

Module 05:

"The Thread Pool"



TEKNOLOGISK
INSTITUT

Agenda

- ▶ **The Thread Pool**
- ▶ Asynchronous Delegates
- ▶ Timers

The Thread Pool



["Parade-13"](#) by [azkid2lt](#) is licensed under [CC BY-NC-ND 2.0](#)

The Thread Pool

- ▶ A regiment of virtual threads at your disposal
- ▶ Avoids the overhead of creating and managing threads manually
 - Resource consumption
 - Timing overhead at creation
 - No administrative burdens
 - Automatic queueing and “lining up” for more work
- ▶ Limits the amount of worker threads running concurrently



Use Cases for Thread Pool

- ▶ Explicitly queuing using ThreadPool class
- ▶ Invoking asynchronous delegates
- ▶ Timer ticks

- ▶ WPF, WCF, ASP.NET frameworks

- ▶ Using Task Parallel Library (Day 2)
 - Parallel class
 - PLINQ
 - CPU-bound tasks

Up shortly

Later...

Using the Thread Pool

- ▶ Explicitly queue for **ThreadPool** execution

```
for (int i = 0; i < 100; i++)  
{  
    ThreadPool.QueueUserWorkItem(Go); // <-- Parameter is null  
    ThreadPool.QueueUserWorkItem(Go, i);  
    ThreadPool.QueueUserWorkItem(_ => Console.WriteLine(i));  
}
```

```
void Go(object parameter)  
{  
    Console.WriteLine($"{number} printed");  
}
```

- ▶ Executed **WaitCallback** should be "small and quick"

Thread Pool Important Facts

- ▶ Pooled threads are always background threads
- ▶ Thread priority can be changed temporarily
 - Will revert back to Normal when recycled
- ▶ Cumbersome to debug
 - **Name** property cannot be set
 - **Thread.CurrentThread.IsThreadPoolThread** property
 - But short-lived anyway...!

Optimizing the Thread Pool

- ▶ The Thread Pool has advanced heuristics for adding and removing thread pool threads when it sees fit

```
ThreadPool.GetAvailableThreads( out int availWT, out int availCPT);
```

```
ThreadPool.GetMaxThreads(out int maxWT, out int maxCPT);
```

```
ThreadPool.GetMinThreads(out int minWT, out int minCPT);
```

- ▶ Actual min/max thread counts depend upon
 - hardware,
 - CLR version,
 - hosting environment,
 - ...
- ▶ You should probably never touch these numbers directly!

Best Practices for using Thread Pool

▶ Do...

- Keep executed methods short-lived

▶ Don't...

- Block (too many) thread pool threads!
- Interfere with Thread Pool intrinsics by invoking
 - `ThreadPool.SetMinThreads()`
 - `ThreadPool.SetMaxThreads()`



Agenda

- ▶ Thread Pool
- ▶ **Asynchronous Delegates**
- ▶ Timers

Synchronous Delegate Invocation

- ▶ Usual delegate invocation is synchronous

```
Func<int, int, int> del = Add;  
int result = del.Invoke(42, 87);  
Console.WriteLine(result);
```

```
int Add(int a, int b)  
{  
    return a + b;  
}
```

Asynchronous Delegate Invocation

- ▶ All delegates can be invoked asynchronously as well

```
Func<int, int, int> del = Add;  
IAsyncResult state = del.BeginInvoke(42, 87, null, null);  
  
// ...  
  
int result = del.EndInvoke(state);  
Console.WriteLine(result);
```

```
int Add(int a, int b)  
{  
    return a + b;  
}
```

BeginInvoke() and EndInvoke()

▶ BeginInvoke

- Starts computation on thread pool thread.
- Returns **IAsyncResult** to caller immediately ("ticket")

▶ EndInvoke

- Waits for the computation to finish, i.e. blocks!
- Receives return (and **out** and **ref**) value(s)
- Throws any unhandled exception from computation to caller

Asynchronous Callback

- ▶ All delegates can be invoked "nonblocking" with callback

```
Func<int, int, int> del = Add;  
IAsyncResult state = del.BeginInvoke(42, 87, Callback, del);  
  
// ...
```

```
static void Callback( IAsyncResult state )  
{  
    var del = state.AsyncState as Func<int,int,int>;  
    int result = del.EndInvoke(state);  
  
    Console.WriteLine(result);  
}
```



Agenda

- ▶ Thread Pool
- ▶ Asynchronous Delegates
- ▶ **Timers**

An Overview of Timers

- ▶ We will see three interesting timer classes
- ▶ ThreadPool-based:
 - `System.Threading.Timer`
 - `System.Timers.Timer`
- ▶ Dispatcher-based:
 - `System.Windows.Threading.DispatcherTimer`

Up shortly

Case Study A

System.Threading.Timer

- ▶ A simplistic timer

```
Timer timer = new Timer(  
    OnTimerCallback,  
    null,  
    2000,  
    1000  
);
```

```
void OnTimerCallback(object data)  
{  
    Console.WriteLine( $"The time is {DateTime.Now}");  
}
```

- ▶ Keeps ticking
 - Use **Timer.Change()** to change intervals (or disable) timer

System.Timers.Timer

- ▶ Adds functionality to **System.Threading.Timer**

```
Timer timer = new Timer( 3000 );  
timer.Elapsed += OnTimerElapsed; // CLR event  
timer.AutoReset = false;         // <-- Raise only once  
timer.Start();  
...  
timer.Stop();
```

```
void OnTimerElapsed(object sender, ElapsedEventArgs e)  
{  
    Console.WriteLine($"The time is {e.SignalTime}");  
}
```

- ▶ Also contains **SynchronizingObject** property for WPF



Summary

- ▶ Thread Pool
- ▶ Asynchronous Delegates
- ▶ Timers



WINCUBATE

Jesper Gulmann Henriksen

PhD, MCT, MCSD, MCPD

Phone : +45 22 12 36 31

Email : jgh@wincubate.net

WWW : <http://www.wincubate.net>

Ringgårdsvej 4A

8270 Højbjerg

Denmark