ThreadPool

ThreadPool

- a pool of two types of threads:
  - **worker threads** - execute work item callbacks, timer expiration callbacks, and wait for registration callbacks
  - **I/O completion port thread** (IOCP threads) - execute only I/O completion callbacks
- uses background threads
- is a process-wide, single pool
- threads destroyed after some time (trimming)
- used internally by many services (WCF, Remoting, ASP.NET/ASP.NET Core), timers, TPL, PLINQ, *event-based asynchronous pattern* (EAP) and `BeginXXX` methods (*asynchronous programming model pattern*), even multi-tier JIT
- has a self balancing mechanism - tries to keep a "good" number of threads
- no priority support of work items - we need to write our own
- resets properties of the thread returned to the pool, including any culture that has been left behind, thread priority and the thread name (i.e., changes made with the `Thread.Name` property) ensure that the thread is still marked as a background thread (i.e., `Thread.IsBackground` is true) so that it won't hold up process exit

ThreadPool - API

Most commonly used:

```
ThreadPool.QueueUserWorkItem(WaitCallback)
```

Sometimes used (we'll cover them too):

```
ThreadPool.GetAvailableThreads(Int32, Int32)
ThreadPool.GetMaxThreads(Int32, Int32)
ThreadPool.GetMinThreads(Int32, Int32)
ThreadPool.SetMaxThreads(Int32, Int32)
ThreadPool.SetMinThreads(Int32, Int32)
```

Much more advanced (we'll cover them only briefly):

```
ThreadPool.RegisterWaitForSingleObject(WaitHandle, WaitOrTimerCallback, Object, Int32, Boolean)
ThreadPool.BindHandle(SafeHandle)
ThreadPool.UnsafeQueueNativeOverlapped(NativeOverlapped*)
ThreadPool.UnsafeQueueUserWorkItem(WaitCallback, Object)
ThreadPool.UnsafeRegisterWaitForSingleObject(WaitHandle, WaitOrTimerCallback, Object, Int32, Boolean)
ThreadPool.UnsafeRegisterWaitForSingleObject(WaitHandle, WaitOrTimerCallback, Object, UInt32, Boolean)
```

ThreadPool - basic API

`ThreadPool.QueueUserWorkItem`

Queues a *work item* (the delegate) to be executed **at some time** by one of the `ThreadPool`'s *worker threads*:

- does not guarantee to be executed immediately
- there is no direct way to wait for it or get a result (*"fire & forget"*)
- has some caveats (we'll return to that...)

```
public static void Main()
{
   ThreadPool.QueueUserWorkItem(DoWork);
   Console.ReadLine();
}

static void DoWork(object? state)
{
   // Do something...
}
```

ThreadPool - exception handling

- if work item throws an exception, it is unhandled - kaput!
- we can use `AppDomain.CurrentDomain.UnhandledException` to log & save the work (but not to prevent the app closing)

```
public static void Main()
{
   AppDomain.CurrentDomain.UnhandledException += CurrentDomainOnUnhandledException;
   ThreadPool.QueueUserWorkItem(DoWork);
   Console.ReadLine();
}

private static void CurrentDomainOnUnhandledException(object sender, UnhandledExceptionEventArgs e)
{
   Console.WriteLine($"{(e.IsTerminating ? "Terminating" : "Non-terminating")} fault: {e.ExceptionObject}");
}

static void DoWork(object? state)
{
   throw new NullReferenceException();
}
```

ThreadPool - termination

- again: threads in the thread pool are **background threads**
- .NET process doesn't wait for work items (including those queued)
- again: cooperate by observing passed `CancelationToken.IsCancellationRequested`

ThreadPool - termination

Want to make sure processing will end?

- consider using different mechanism - separate thread(s) with some synchronization
- you *theoretically* can change thread pool's thread to be a foreground thread

ThreadPool - termination

Guarding by making threads foreground

```
public static void Main()
{
   ThreadPool.QueueUserWorkItem(DoWork);
   // main thread ends but will DoWork __if already queued__
}

static void DoWork(object? state)
{
   Thread.CurrentThread.IsBackground = false;
   // ...
}
```

It works because `IsBackground` is reset when the thread ends executing work item (so no other work items will be influenced).

Obvious two problems:

- the main thread (and the whole app) may end before queued work item was started
  - we could overcome that by observing `ThreadPool.PendingWorkItemCount` from another foreground thread
- even if enqueued, the `DoWork` may be terminated by app close before setting `IsBackground` flag
  - it needs some additional scheduling/synchronization mechanism

ThreadPool - managing the pool

- the thread pool starts with some default number of threads in it (zero?).
- then it needs to add/remove them as needed:
  - tries to keep the number of threads low, tunes to the workload, and **avoids creating new threads as long as possible**
  - too few threads - if many work items waiting, we waste CPU
  - too many threads - context switching/cache misses and CPU trashing (a lot of waits ended at the same time)
- in fact - what is the "wait ratio" of work items?
- thread pool maintains a minimum and a maximum number of threads (separately for worker and IOCP)

ThreadPool - managing the pool

maximum number of threads

- means a real maximum
- there will be no more worker/IOCP threads created than this value
- if work items count exceeded - the work item will be **queued** (until one worker threads becomes available)

ThreadPool - managing the pool

maximum number of threads

- default value for worker threads:
  - .NET Framework 2.0 - 25 per core
  - .NET Framework 3.5 - 250 per core
  - .NET Framework 4.0 in a 32-bit environment - 1,023
  - **.NET Core x64**, .NET Framework 4.0+ - 32,768
  - .NET Core x86 - *X*
    - *X* = (*half of 32-bit address space*)/(*default stack size*)
    - default stack size taken from EXE (typically 1.5MB on Windows) or 256 kB (assumed on Linux)
    - thus typically *X* is 682 (Windows)
- deafault value for IOCP threads:
  - 1000

More details in `win32threadpool.cpp` for brave ones

ThreadPool - managing the pool

minimum number of threads

- does not mean a real minimum!
- there may be fewer threads than this value (including 0)
- if there are fewer current threads than a minimum - create threads **immediately** as needed
- if there are more current threads than a minimum - create every new threat **at maximum 0.5 second intervals**

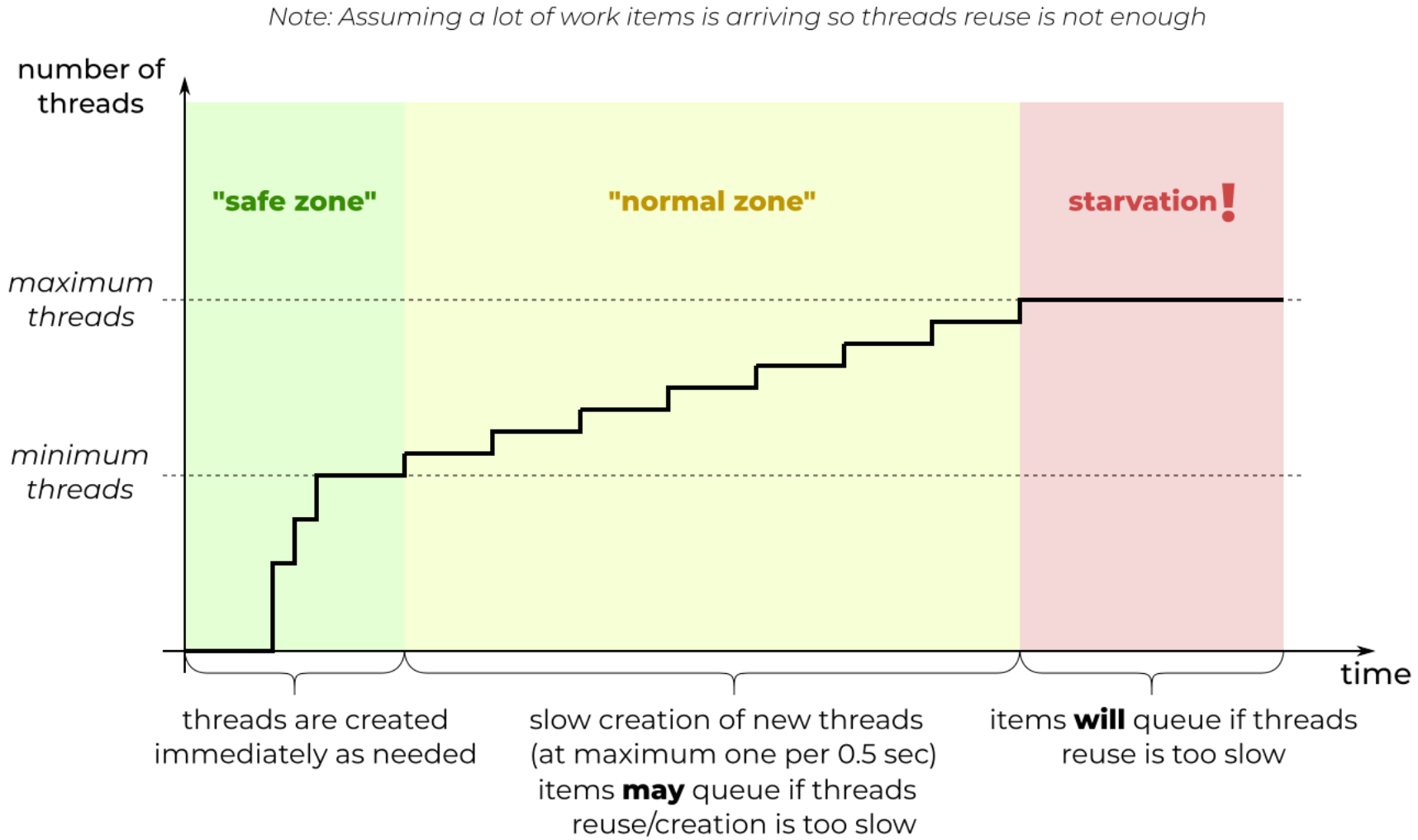
ThreadPool - managing the pool

minimum number of threads

- default value for worker & IOCP threads: number of processors (logical cores)
- raising it may improve concurrency and better "burst response" in case of many blocked threads

IOCP threads throttled by IOCP/Windows

ThreadPool - managing the pool



ThreadPool - managing the pool

Two main algorithms for managing threads in "normal zone":

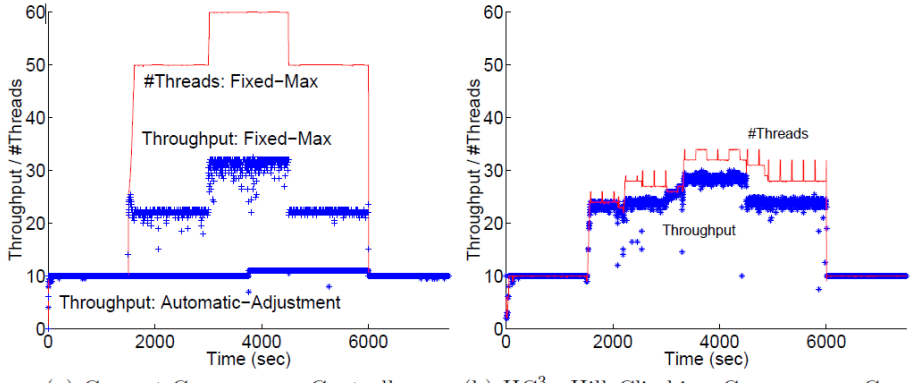
- a starvation-avoidance mechanism - adding worker threads if it sees no progress being made on queued items
- a *hill-climbing heuristic* - maximize throughput while using as few threads as possible (injecting/removing threads)

An opportunity to inject threads (whichever is shorter):

- every time a work item completes, or
- **at maximum 0.5 second intervals** (if *minimum* exceeded)

ThreadPool - managing the pool

Hill-climbing described in *"Optimizing Concurrency Levels in the .NET ThreadPool"* paper published by Microsoft.



(a) Current Concurrency Controller (b) HC^3: Hill Climbing Concurrency Controller

Figure 3: *Performance of Concurrency Controllers for the CLR ThreadPool for a Dynamic Workload*

In general: *"adapt quickly, but not too quickly"*, basing on:

- current thread count,
- current number of items completed/time,
- queueing rate,
- CPU utilization

After a sufficient period of inactivity, the pool manager may “retire” threads.

ThreadPool - managing the pool

Summary

- in .NET Core it is pretty hard to have a real **thread pool starvation**
  - exceeding 32,768 work items...
- but we may lead to *"soft"* thread pool starvation
  - if we have more than *minimum* and there is a "rush" of requests, new threads will be created very slowly
  - work items will queue due to not enough worker threads
  - hill-climbing tries to accommodate that (and eventually will)
  - such state is mostly dangerous for unexpected "peaks" of wok items
- it may influence... everything: HTTP request processing, `async` calls, timers, even JIT

ThreadPool - configuration

.NET Framework

Configuration element in `machine.config` (yes, system-wide):

```
<processModel autoConfig="false" minIoThreads="250" />
```

Remember: *"the value specified in this configuration element is a per-core setting"*

.NET Core

`[appname].runtimeconfig.json`

```
{
    "runtimeOptions": {

        "configProperties": {
            "System.GC.Server": true,
            "System.GC.Concurrent": true,
            "System.Threading.ThreadPool.MinThreads": 4,
            "System.Threading.ThreadPool.MaxThreads": 8
        },
        ...
    }
}
```

ThreadPool - configuration

Programatically

```
ThreadPool.SetMinThreads(int workerThreads, int completionPortThreads)
ThreadPool.SetMaxThreads(int workerThreads, int completionPortThreads)
```

Sidenote: You cannot set the maximum number of worker threads or I/O completion threads to a number smaller than the number of processors on the computer.

Demo: `ThreadPool` **use with different API**

ThreadPool - waits

Registered wait - a special, single *wait thread* created for every 63 objects registered - This thread manages waiting on objects and queuing the callbacks to run on one of the thread pool's worker threads when an object is signaled

ThreadPool - waits

`ThreadPool.RegisterWaitForSingleObject` - asynchronously waiting on a wait handle, instead of a synchronous `.WaitOne()` that blocks a thread

```
class CallbackInfo
{
    public RegisteredWaitHandle Handle;
}

class RegisteredWaitDemo
{
    public static void Main(string[] args)
    {
        AutoResetEvent evt = new AutoResetEvent(false);
        var callbackInfo = new CallbackInfo();
        callbackInfo.Handle = ThreadPool.RegisterWaitForSingleObject(evt, CallBack, state: callbackInfo,
            timeout: TimeSpan.FromSeconds(3.0), executeOnlyOnce: true);

        Thread.Sleep(2000); // Start long running operation
        evt.Set(); // ... and signal it has finished
        // Dispose AutoResetEvent or reuse for other notifications
    }

    private static void CallBack(object? state, bool timedOut)
    {
        var callbackInfo = state! as CallbackInfo;
        callbackInfo?.Handle?.Unregister(null);   // Good practice to unregister
    }
}
```

ThreadPool - waits

Executes `action` on a thread pool thread after `sleep` milliseconds - consumes only *wait thread* for waiting:

```
public static void ExecuteAfterWait<T>(int sleep, Action<T> action, T arg)
{
   AutoResetEvent evt = new AutoResetEvent(false);
   RegisteredWaitHandle? handle = null;
   handle = ThreadPool.RegisterWaitForSingleObject(evt, (state, timedOut) =>
   {
       handle!.Unregister(evt);
       action(arg);
   }, null, sleep, true);
}
```

ThreadPool internals - task queues (since .NET 4)

A *work item* will land in one of two queues:

- global task queue
  - if the item has been enqueued from a non-thread pool thread
  - if we use `ThreadPool.QueueUserWorkItem`/`ThreadPool.UnsafeQueueUserWorkItem`
  - if we use `TaskCreationOptions.PreferFairness` flag with `Task.Factory.StartNew`
  - if we call `Task.Yield` on the default task scheduler
- local task queues for each worker thread
  - if the item has been enqueued from a thread pool thread

ThreadPool internals - dequeuing

Worker thread looks:

- into its local queue (LIFO order)
  - LIFO to preserve cache/data locality
- if empty, into global queue (FIFO order)
- if empty, into local queues of other threads (*"work stealing"*, FIFO order)

ThreadPool internals - dequeuing

".NET Threadpool starvation, and how queuing makes it worse" by Criteo - great article about example deadlock scenario because of that:

ThreadPool internals - dequeuing

".NET Threadpool starvation, and how queuing makes it worse" by Criteo - great article about example deadlock scenario because of that:

1. a constant burst of requests comes

ThreadPool internals - dequeuing

".NET Threadpool starvation, and how queuing makes it worse" by Criteo - great article about example deadlock scenario because of that:

1. a constant burst of requests comes
2. thread pool becomes starved (*softly*)

ThreadPool internals - dequeuing

".NET Threadpool starvation, and how queuing makes it worse" by Criteo - great article about example deadlock scenario because of that:

1. a constant burst of requests comes
2. thread pool becomes starved (*softly*)
3. enqueue 5 items per second **to the global queue** (because of burst)

ThreadPool internals - dequeuing

".NET Threadpool starvation, and how queuing makes it worse" by Criteo - great article about example deadlock scenario because of that:

1. a constant burst of requests comes
2. thread pool becomes starved (*softly*)
3. enqueue 5 items per second **to the global queue** (because of burst)
4. each of such items enqueues one more item into the local queue **and waits for it**

ThreadPool internals - dequeuing

".NET Threadpool starvation, and how queuing makes it worse" by Criteo - great article about example deadlock scenario because of that:

1. a constant burst of requests comes
2. thread pool becomes starved (*softly*)
3. enqueue 5 items per second **to the global queue** (because of burst)
4. each of such items enqueues one more item into the local queue **and waits for it**
5. when a new thread is created (because of starvation), it looks into:
    - its local queue - but it is empty, as it was just created
    - the global queue - which is constantly growing

ThreadPool internals - dequeuing

".NET Threadpool starvation, and how queuing makes it worse" by Criteo - great article about example deadlock scenario because of that:

1. a constant burst of requests comes
2. thread pool becomes starved (*softly*)
3. enqueue 5 items per second **to the global queue** (because of burst)
4. each of such items enqueues one more item into the local queue **and waits for it**
5. when a new thread is created (because of starvation), it looks into:
    - its local queue - but it is empty, as it was just created
    - the global queue - which is constantly growing
6. enqueueing to the global queue (**5/sec**) is faster that the threadpool grow (**0.5/sec**) so we never recover - locally queued tasks are not processed as newly created threads are taking tasks from the global queue. So worker threads are blocked waiting...

ThreadPool internals - dequeuing

".NET Threadpool starvation, and how queuing makes it worse" by Criteo - great article about example deadlock scenario because of that:

1. a constant burst of requests comes
2. thread pool becomes starved (*softly*)
3. enqueue 5 items per second **to the global queue** (because of burst)
4. each of such items enqueues one more item into the local queue **and waits for it**
5. when a new thread is created (because of starvation), it looks into:
    - its local queue - but it is empty, as it was just created
    - the global queue - which is constantly growing
6. enqueueing to the global queue (**5/sec**) is faster that the threadpool grow (**0.5/sec**) so we never recover - locally queued tasks are not processed as newly created threads are taking tasks from the global queue. So worker threads are blocked waiting...

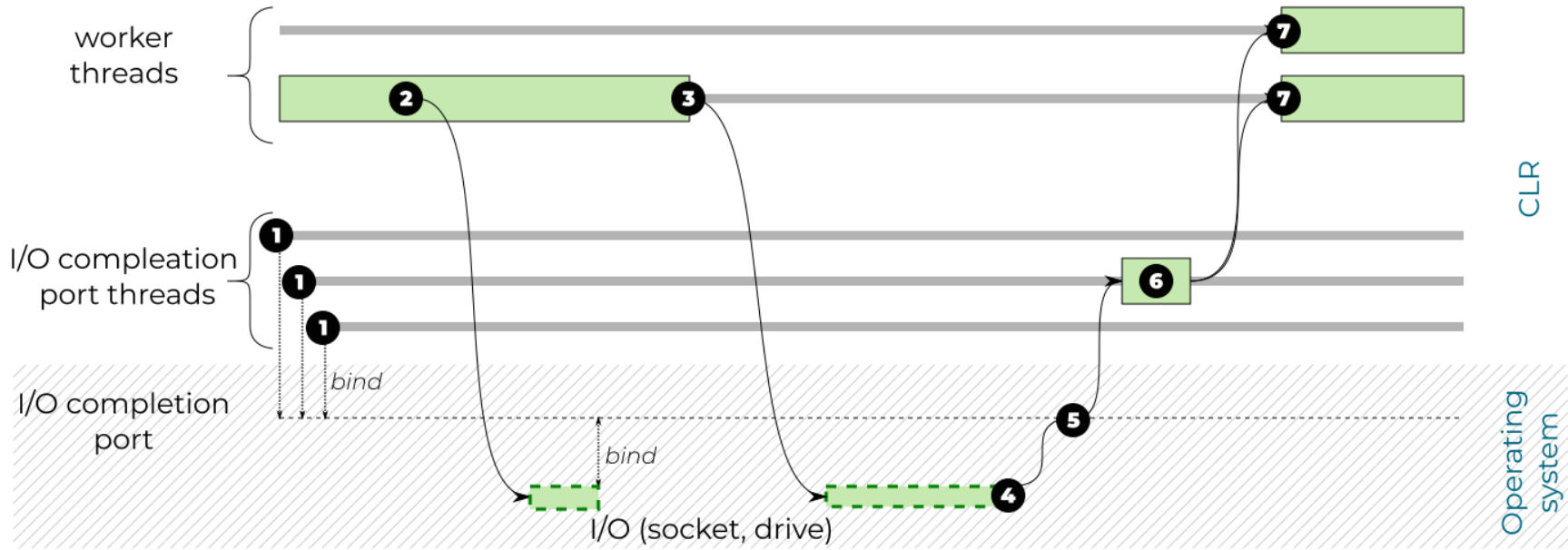
Solution:

- change step 3 to enqueue **to the local queue**
- when a new thread is created (because of starvation), it looks into:
    - its local queue - but it is empty, as it was just created
    - the global queue - but it is empty, as we not enqueue there anything
    - steals the work from the other local queue
- never, ever wait synchronously!

ThreadPool internals - I/O operations

- we issue an **overlapped I/O**
  - it may complete immediately
  - or, returns a status code indicating the I/O operation is still pending.
- I/O completion port - to observe the result
  - it may represent many simultaneous ongoings I/O operations
  - one or many threads may wait for the IOCP
  - then it executes some callback with the incoming data
- in case of .NET ThreadPool:
  - one of IOCP threads will observe the change
  - mark the task complete
  - the continuation is queued to the target `SynchronizationContext` or the thread pool
- IOCP threads are blocked, yes - they block on the IOCP itself.
  - one IOCP thread can handle a huge number of IOCP notifications,

ThreadPool internals - I/O operations



ThreadPool - monitoring

Programatically:

```
int workerThreads = 0;
int completionPortThreads = 0;
ThreadPool.GetAvailableThreads(out workerThreads, out completionPortThreads);

ThreadPool.GetMinThreads(out int workerThreads, out int completionPortThreads);
ThreadPool.GetMaxThreads(out int workerThreads, out int completionPortThreads);
```

ThreadPool - monitoring

`dotnet counters`:

```
[System.Runtime]
    % Time in GC since last GC (%)                      0
    Allocation Rate / 1 sec (B)                         0
    CPU Usage (%)                                       0
    Exception Count / 1 sec                             0
    GC Heap Size (MB)                                   1
    Gen 0 GC Count / 60 sec                             0
    Gen 0 Size (B)                                      0
    Gen 1 GC Count / 60 sec                             0
    Gen 1 Size (B)                                      0
    Gen 2 GC Count / 60 sec                             0
    Gen 2 Size (B)                                      0
    LOH Size (B)                                        0
    Monitor Lock Contention Count / 1 sec               0
    Number of Active Timers                             0
    Number of Assemblies Loaded                        10
    ThreadPool Completed Work Item Count / 1 sec        0
    ThreadPool Queue Length                         1,978
    ThreadPool Thread Count                            25
    Working Set (MB)                                   21
```

or... `ASP.NET 4.0.x/Request Queued` performance counter in case of ASP.NET Framework

ThreadPool - monitoring

PerfView (standard *.NET* session) or `dotnet-trace` (opened in PerfView)

Event Name	Time MSec	Process Name	Rest
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	4,682.745	StandaloneThreadPoolConsoleApp (24324)	ThreadID="19,912" ProcessorNumber="2" ActiveWorkerThreadCount="1" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	4,779.733	StandaloneThreadPoolConsoleApp (24324)	ThreadID="19,912" ProcessorNumber="7" ActiveWorkerThreadCount="1" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	4,821.481	StandaloneThreadPoolConsoleApp (24324)	ThreadID="11,888" ProcessorNumber="6" ActiveWorkerThreadCount="2" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	24,854.348	StandaloneThreadPoolConsoleApp (24324)	ThreadID="11,068" ProcessorNumber="6" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	25,148.793	StandaloneThreadPoolConsoleApp (24324)	ThreadID="11,068" ProcessorNumber="5" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	25,248.163	StandaloneThreadPoolConsoleApp (24324)	ThreadID="11,068" ProcessorNumber="5" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	25,856.002	StandaloneThreadPoolConsoleApp (24324)	ThreadID="11,068" ProcessorNumber="6" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	25,962.181	StandaloneThreadPoolConsoleApp (24324)	ThreadID="11,068" ProcessorNumber="5" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	26,058.291	StandaloneThreadPoolConsoleApp (24324)	ThreadID="11,068" ProcessorNumber="5" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	27,177.619	StandaloneThreadPoolConsoleApp (24324)	ThreadID="11,068" ProcessorNumber="5" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	27,778.867	StandaloneThreadPoolConsoleApp (24324)	ThreadID="11,068" ProcessorNumber="6" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	31,296.122	StandaloneThreadPoolConsoleApp (24324)	ThreadID="11,068" ProcessorNumber="2" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	31,411.726	StandaloneThreadPoolConsoleApp (24324)	ThreadID="11,068" ProcessorNumber="5" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	32,832.704	StandaloneThreadPoolConsoleApp (24324)	ThreadID="11,068" ProcessorNumber="6" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	32,832.718	StandaloneThreadPoolConsoleApp (24324)	ThreadID="19,912" ProcessorNumber="4" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	34,860.919	StandaloneThreadPoolConsoleApp (24324)	ThreadID="19,912" ProcessorNumber="6" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	34,860.942	StandaloneThreadPoolConsoleApp (24324)	ThreadID="14,728" ProcessorNumber="4" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	34,861.896	StandaloneThreadPoolConsoleApp (24324)	ThreadID="14,728" ProcessorNumber="2" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	34,861.899	StandaloneThreadPoolConsoleApp (24324)	ThreadID="14,028" ProcessorNumber="0" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	34,861.911	StandaloneThreadPoolConsoleApp (24324)	ThreadID="14,728" ProcessorNumber="2" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	34,861.923	StandaloneThreadPoolConsoleApp (24324)	ThreadID="3,212" ProcessorNumber="3" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Click on Update button to update.	34,862.549	StandaloneThreadPoolConsoleApp (24324)	ThreadID="14,028" ProcessorNumber="0" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	34,862.901	StandaloneThreadPoolConsoleApp (24324)	ThreadID="24,248" ProcessorNumber="3" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	34,862.901	StandaloneThreadPoolConsoleApp (24324)	ThreadID="28,344" ProcessorNumber="6" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	34,862.902	StandaloneThreadPoolConsoleApp (24324)	ThreadID="24,248" ProcessorNumber="3" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	34,862.906	StandaloneThreadPoolConsoleApp (24324)	ThreadID="14,028" ProcessorNumber="7" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	34,862.910	StandaloneThreadPoolConsoleApp (24324)	ThreadID="11,888" ProcessorNumber="1" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Wait	34,862.913	StandaloneThreadPoolConsoleApp (24324)	ThreadID="28,344" ProcessorNumber="6" ActiveWorkerThreadCount="8" RetiredWorkerThreadCount="0" ClrInstanceID="11"

ThreadPool - monitoring

Dump analysis with the help of SOS `threadpool` command:

```
> dotnet dump collect -p 29924
Writing full to dump_20200515_131534.dmp
Complete

> dotnet dump analyze .\dump_20200515_131534.dmp
Loading core dump: .\dump_20200515_131534.dmp ...
Ready to process analysis commands. Type 'help' to list available commands or 'help [command]' to get detailed
help on a command.
Type 'quit' or 'exit' to exit the session.
> threadpool
CPU utilization: 17%
Worker Thread: Total: 8 Running: 8 Idle: 0 MaxLimit: 8 MinLimit: 8
Work Request in Queue: 1
    AsyncTimerCallbackCompletion TimerInfo@000001E91E19FE50
--------------------------------------
Number of Timers: 0
--------------------------------------
Completion Port Thread:Total: 0 Free: 0 MaxFree: 16 CurrentLimit: 0 MaxLimit: 1000 MinLimit: 8
```

Scenario - StackExchange.Redis

Are you seeing a high number of busyio or busyworker threads in the timeout exception?

StackExchange.Redis:

- has its own dedicated thread-pool (to have better control) - serves most calls
- fall-backs to .NET `ThreadPool` if dedicated is being full

*"Note that **StackExchange.Redis can hit timeouts** if either the IOCP threads or the worker threads (.NET global thread-pool, or the dedicated thread-pool) become saturated without the ability to grow."*

*"Basically, if you're hitting the global thread pool (rather than the dedicated StackExchange.Redis thread-pool) it means that when the number of Busy threads is greater than Min threads, **you are likely paying a 500ms delay** before network traffic is processed by the application."*

"If we look at an example error message from StackExchange.Redis 2.0, you will see that it now prints ThreadPool statistics (see IOCP and WORKER details below)"

```
Timeout performing GET MyKey (1000ms), inst: 2, qs: 6, in: 0, mgr: 9 of 10 available,
IOCP: (Busy=6,Free=994,Min=4,Max=1000),
WORKER: (Busy=3,Free=997,Min=4,Max=1000)
```

"A good starting place is 200 or 300, then test and tweak as needed."

ThreadPool - advanced configuration

Advanced CLR configuration knobs about thread pool:

- `ThreadPool_DebugBreakOnWorkerStarvation` - breaks into the debugger if the `ThreadPool` detects work queue starvation (default 0)
- `ThreadPool_DisableStarvationDetection` - disables the `ThreadPool` feature that forces new threads to be added when workitems run for too long (default 0)
- `ThreadPool_EnableWorkerTracking` - enables **extra expensive** tracking of how many workers threads are working simultaneously (default 0)
- `ThreadPool_ForceMaxWorkerThreads` - overrides the `MaxThreads` setting for the `ThreadPool` worker pool (default 0)
- `ThreadPool_ForceMinWorkerThreads` - overrides the `MinThreads` setting for the `ThreadPool` worker pool (default 0)
- `ThreadPool_UnfairSemaphoreSpinLimit` - maximum number of spins per processor a thread pool worker thread performs before waiting for work (default 0x32)
- `ThreadpoolTickCountAdjustment` - internal setting used only in the Debug build of the runtime
- `HillClimbing_` - *a bunch of settings for* Hill climbing* algorithm

ThreadPool - advanced configuration

`ThreadPool_EnableWorkerTracking` **example -** ***"enables extra expensive tracking of how many workers threads are working simultaneously"***

Event Name	Time MSec	Process Name	Count
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkingThreadCount/Start	30,671.967	StandaloneThreadPoolConsoleApp (24324)	7
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkingThreadCount/Start	31,170.799	StandaloneThreadPoolConsoleApp (24324)	7
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkingThreadCount/Start	31,672.373	StandaloneThreadPoolConsoleApp (24324)	8
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkingThreadCount/Start	32,171.680	StandaloneThreadPoolConsoleApp (24324)	7
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkingThreadCount/Start	32,671.572	StandaloneThreadPoolConsoleApp (24324)	7
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkingThreadCount/Start	33,171.331	StandaloneThreadPoolConsoleApp (24324)	7
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkingThreadCount/Start	33,671.010	StandaloneThreadPoolConsoleApp (24324)	6
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkingThreadCount/Start	34,171.363	StandaloneThreadPoolConsoleApp (24324)	6
Microsoft-Windows-DotNETRuntime/ThreadPoolWorkingThreadCount/Start	34,672.399	StandaloneThreadPoolConsoleApp (24324)	6

ThreadPool internals - I/O operations (cont.)

I/O completion port is exposed through the .NET `ThreadPool.BindHandle(SafeHandle handle)` method:

- A `SafeHandle` that holds the operating system handle. The handle must have been opened for *overlapped I/O*
- you can use it to serve high-throughput I/O devices, use overlapped I/O with completion ports
- you can bind a given handle to be served by IOCP Threads

Demo: `ThreadPool.BindHandle` **usage**

ThreadPool - "unsafe" API

`ThreadPool.UnsafeQueueUserWorkItem()`

`ExecutionContext` which includes various security information (impersonation information established for the thread), is **NOT captured** at the time of the call (on the queuing thread) and so, then **NOT used** when invoking the callback on the thread pool. But **faster**.

`ThreadPool.UnsafeQueueNativeOverlapped()`

Queues an overlapped I/O operation for execution (on IOCP threads pool).

`ThreadPool.UnsafeRegisterWaitForSingleObject()`

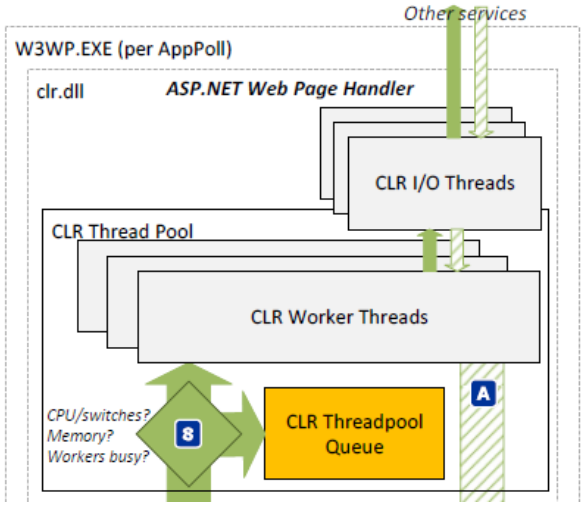
Unsafe version of `RegisterWaitForSingleObject`, so no `ExecutionContext` capturing.

ThreadPool - Materials

- https://dschenkelman.github.io/2013/10/29/asynchronous-io-in-c-io-completion-ports/
- Understanding the Windows I/O System
- Multithreaded Asynchronous I/O & I/O Completion Ports
- The CLR Thread Pool 'Thread Injection' Algorithm
- Concurrency - Throttling Concurrency in the CLR 4.0 ThreadPool
- New and Improved CLR 4 Thread Pool Engine
- Diagnosing .NET Core ThreadPool Starvation with PerfView

IIS hosting

Kestrel hosting



Monitoring	Tuning
.NET CLR Networking 4.x **/HttpWebRequests Queued/Sec**	.NET config: maxConnection
ASP.NET 4.0.x **/Request Queued** /Request Execution Time	.NET config: /maxWorkerThreads /minWorkerThreads /maxIoThreads /minIoThreads /autoConfig
Process /Thread Count	
`!threadpool`	

Kestrel hosting

Kestrel hosting

- asynchronous I/O initially based on `libuv` library (node.js) - single-threaded even looping model
- all other work done by worker threads
- *"Kestrel also supports running multiple event loops, thereby making it a more robust choice than Node.js. The number of event loops that are used depends on the number of logical processors on the machine"*

Kestrel hosting - ASP.NET Core 1.x

```
WebHost.CreateDefaultBuilder(args)
       .UseLibuv(opts => opts.ThreadCount = 4)
       .UseStartup<Startup>();
```

Kestrel hosting - ASP.NET Core 2.1+

- `libuv` replaced by **managed sockets**
  - the Linux transport makes use of non-blocking sockets and *epoll*
  - *"The big debate we usually have around running code on epoll threads is blocking user code. As a result, we dispatch from a small number of epoll threads to the thread pool (which grows and supports work stealing)"* - David Fowler
- fall-back possible with `Microsoft.AspNetCore.Server.Kestrel.Transport.Libuv` NuGet package

Kestrel hosting - ASP.NET Core 2.1+

```
Host.CreateDefaultBuilder(args)
    .ConfigureKestrel((context, options) =>
    {
        options.Limits.MaxConcurrentConnections = 100;
        options.Limits.MaxConcurrentUpgradedConnections = 100; // limit for connections that have been upgraded
                                                               // from HTTP or HTTPS to another protocol (for
                                                               // example, on a WebSockets request)

    }
```

appsettings.json or appsettings.{Environment}.json:

```
{
  "Kestrel": {
    "Limits": {
      "MaxConcurrentConnections": 100,
      "MaxConcurrentUpgradedConnections": 100
    },
    ...
  }
}
```

Kestrel hosting - ASP.NET Core 2.1+

`SocketTransportOptions.IOQueueCount` - the number of I/O queues used to process requests. Set to 0 to directly schedule I/O to the `ThreadPool`. Defaults to `ProcessorCount` rounded down and clamped between 1 and 16.

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
   Host.CreateDefaultBuilder(args)
       .ConfigureWebHostDefaults(webBuilder =>
       {
           webBuilder
               .UseSockets(options => options.IOQueueCount = 0)
               .UseStartup<Startup>();
       });
```

Kestrel hosting - custom

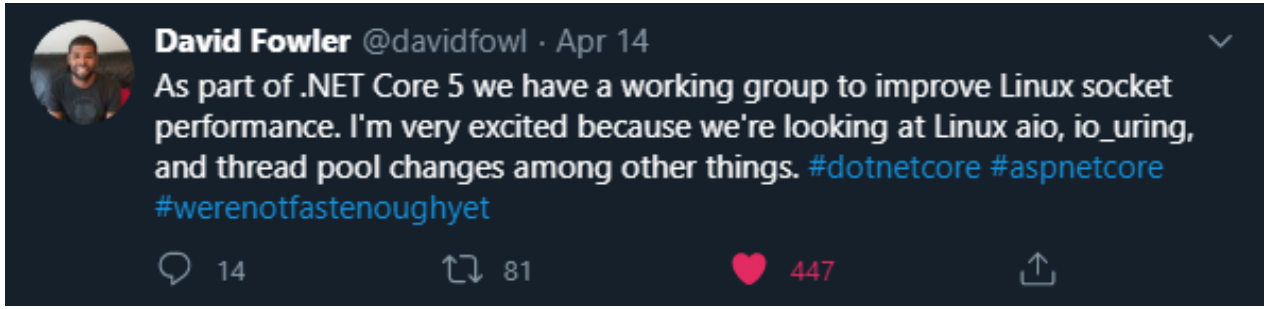
https://github.com/redhat-developer/kestrel-linux-transport - custom `Transport` abstraction implementation for Kestrel, uses `SO_REUSEPORT` and *POSIX AIO* (asynchronous I/O) interface

- reference `RedHat.AspNetCore.Server.Kestrel.Transport.Linux` package
- configure `WebHost`:

```
public static IWebHost BuildWebHost(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder
                        .UseLinuxTransport()
                        .UseStartup<Startup>();
                });
```

Kestrel hosting - future

.NET 5 on Linux: AIO, io_uring, and thread pool changes



ThreadPool - asynchronous delegates

ThreadPool - asynchronous delegates

- `ThreadPool.QueueUserWorkItem` doesn't expose any direct way to return a result
- *asynchronous delegate* allows any number of typed arguments to be passed in both directions
- `BeginInvoke` to start and then `EndInvoke`:
  - waits for the asynchronous delegate to execute and finish on the thread pool,
  - it receives the return value
  - it (re)throws an unhandled exception

```
Func<T1, T2> method = DoWork;
IAsyncResult handle = method.BeginInvoke(args);
// async part
T2 result = method.EndInvoke(handle);
...
```

or

```
Func<T1, T2> method = DoWork;
method.BeginInvoke(args, OnDone, method);
// async part
...
static void OnDone<T1, T2>(IAsyncResult handle)
{
  var target = (Func<T1, T2>)handle.AsyncState;
  T2 result = target.EndInvoke(handle);
  ...
}
```

ThreadPool - asynchronous delegates

No-no, legacy!

Asynchronous Programming Model (APM)

Asynchronous Programming Model (APM)

- pair of `Begin.../End...` methods, for example:

```
public IAsyncResult BeginRead(byte[] buffer, int offset,
                              int count, AsyncCallback callback,
                              object state);
public int EndRead(IAsyncResult asyncResult); // waits for result
```

- `IAsyncResult`:
  - `IsCompleted`
  - `CompletedSynchronously`
  - `AsyncWaitHandle`
  - `AsyncState`
- accessing results:
  - pooling `IsCompleted` and then calling `End...`
  - waiting on `AsyncWaitHandle` (with timeout), i.e. with `ThreadPool.RegisterWaitForSingleObject`
  - using `delegate void AsyncCallback(IAsyncResult iar)`
  - `Task<int>.Factory.FromAsync(fs.BeginRead, fs.EndRead, buffer, 0, buffer.Length, null)`

Event Asynchronous Pattern (EAP)

Event Asynchronous Pattern (EAP)

- asynchronous operation start and C# event handler for results
- implicit multi-threading
- implicit synchronization with proper context (UI)
- very rare: `WebClient`, `BackgroundWorker` and exotic: `SoundPlayer`, `System.Windows.Forms.PictureBox.LoadAsync`
- those `...Async` methods are not our beloved async, i.e. `WebClient.DownloadStringAsync` and `WebClient.DownloadStringAsyncTask`

Event Asynchronous Pattern (EAP)

`WebClient`

```
public static void Main(string[] args)
{
    WebClient client = new WebClient();
    client.DownloadStringCompleted += OnDownloadCompleted;
    client.DownloadStringAsync(new Uri(@"https://google.com"));
    //client.CancelAsync();
}

private static void OnDownloadCompleted(object sender,
                                        DownloadStringCompletedEventArgs args)
{
    // UI is safe here
    if (args.Cancelled)
        Console.WriteLine("Canceled");
    else if (args.Error != null)
        Console.WriteLine("Exception: " + args.Error.Message);
    else
        Console.WriteLine(args.Result);
}
```

Event Asynchronous Pattern (EAP)

`BackgroundWorker`

General-purpose "worker" with cancellation, progress, UI thread sync - based on the thread pool.

```
public static void Run(string[] args)
{
    var worker = new BackgroundWorker();
    worker.DoWork += WorkerOnDoWork;
    worker.ProgressChanged += WorkerOnProgressChanged;
    worker.RunWorkerCompleted += WorkerOnRunWorkerCompleted;
    worker.RunWorkerAsync(worker);
    // Dispose or reuse the worker
}

private static void WorkerOnRunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
}

private static void WorkerOnProgressChanged(object sender, ProgressChangedEventArgs e)
{
}

private static void WorkerOnDoWork(object sender, DoWorkEventArgs e)
{
    BackgroundWorker worker = e.Argument as BackgroundWorker;
    // Use worker.CancellationPending, worker.ReportProgress(int) and e.Result
}
```

Exercise

.\Module02-AsyncBasics\Exercises

Implement methods from `ThreadPoolExercises.Core` project to pass tests from `ThreadPoolExercises.Tests`. You can use `ThreadPoolExercises` as a playground.

In the end, use `ThreadPoolExercises.Benchmarks` to benchmark implemented methods.