Concurrent Programming in .NET

Wintellect Core Services

Microsoft Regional Director

Consulting
Custom software application development and architecture

Instructor Led Training
Microsoft's #1 training vendor for DevDiv for 14 years

On-Demand Training
World class, subscription based online training

2011 MICROSOFT. PARTNER OF THE YEAR WINNER Microsoft Partner Network

Microsoft GOLD CERTIFIED Partner

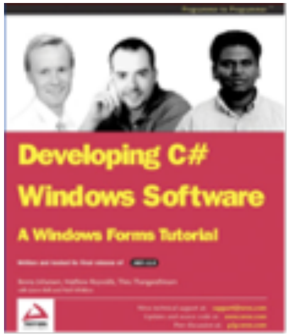
Xamarin PREMIER CONSULTING PARTNER

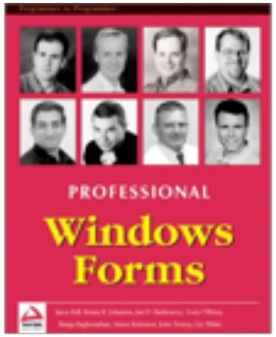
Xamarin Certified Mobile Developer

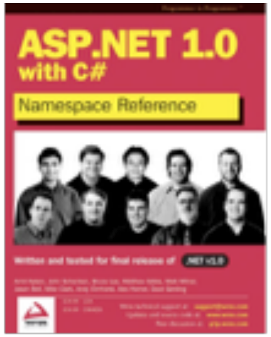
Jason Bell


TEXAS INSTRUMENTS
HOME COMPUTER
READY-PRESS ANY KEY TO BEGIN
©1981 TEXAS INSTRUMENTS

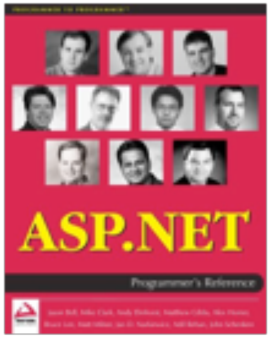
Jason Bell


Developing C#
Windows Software
A Windows Forms Tutorial


PROFESSIONAL
Windows
Forms


ASP.NET 1.0
with C#
Namespace Reference


ASP.NET 1.0
with VB.NET
Namespace Reference


ASP.NET
Programmer's Reference


msdn
magazine
ASP.NET
Windows XP
Security

GitHub

github.com/
Wintellect/WintellectWebinars

Agenda

- Concurrency vs. Parallelism
- Threads
- CPU-Bound vs I/O-Bound
- Tasks in .NET
- Async/Await
- Q & A

Concurrency vs. Parallelism

- Operations that are being executed concurrently are **in-progress** at the same time
- Operations that are being executed in parallel are actually **executing** at the same time

Concurrency vs. Parallelism

- Concurrency can keep a GUI application **responsive**
  - GUI thread can still respond to events while other work is performed
- Concurrency is used in Web applications for **scalability**
  - One process per application, one thread per request
- Parallelism can by used to perform a task in less time
  - "Divide and conquer"

Threads

- A **thread** is an operating system construct used to encapsulate a unit of executable work
- Threads are **pre-emptively scheduled** for execution on the available processors
  - Scheduler uses a **priority**-based system to determine how threads are scheduled

Threads

- Threads are **expensive** to create and consume memory
- Too many threads leads to excessive **context switching** and hurts performance
- The **.NET Thread Pool** is a collection of threads that are available to be used in your application
  - Using a thread pool thread avoids the overhead associated with creating a dedicated thread

CPU-Bound vs I/O-Bound

- **CPU-bound** tasks can be completed in less time if more CPU resources are used
  - Complex calculations or in-memory processing
  - Potentially a good candidate for **parallelism** (do the work more efficiently)

CPU-Bound vs I/O-Bound

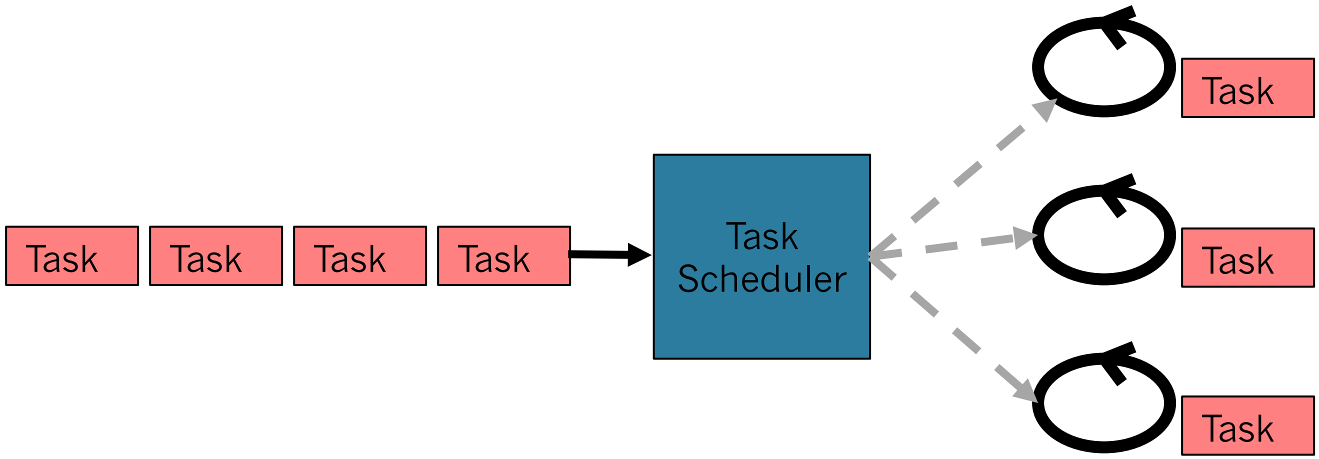
- **I/O-bound** tasks depend on an external resource that involves some latency
  - Disk and network operations
  - A good candidate for concurrency (wait more efficiently)

CPU-Bound vs I/O-Bound

- The operating system provides mechanisms that can be used to efficiently wait for an I/O-bound operation to complete
  - Signals, I/O completion ports, Kqueue
- Goal is to avoid monopolizing a thread simply to wait for an I/O-bound operation to finish (spinning)

Tasks in .NET

- A Task is a schedulable unit of work
  - Wraps a delegate that is the actual work
- Task in queued for execution by a TaskScheduler



Tasks in .NET

- Tasks are created by passing a **delegate** to the constructor
  - Call **Start** to queue the task to the scheduler
  - Can also use **Factory** property

```
Action a = delegate
{
  Console.WriteLine("Hello from task");
};
Task tsk = new Task(a);
tsk.Start();
```

```
Action a = delegate
{
  Console.WriteLine("Hello from task");
};
Task tsk = Task.Factory.StartNew(a);
```

Tasks in .NET

- **Scheduler** maps work on to threads
- Multiple scheduler implementations
- Can control the scheduler to use when starting a task
  - Use **thread pool** threads or **dedicated** threads
  - Use the GUI thread for updating the UI

Tasks in .NET

- Data can be passed to a task using **Action<object>**
- Data can also be passed **implicitly** using anonymous delegate rules

```
Guid jobId = Guid.NewGuid();

Action<object> a = delegate(object state)
{
  Console.WriteLine("{0}: Hello {1}", jobId, state);
};

Task tsk = new Task(a, "World");
tsk.Start();
```

Tasks in .NET

- Generic version of Task available
  - **T** is return type
  - Accessed from the task **Result**

```
Func<object, int> f = delegate(object state)
{
  Console.WriteLine("Hello {0}", state);
  return 42;
};
Task<int> tsk = new Task<int>(f, "World");
tsk.Start();
//...
Console.WriteLine("Task return is: {0}", tsk.Result);
```

Tasks in .NET

- Can wait for one or more tasks to end using **Wait**, **WaitAll** or **WaitAny**
- Can pass **timeout** for wait

```
Task t = new Task( DoWork );
t.Start();
t.Wait();
```

```
Task t1 = new Task( DoWork );
t1.Start();
Task t2 = new Task( DoOtherWork );
t2.Start();

if (!Task.WaitAll(new Task[]{t1, t2}, 2000))
{
  Console.WriteLine( "wait timed out" );
}
```

Tasks in .NET

- Tasks can end in one of three states
  - RanToCompletion: everything completed normally
  - Canceled: task was cancelled
  - Faulted: an unhandled exception occurred on the task
- Unhandled Exceptions get thrown when waiting on a task

```
Task t = new
Task(DoWork);
t.Start();

if (!t.Wait(1000))
{
}
```

```
private static void DoWork()
{
  throw new Exception();
}
```

Tasks in .NET

- Tasks support cancellation
  - Modelled by **CancellationToken**
- Token can be passed into many APIs

```
CancellationTokenSource source = new CancellationTokenSource();

Task t1 = new Task( DoWork, source.Token );
t1.Start();

t1.Wait(source.Token);
```

Tasks in .NET

- CancellationTokenSource has **Cancel** method to trigger the cancellation of tasks and blocking APIs

```
CancellationTokenSource source = new CancellationTokenSource();

Task t1 = new Task( DoWork, source.Token );
t1.Start();

source.Cancel();
```

Tasks in .NET

- Cancellation has different effects depending on state of task
  - Unscheduled tasks are never run
  - Scheduled tasks must **cooperate** to end

```
private static void DoWork(object o)
{
  CancellationToken tok = (CancellationToken)o;

  while (true)
  {
    Console.WriteLine("Working ...");
    Thread.Sleep(1000);
    tok.ThrowIfCancellationRequested();
  }
}
```

Tasks in .NET

- Tasks can be chained together by using **ContinueWith**
- New task will be scheduled when previous one finishes

```
Task t = new Task(DoWork);

t.ContinueWith(tPrev => Console.WriteLine(tPrev.Status));
t.Start();
```

Tasks in .NET

- Tasks can be chained depending on the outcome of the previous task
  - RunToCompletion, Canceled, Faulted
- **TaskContinuationOptions** flag passed to ContinueWith

```
Task<int> t = new Task<int>(GetData);

t.ContinueWith(ProcessData,
                   TaskContinuationOptions.OnlyOnRanToCompletion);

t.Start();
```

Async I/O

- Some .NET APIs model async I/O
  - No thread is consumed while I/O-bound operation takes place
  - Uses IO Completion ports
- APIs that end with "Async" return a Task

Async/Await

- C# now has the **async** and **await** keywords
- Enables continuations whilst maintaining the readability of sequential code
- Built around Task and Task<T>

```
private async void Button_Click(object sender, RoutedEventArgs e) {
   calcButton.IsEnabled = false;
   Task<double> piResult = CalcPiAsync(1000000000);

    // If piResult not ready , return from method, allowing UI to continue
    double pi = await piResult;
   // piResult now available continues to run on UI thread

    calcButton.IsEnabled = true;
    this.pi.Text = pi.ToString();
 }
```

Conclusion

- Evaluation
- Recording of this webinar

Q & A