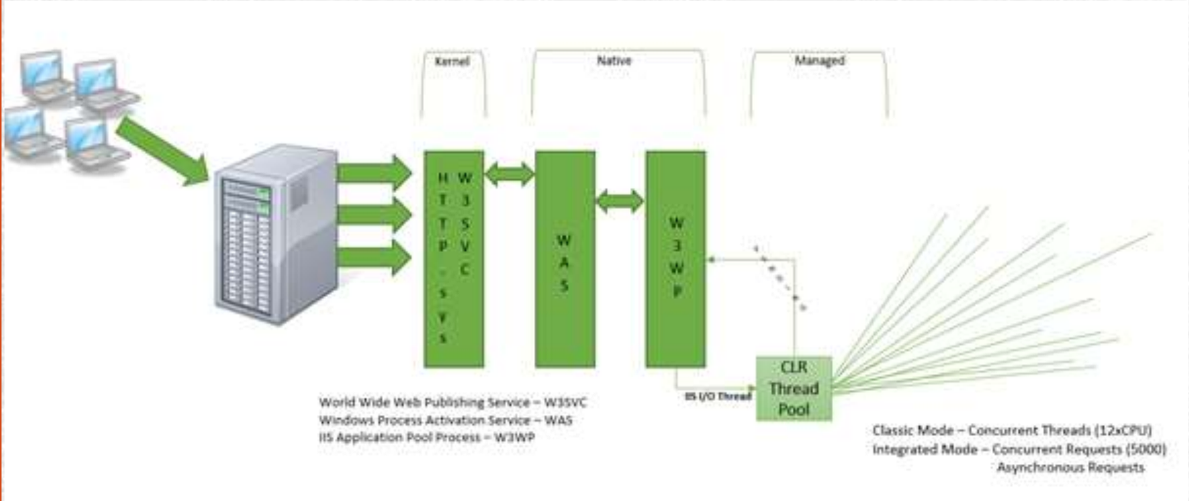
CONTINUED…

- Threadpool size (no. of threads)
  - Default minimum = No. of processors
  - Default maximum = 5000 (.NET4.5)
- Thread Injection: Starvation avoidance and hill-climbing algorithm
- Threads are added or removed every 500ms
- Thread pool categorizes its threads as
  - **Worker threads**
  - **I/O completion port (IOCP) threads**

IIS: REQUEST HANDLING

Web Dev? You Need To Understand Me

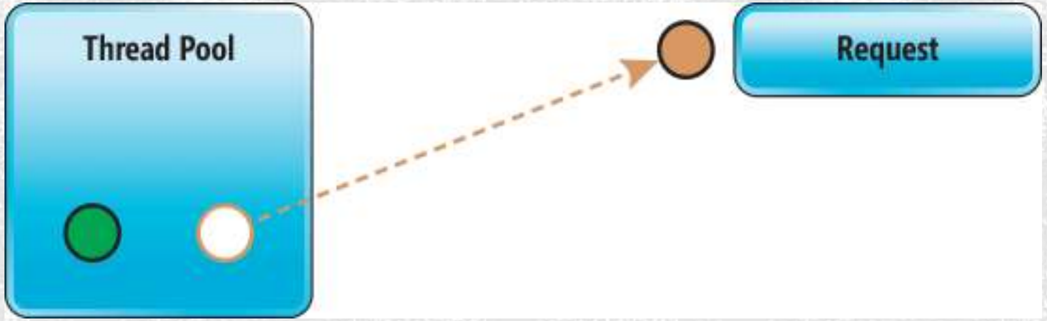
- An app/web server
- Kernel mode and user mode (Native mode)
- App pool: Grouping of URLs that is routed to one or more worker processes.


Kernel
Native
Managed
HTTP.sys
W3SVC
WAS
W3WP
IIS I/O Thread
CLR Thread Pool
World Wide Web Publishing Service – W3SVC
Windows Process Activation Service – WAS
IIS Application Pool Process – W3WP
Classic Mode – Concurrent Threads (12xCPU)
Integrated Mode – Concurrent Requests (5000)
Asynchronous Requests

OLD SYNCHRONOUS WAY

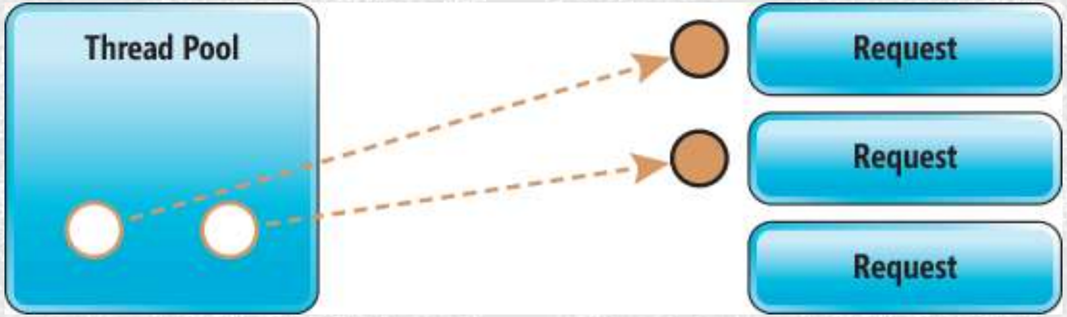
You Devs Love Me, Don't You?

- ASP.NET takes one of its thread pool threads and assigns it to just arrived request
- Request handler call that external resource synchronously and blocks it until result returns



BAD PART
All Busy, Try Later

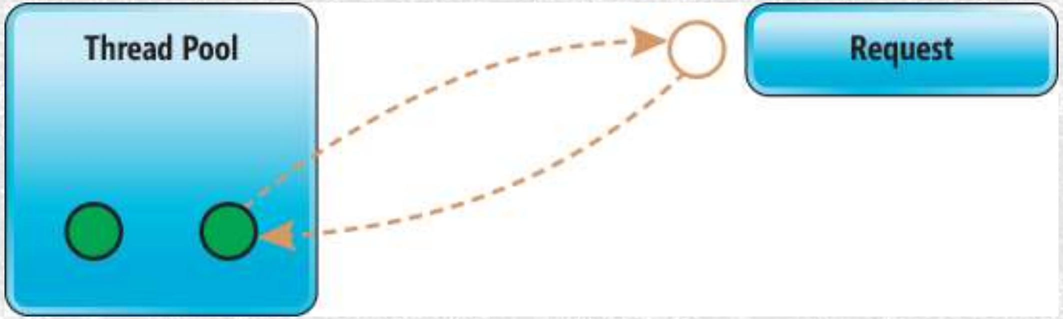
- Thread count saturation
- Available threads blocked and wasted
- New request **have to wait** and in **danger of 503**



ASYNCHRONOUS WAY

Don't Trust Me? Try Then

- Async don't waste precious threadpool threads
- Server could cope new request easily
- Smaller number of threads to handle a larger number of requests.



THREADPOOL SIZE
Just Increase Me And Forget Async Altogether. I Do Joke Too ☺

- Async does not replace the thread pool, rather makes optimum use of it
- Scales both further and faster than blocking threadpool threads
- Less memory pressure
- Can respond to sudden swings in request volume

Common question: *What About the Thread Doing the Asynchronous Work? There must something monitoring at it, right?*

No, not at all

TASKS
Some Call Me FUTURES Others PROMISES

- Best of both worlds
- **TaskScheduler**
  - Thread Pool Task Scheduler
  - Synchronization Context Task Scheduler
- **Task Factories**
- **Task Continuation, Progress and Cancellation**
- All newer high-level concurrency APIs are all built on Task.
- Task<T> **promises** to return us a T saying: "**not right now honey, I'm kinda busy, why don't you come back later? In fact, I will inform you when I am done or you may cancel me any time you want**"

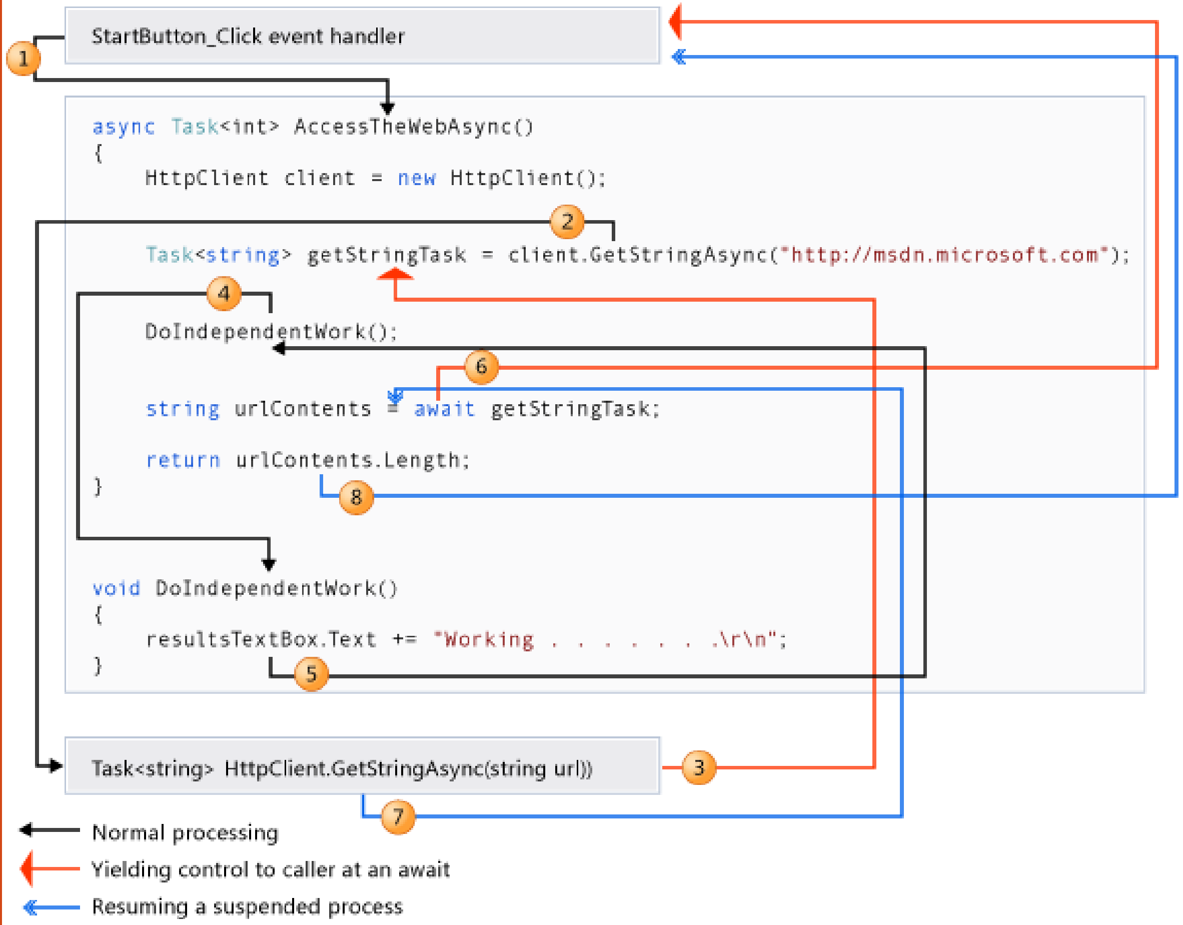
CPU-BOUND TASK
I Love CPU, Not You; Leave Me Alone

- **Async Don'ts**
  - **Too lightweight I/O (<30ms)**
  - **CPU-Intensive Operations**
- Historic problems with async:
  - Asynchronous code is difficult
  - Database backend is a bottleneck
- But today (past few years)
  - Bottleneck pushed back to app server

ASP.NET ASYNC PATTERNS Where Were You? As If You Cared? Always There, Just Bit Shy

- In Asp.Net since very beginning
  - Asynchronous Web pages introduced in ASP.NET 2.0
  - MVC got asynchronous controllers in ASP.NET MVC 2
- However, always been awkward to write and difficult to maintain
- Now, the tables have turned
  - In ASP.NET 4.5, async-await is savior
- More and more companies are embracing async and await on ASP.NET.

ASYNC FLOW
I Yield Control Back To The Caller

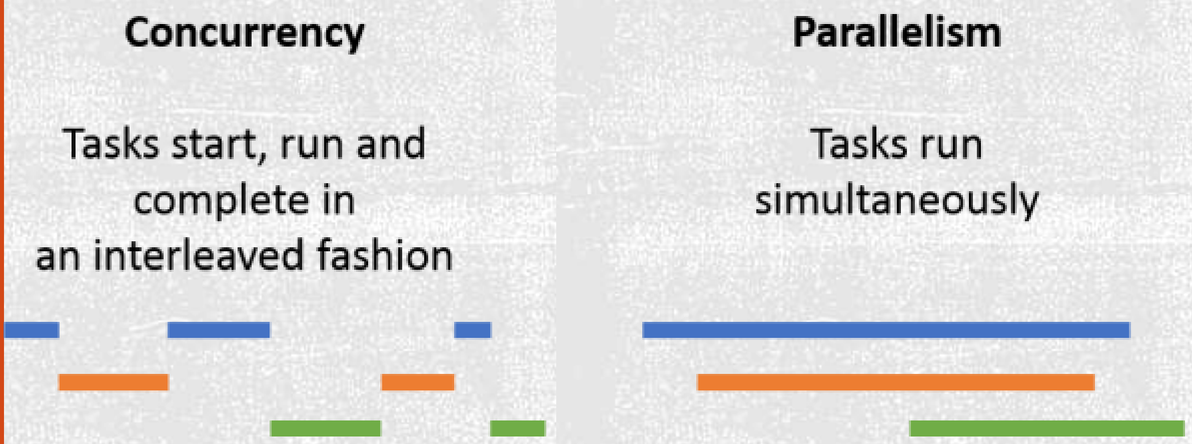


```
async Task<int> AccessTheWebAsync()
{
    HttpClient client = new HttpClient();

    Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");

    DoIndependentWork();

    string urlContents = await getStringTask;

    return urlContents.Length;
}

void DoIndependentWork()
{
    resultsTextBox.Text += "Working . . . . . . .\r\n";
}
```

StartButton_Click event handler
Task<string> HttpClient.GetStringAsync(string url))
1
2
3
4
5
6
7
8
Normal processing
Yielding control to caller at an await
Resuming a suspended process

NOT TO CONFUSE WITH TPL

We Sound Similar But Inherently Different

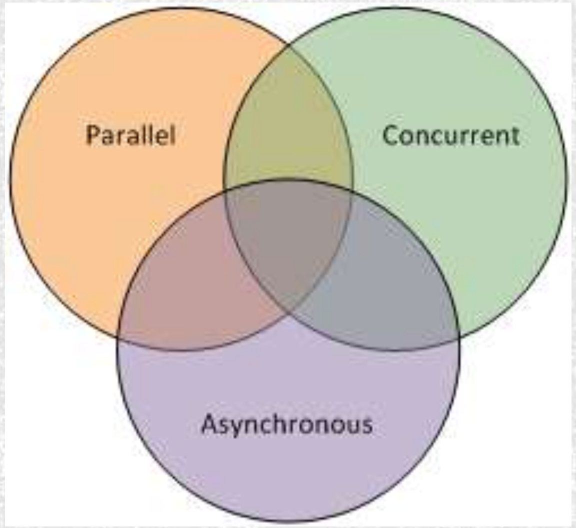


Concurrency

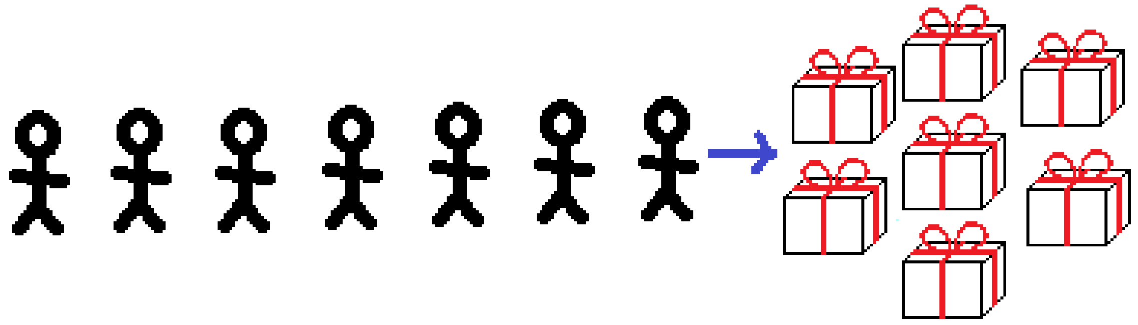
Tasks start, run and complete in an interleaved fashion

Parallelism

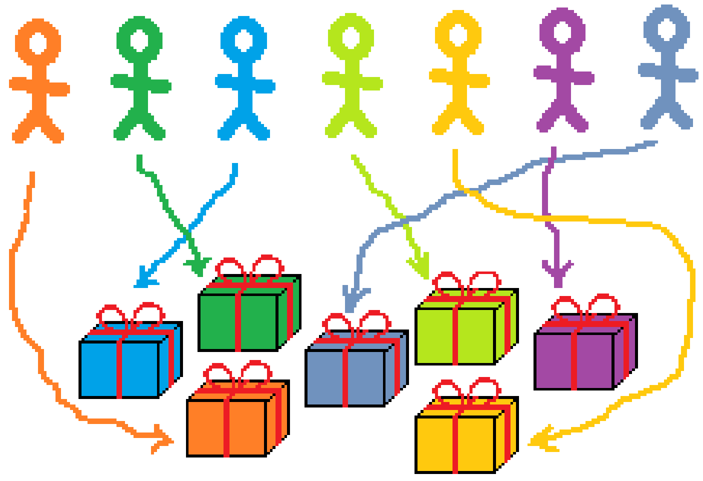
Tasks run simultaneously



TPL…

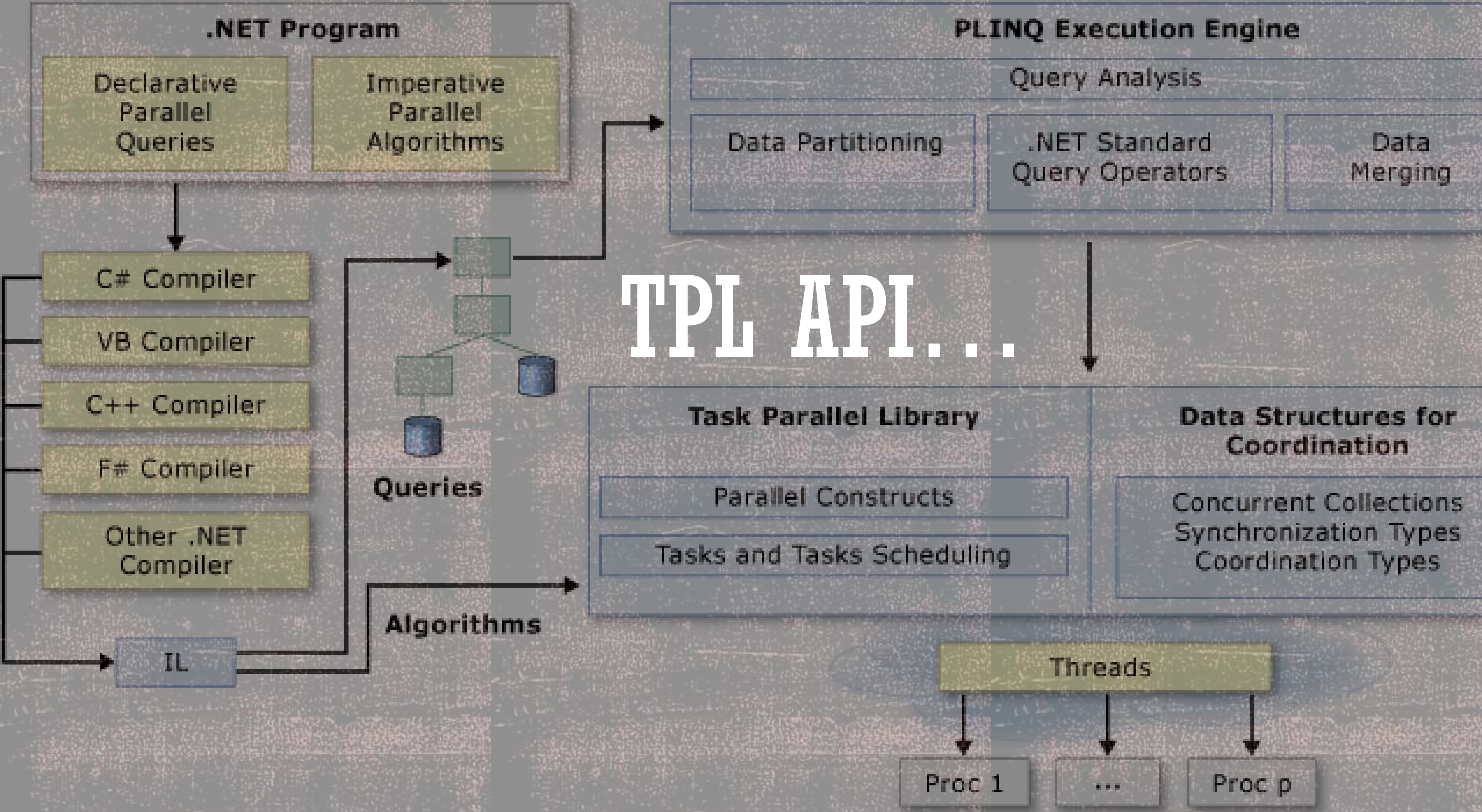


Concurrency: 7 kids queueing for presents from 1 heap



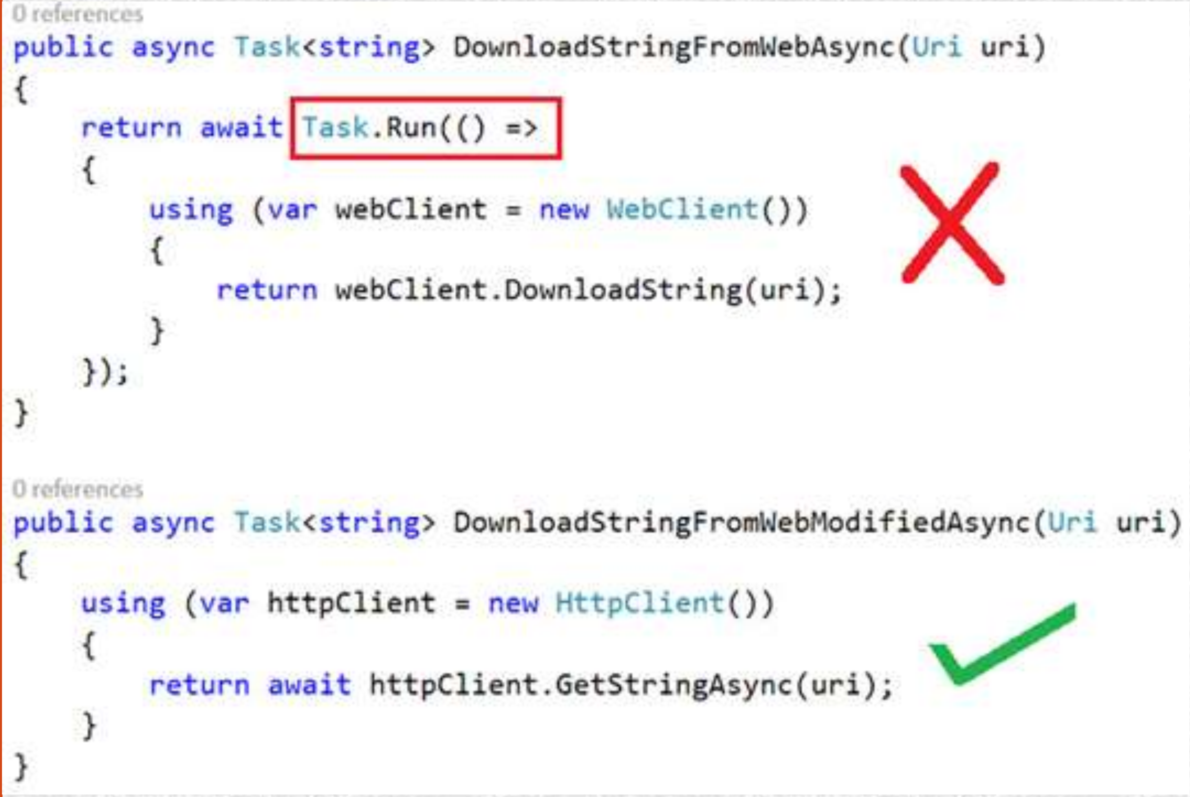
Parallelism: 7 kids getting 7 labeled presents, no queue

TPL API...


.NET Program
Declarative Parallel Queries
Imperative Parallel Algorithms
C# Compiler
VB Compiler
C++ Compiler
F# Compiler
Other .NET Compiler
IL
Queries
Algorithms
PLINQ Execution Engine
Query Analysis
Data Partitioning
.NET Standard Query Operators
Data Merging
Task Parallel Library
Parallel Constructs
Tasks and Tasks Scheduling
Data Structures for Coordination
Concurrent Collections
Synchronization Types
Coordination Types
Threads
Proc 1
...
Proc p

DEMO TIME...

ASYNC BEST PRACTICES
Be Clever Else You Are Busted



```
0 references
public async Task<string> DownloadStringFromWebAsync(Uri uri)
{
    return await Task.Run(() =>
    {
        using (var webClient = new WebClient())
        {
            return webClient.DownloadString(uri);
        }
    });
}

0 references
public async Task<string> DownloadStringFromWebModifiedAsync(Uri uri)
{
    using (var httpClient = new HttpClient())
    {
        return await httpClient.GetStringAsync(uri);
    }
}
```

Old	New	Description
task.Wait	await task	Wait/await for a task to complete
task.Result	await task	Get the result of a completed task
Task.WaitAny	await Task.WhenAny	Wait/await for one of a collection of tasks to complete
Task.WaitAll	await Task.WhenAll	Wait/await for every one of a collection of tasks to complete
Thread.Sleep	await Task.Delay	Wait/await for a period of time
Task constructor	Task.Run or TaskFactory.StartNew	Create a code-based task

ASYNC GUIDELINES

Problem	Solution
Create a task to execute code	Task.Run or TaskFactory.StartNew (not the Task constructor or Task.Start)
Create a task wrapper for an operation or event	TaskFactory.FromAsync or TaskCompletionSource<T>
Support cancellation	CancellationTokenSource and CancellationToken
Report progress	IProgress<T> and Progress<T>
Handle streams of data	TPL Dataflow or Reactive Extensions
Synchronize access to a shared resource	SemaphoreSlim
Asynchronously initialize a resource	AsyncLazy<T>
Async-ready producer/consumer structures	TPL Dataflow or AsyncCollection<T>

COMMON ASYNC PROBLEMS AND SOLUTIONS

REENTRANCY
Don't Exploit Me For God Sake Else My Curse Will Hurt You

- Reentering an asynchronous operation before it has completed
- Prevent reentrancy or it can cause unexpected results
  - Disable subsequent invokes until its done
  - Cancel and Restart operation
  - Run multiple operations and Queue the output

FINE TUNING

Want Precision And Flexibility To Your Async App? More APIs For You

- **CancellationToken**, **Task.WhenAll** and **Task.WhenAny**
- Start multiple tasks and await their completion by monitoring a single task.
- Use cases:
  - Cancel an Async Task or a List of Tasks
  - Cancel Async Tasks after a Period of Time
  - Cancel Remaining Async Tasks after One Is Complete
  - Start Multiple Async Tasks and Process Them As They Complete

INTERROGATIONS

BONUS READING...

EXAMPLE: HOW ASYNC SCALES?

- To understand why asynchronous requests scale, let’s trace a (simplified) example of an asynchronous I/O call. Let’s say a request needs to write to a file. The request thread calls the asynchronous write method. WriteAsync is implemented by the Base Class Library (BCL), and uses completion ports for its asynchronous I/O. So, the WriteAsync call is passed down to the OS as an asynchronous file write. The OS then communicates with the driver stack, passing along the data to write in an I/O request packet (IRP).
- This is where things get interesting: If a device driver can’t handle an IRP immediately, it must handle it asynchronously. So, the driver tells the disk to start writing and returns a “pending” response to the OS. The OS passes that “pending” response to the BCL, and the BCL returns an incomplete task to the request-handling code. The request-handling code awaits the task, which returns an incomplete task from that method and so on. Finally, the request-handling code ends up returning an incomplete task to ASP.NET, and the request thread is freed to return to the thread pool.

CONTINUED…

- Now, consider the current state of the system. There are various I/O structures that have been allocated (for example, the Task instances and the IRP), and they're all in a pending/incomplete state. However, there's no thread that is blocked waiting for that write operation to complete. Neither ASP.NET, nor the BCL, nor the OS, nor the device driver has a thread dedicated to the asynchronous work.
- When the disk completes writing the data, it notifies its driver via an interrupt. The driver informs the OS that the IRP has completed, and the OS notifies the BCL via the completion port. A thread pool thread responds to that notification by completing the task that was returned from WriteAsync(); this in turn resumes the asynchronous request code.
- Yes, there were a few threads "borrowed" for very short amounts of time during this completion-notification phase, but no thread was actually blocked while the write was in progress.

CONTINUED…

- Above example is drastically simplified, but it gets across the primary point: no thread is required for true asynchronous work. No CPU time is necessary to actually push the bytes out.
- At the device driver level, all non-trivial I/O is asynchronous. Many developers have a mental model that treats the "natural API" for I/O operations as synchronous, with the asynchronous API as a layer built on it. However, that's completely backward: in fact, the natural API is asynchronous; and it's the synchronous APIs that are implemented using asynchronous I/O.