BRIEF HISTORY OF (.NET) THREADING

David Mohundro

@drmohundro

HI...

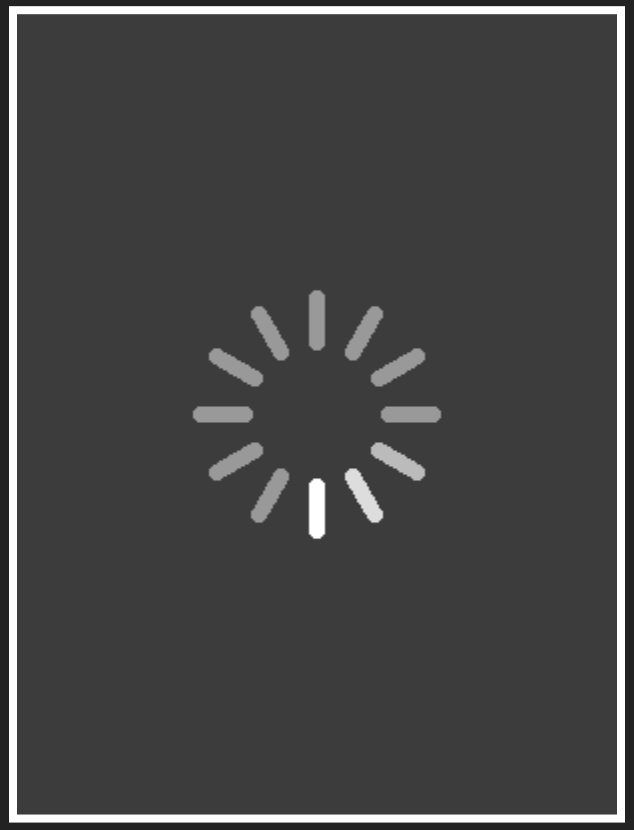
I'm David Mohundro

Clear Function

OUTLINE

- Prehistory
- Historical Threading
- Modern Threading

BUT, BEFORE WE START...



WHY THREADING?

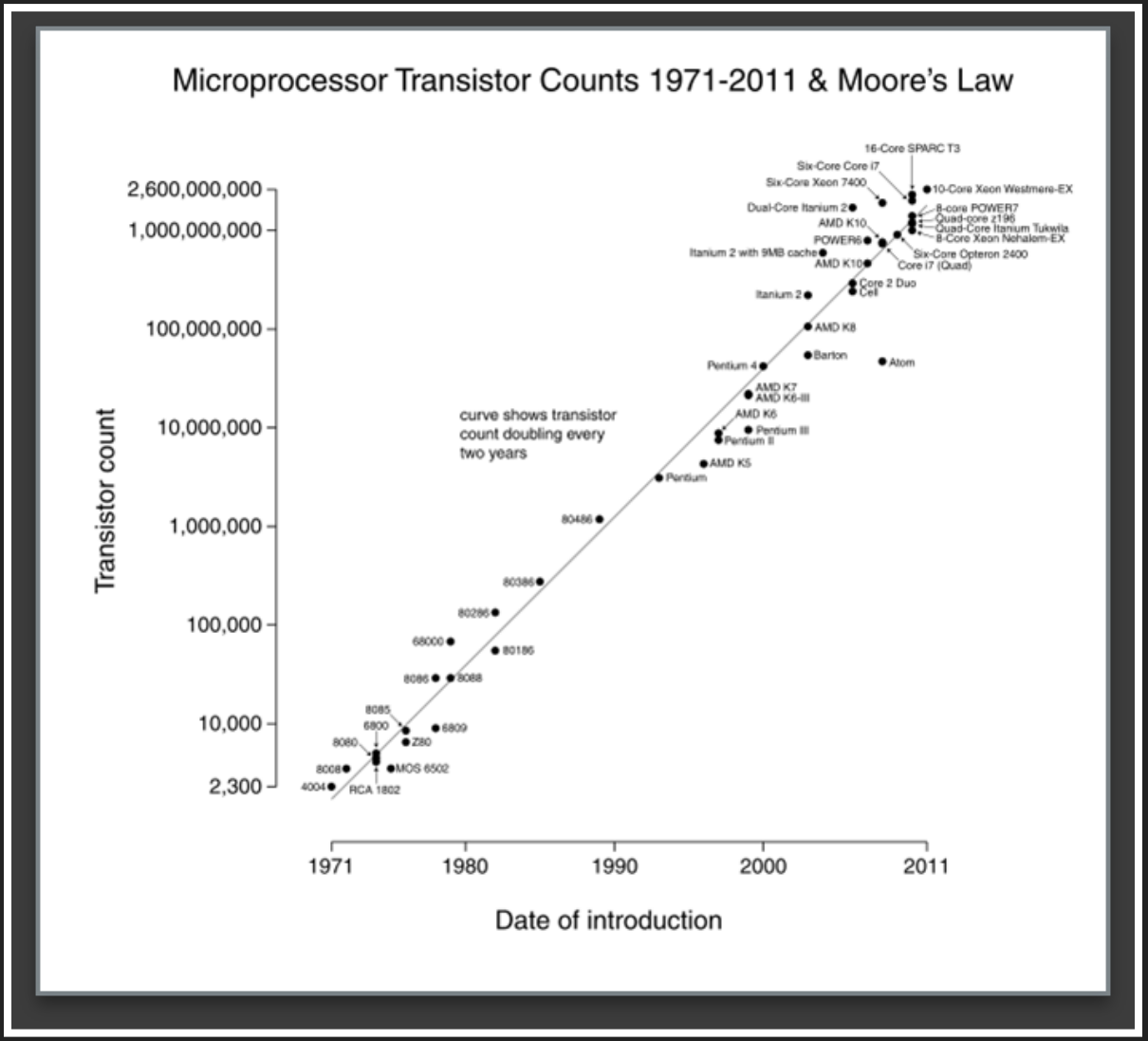
WHAT DOES IT SOLVE?

(discuss)


PREHISTORY
(before .NET)

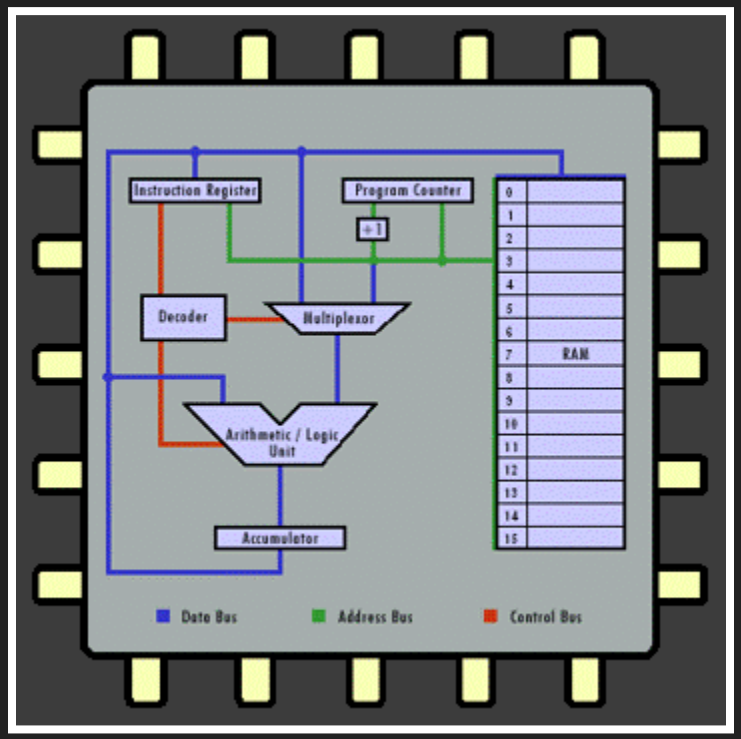
QUICK REVIEW... MOORE'S LAW

- 1965
- Number of transistors on integrated circuits doubles approximately every two years



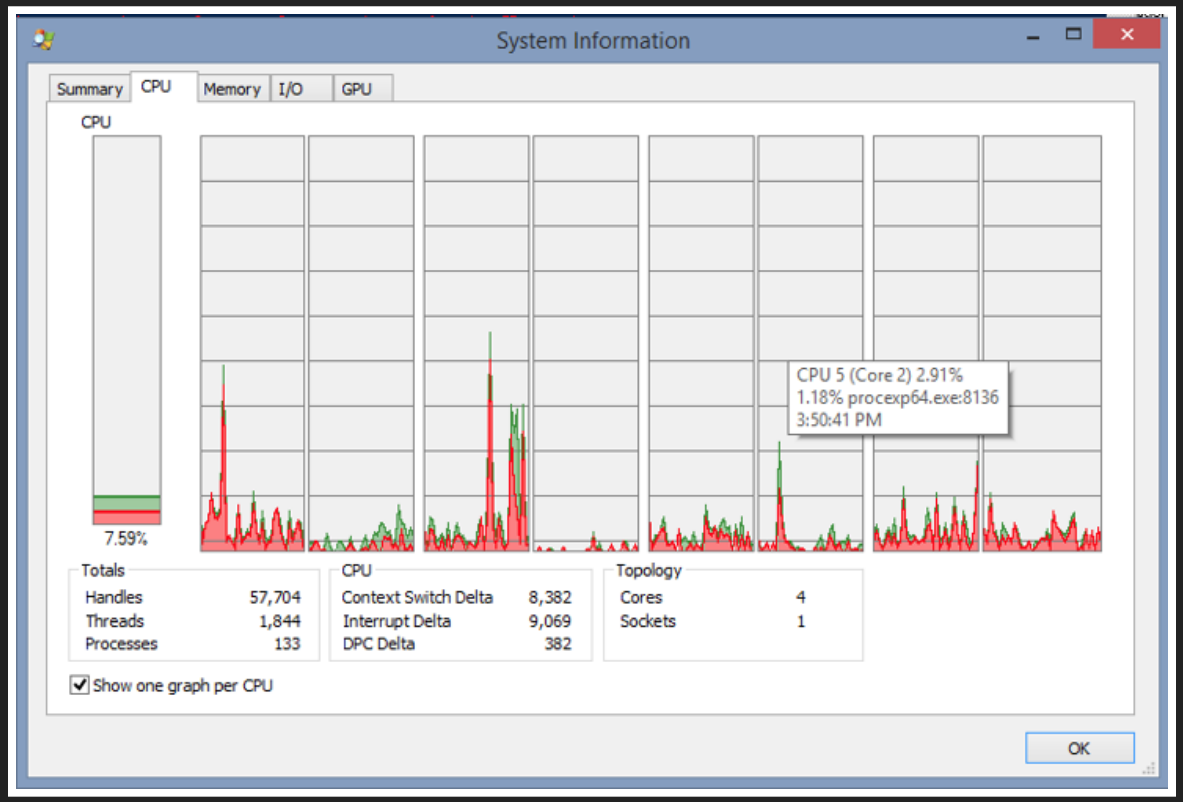
HOW DOES A CPU WORK?

- The ALU (Arithmetic Logic Unit) is the brain of the CPU
- As CPU architecture stands today, it can only do one thing at a time... it just does it very quickly



MULTITASKING

Multitasking is a method where multiple tasks, also known as processes, are performed during the same period of time

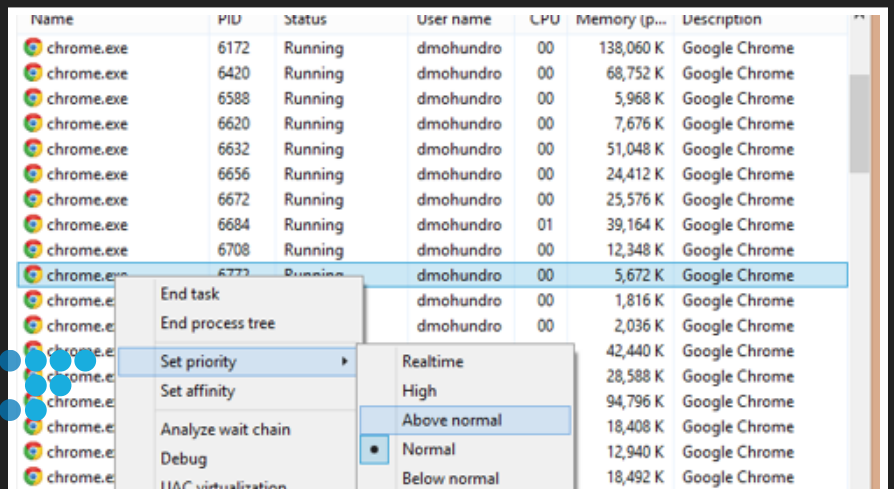


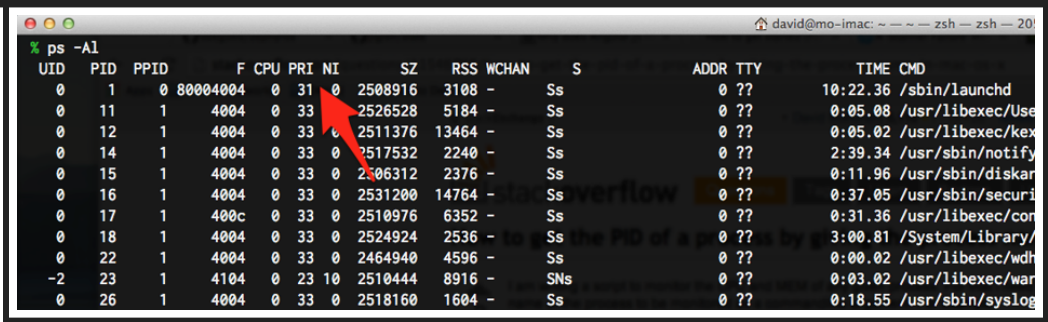
SCHEDULING

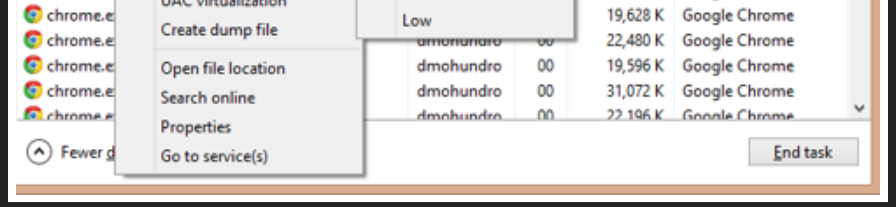
- Early DOS and Windows OS's weren't actually multitasked... at least not until Windows 3.1
  - Win3.1x used a non-preemptive scheduler
  - Win95 introduced a "rudimentary preemptive scheduler"
- Windows NT+, OSX, etc. all now use a multilevel feedback queue
  - Linux 2.6.23+, on the other hand, uses the Completely Fair Scheduler (CFS)

MULTILEVEL FEEDBACK QUEUES

What Windows uses today (and almost every other modern OS)





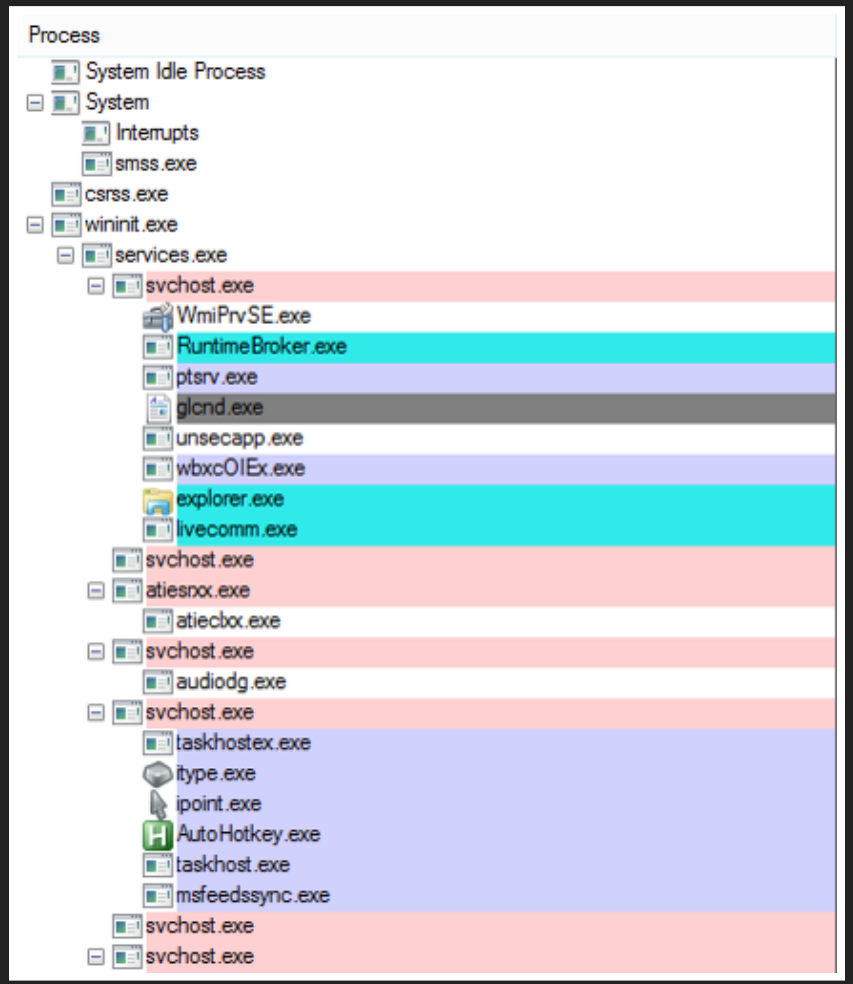


SO, WHAT DOES THIS HAVE TO DO WITH THREADING?

Lots, as you'll see...

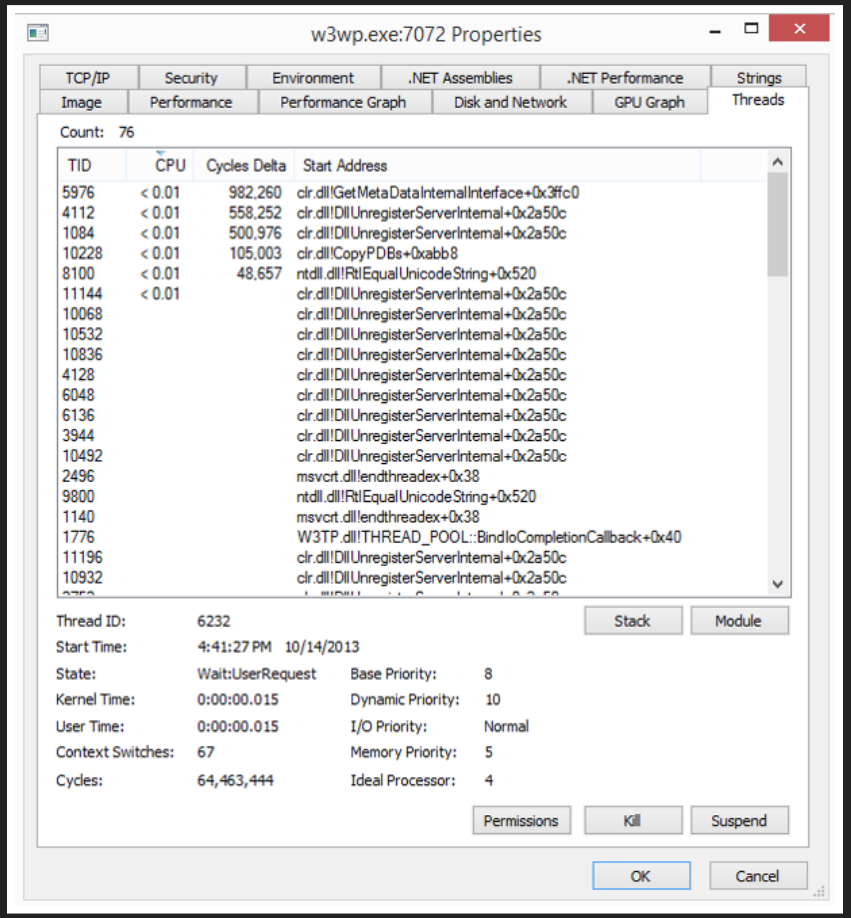
PROCESS

- Very heavy
- Container for threads
- Cannot share memory with other processes
- Scheduled by the OS



THREAD

- *Every* process has *at least* one thread
- Process memory is shared between threads
  - Insert ominous music here



SO, SHOULD I REALLY CARE?

WELL SURE YOU SHOULD CARE

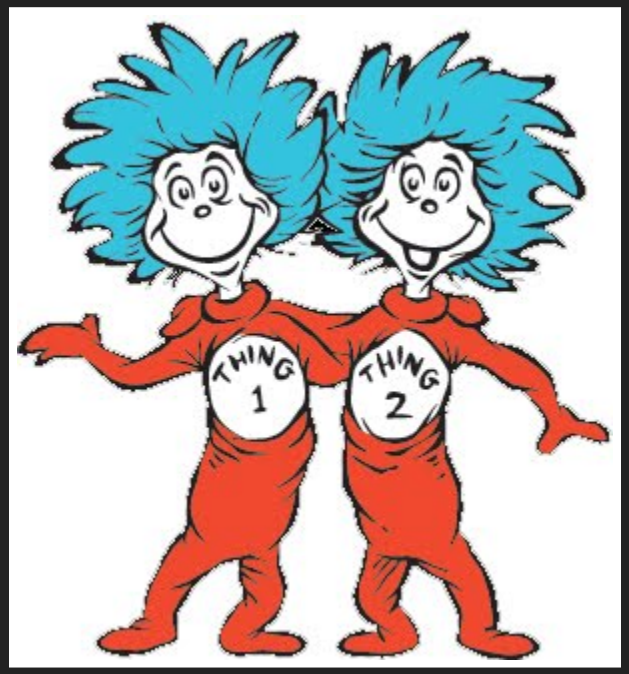
- It helps you understand why you shouldn't believe that adding threads makes your application faster
  - It *can* make it faster, but only in specific scenarios

BACK TO OUR QUESTION...

WHAT DOES THREADING SOLVE?

You said... (discuss)

THREADING SOLVES TWO THINGS

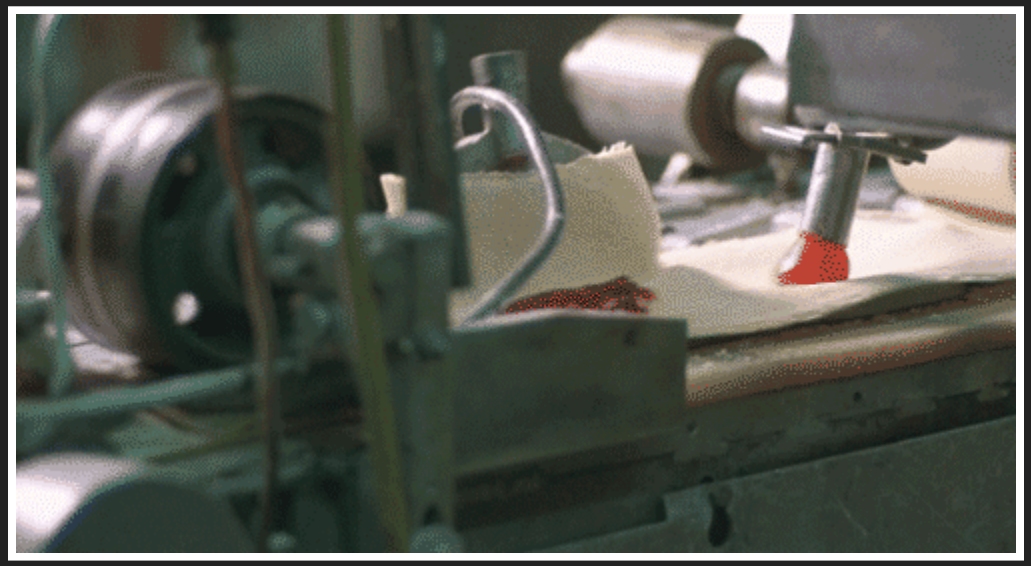


PARALLELISM

- Computational intensive processing that doesn't depend on the output of other steps
- Retrieving data from multiple services in parallel

REMOVING BLOCKING

- Not blocking the UI or web thread
- Blocking usually means blocking I/O (database, network, file, etc.)



DOES THAT SURPRISE YOU?

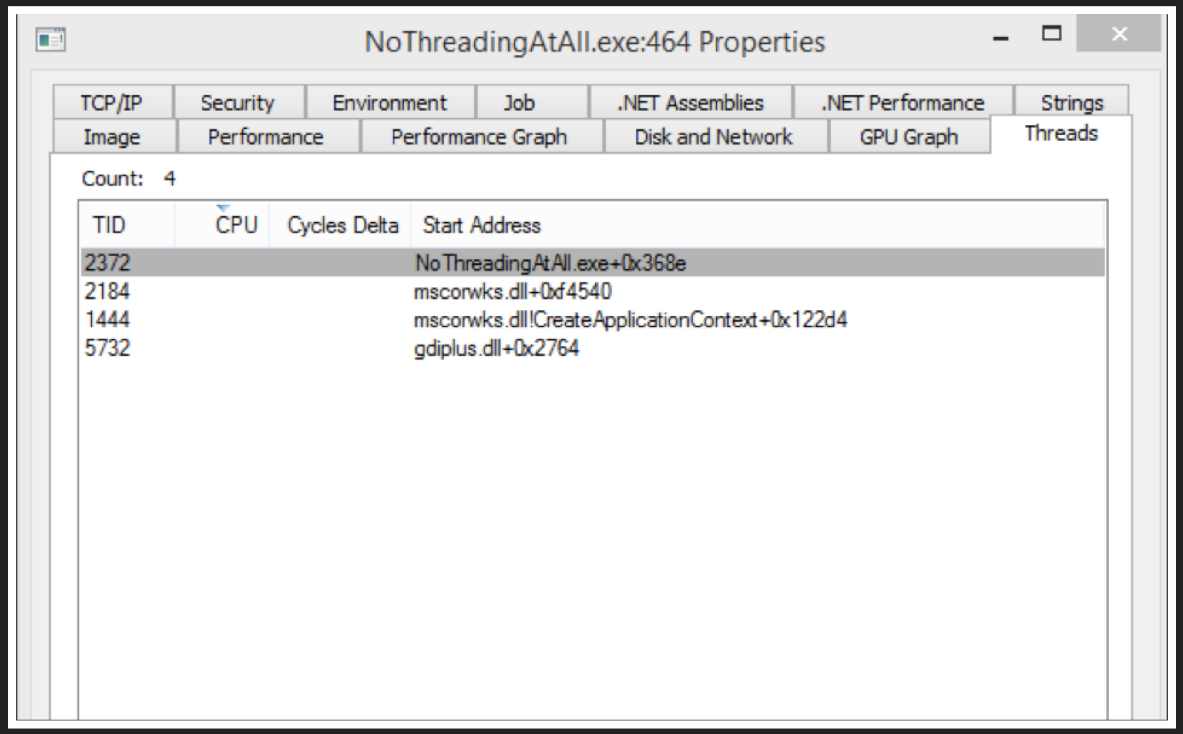
- Does this definition exclude anything?
- Did we miss anything?
- Only one of these *directly* affects speed

(discuss)

BASIC THREADING

(.NET 1.0-2.0)

EXAMPLE 01: NO (ADDITIONAL) THREADING



(wait... why are there four threads???)

EXAMPLE 02: SINGLE THREAD (THE WRONG WAY)

```
var thread = new System.Threading.Thread(DoWork);
thread.Start();
```

- Disclaimer - this code breaks some rules... in fact, it breaks *the golden rule of threading*...

EXAMPLE 02: FIX THE GUI THREAD

```
BeginInvoke(new Action<object>(x => {
   txtCount.Text = x.ToString();
}), i);
```

- In WinForms, you use `Control.Invoke` or `Control.BeginInvoke`
- In WPF, you use `Dispatcher.Invoke` or `Dispatcher.BeginInvoke`
- In addition, see `SynchronizationContext.Current` and use `Send` or `Post`

EXAMPLE 03: BACKGROUNDWORKER

```
using (var worker = new System.ComponentModel.BackgroundWorker()) {
   worker.WorkerReportsProgress = true;

   worker.DoWork += DoWork;
   worker.ProgressChanged += (o, args) => {
      txtCount.Text = args.UserState.ToString();
   };
   worker.RunWorkerCompleted += (o, args) => {
      txtTotalTime.Text = args.Result.ToString();
   };

   worker.RunWorkerAsync();
}
```

EXAMPLE 03: BACKGROUNDWORKER

- Introduced in .NET 2.0 because threading is hard
  - And because everyone was updating the UI thread from the background thread
- It falls in the easy category because...
  - It wasn't written by the threading team, but instead by the WinForms team (even lives under the `System.ComponentModel` namespace)
  - It can be dropped on your design surface
  - It automatically marshals calls back to the UI thread for you (via its events)

EXAMPLE 04: THREADPOOL THREADS

```
ThreadPool.QueueUserWorkItem(DoWork);
```

- Thread construction is expensive
  - They're good for long running background tasks, but if you're doing lots of small tasks, use a `ThreadPool` thread instead
- The CLR provides a "pool of threads" for you that are all ready to go
  - ASP.NET uses the ThreadPool for all of its requests
  - WCF uses the ThreadPool
  - And so on

QUICK ASIDE... WHEN SHOULD I USE WHICH?

- `BackgroundWorker`
  - Only use this if you're in WinForms or WPF or another GUI technology
- `ThreadPool`
  - Efficiency
  - Default to using this
  - Tasks (we're not there yet) are ThreadPool threads…
- `Thread`
  - Long running request
  - You need more control over thread details (e.g. priority, identity, etc.)

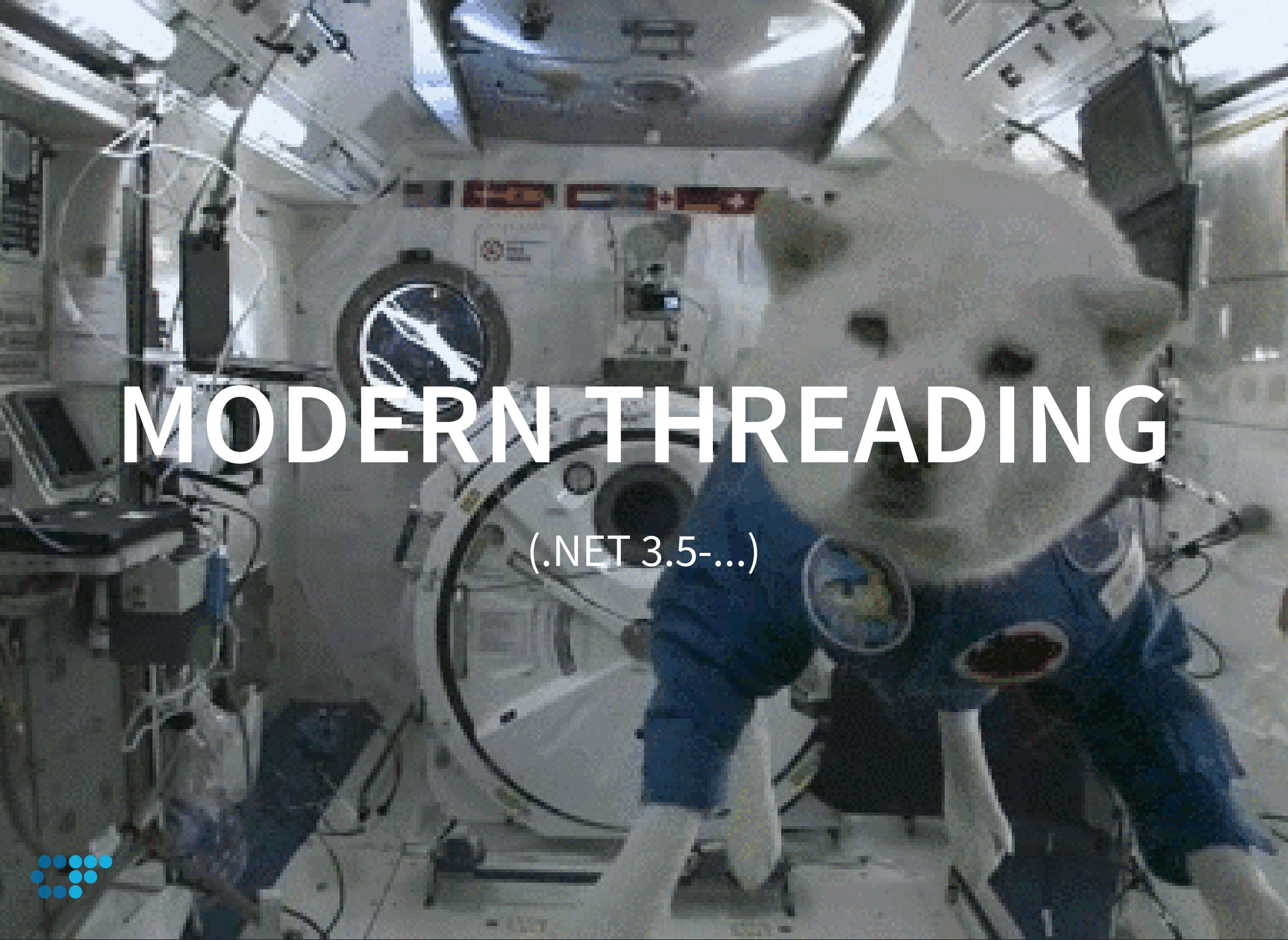
EXAMPLE 05: IASYNCRESULT

```
var iar = dlg.BeginInvoke(
   arg,
   Callback,
   new Tuple<Func<int, int>, int>(dlg, arg));
```

- Sometimes called the Asynchronous Programming Model (APM)
- Most of the time you consume APM instead of writing it yourself
- Usually only have to implement it when you're writing your own libraries

EXAMPLE 06: WHY IASYNCRESULT VERSUS...?

- So... why would I go through all of that complexity as compared to just using a ThreadPool thread on my own?
- It has everything to do with having a blocking thread or not


MODERN THREADING
(.NET 3.5-...)

TASK PARALLEL LIBRARY

- AKA Parallel Extensions AKA PFX AKA TPL...
- Built by Microsoft Research and CTPs were available as early as 2007 for .NET 3.5
- Introduced formally in .NET 4.0
- In addition to `Parallel.ForEach` and friends, also includes PLINQ
- *Very* much about parallelism (as opposed to removing blocking)

TASKS

- Tasks are really just a simpler model for thread pool threads
- `Task.Run(() => { })`
- `Task.WhenAll(...)`
- `Task.Factory.FromAsync`

```
Task<SqlDataReader>.Factory.FromAsync(
    command.BeginExecuteReader,
    command.EndExecuteReader, null);
```

PARALLEL LOOPING

```
foreach (var item in list) {
    // op (not in parallel)
}

for (var idx = 0; idx < 100, idx++)
    // op (not in parallel)
}
```

```
Parallel.ForEach(list, item => {
    // op (all in parallel)
});

Parallel.For(0, 100, idx => {
    // op (all in parallel)
});
```

PLINQ

```
(from x in someResults
 where x % 2 == 0
 select x).
Aggregate((x, y) => x + y);
```

```
(from x in someResults
 where x % 2 == 0
 select x).
AsParallel().
Aggregate((x, y) => x + y);
```

EXAMPLE 07: TPL, PLINQ, ETC.

ASYNC/AWAIT

- Technically, async/await provide compiler support for native continuations by building a state machine for you
- *Very* much about removing blocking calls (as opposed to the TPL)

```
private static async Task<IEnumerable<string>> LoadFilesAsync()
{
  IEnumerable<string> enumerable;
  using (ExecutionTimer.Start("LoadFilesAsync", false, true))
  {
    List<string> lines = new List<string>();
    object linesLock = new object();
    await Extensions.ForEachAsync<string>((IEnumerable<string>) Program
    {
      IEnumerable<string> result = await Program.LoadFileAsync(x);
      lock (linesLock)
        lines.AddRange(result);
    }));
    enumerable = (IEnumerable<string>) lines;
  }
  return enumerable;
}
```

```
private static Task<IEnumerable<string>> LoadFilesAsync()
{
  Program.<LoadFilesAsync>d__13 stateMachine;
  stateMachine.<>t__builder = AsyncTaskMethodBuilder<IEnumerable<string>>.Create();
  stateMachine.<>1__state = -1;
  stateMachine.<>t__builder.Start<Program.<LoadFilesAsync>d__13>(ref stateMachine);
  return stateMachine.<>t__builder.Task;
}
```

EXAMPLE 08: CONTINUATIONS! (AND ASYNC)

- Quick vocabulary
  - Coroutine
    - Yield in C# (and other languages)
    - Compiler support to break a function apart
  - Continuation
    - A pointer to a function to call when done
    - Ajax callbacks, JavaScript pointers, etc.

EXAMPLE 09: WHAT ABOUT .NET CORE???


ASYNC/AWAIT GOTCHAS

ASYNC/AWAIT GOTCHAS (PART 1)

- Async/await will "barf" all over your code
  - You'll think you can just add a single async/await pair here... but then you forget that every call above (unless you block) has to now have it
- Can't use them with properties (i.e. no async getter/setters)
- Can't use them with all LINQ (simple ones yes, complex no)
- Have to return Tasks all over the place now
  - See WCF calls

ASYNC/AWAIT GOTCHAS (PART 2)

- Can't use inline statements with async/await

```
// invalid
if (!await GetIsDoneAsync())

// valid
var isDone = await GetIsDoneAsync()
if (!isDone)
```

- Lots of built-in .NET calls do not yet support async/await
  - Process (wait for exit)
  - I/O (in some places yes, in others not)

GOTCHA: PARALLEL LOOPING OVER ASYNC

So, you've got the following:

```
foreach (var category in categories) {
  await category.GetStatusAsync();
}
```

Let's refactor to this!

```
Parallel.ForEach(categories, category => {
  await category.GetStatusAsync();
}
```

OH NO!


CLOSING

CLOSING TIPS

- If you're using `Parallel.ForEach` and `await` the same time, you're probably doing it wrong
- TPL is for parallelism
- `async`/`await` is for reducing blocking
- Reducing blocking can help with parallelism/throughput, but at the macro level*
- Between TPL and `async`/`await` you're probably rarely going to need TPL...
- Measure performance! Do you need those additional threads? And remember scalability, too!

QUESTIONS?

Presentation is available on Github