Module 07

"Asynchronous Programming"

Agenda

- Introducing Task Parallel Library
- Combining and Handling Tasks
- Threading Issues
- Lab 7
- Discussion and Review

Task Parallel Library

- Task Parallel Library (TPL)
  - Was introduced in .NET 4.0
  - Enhanced in .NET 4.5
    - Special keywords are included in C# 5.0
- Features
  - Task Parallelism
  - Data Parallelism
  - Parallel LINQ
  - Thread-safe collections
- Emerging trends leverage parallelism! Also .NET!

Creating Tasks

- The Task class captures a unit of computation
- Initialized from constructor using a computation described by
  - Action delegate
  - Anonymous method
  - Lambda expression (usually preferred)

```
Task task = new Task( () =>
   Console.WriteLine( "Hello World from Task Parallel Library" )
);
```

- Note: Does not run automatically when created!

Task Execution

- Three approaches to starting tasks
  - Create `Task` object and invoke `Task.Start()`
  - Use `Task.Factory.StartNew()` static
  - Use `Task.Run()` static

```
Task task = Task.Factory.StartNew( () =>
{
   for ( int i = 1 ; i < 100 ; i += 2 )
   {
      Console.WriteLine( "\t" + i );
   }
});
```

- Usually one of the last two options is employed

Waiting for Task Completion

- Tasks can be awaited
  - `Task.Wait()`
  - `Task.WaitAny()` static
  - `Task.WaitAll()` static

```
Task task1 = ...;
Task task2 = ...;
Task task3 = ...;

task1.Wait();

Task.WaitAny( task1, task2, task3 );

Task.WaitAll( task1, task2, task3 );
```

Tasks with Results

- `Task<T>`
  - captures a task returning a result of type `T`
- `Task.Run<T>()` and `Task.StartNew<T>()` also exist

```
Task<DateTime> t = Task.Run<DateTime>( () => DateTime.Now );

Console.WriteLine( t.Result );
```

- Result can be explicitly retrieved via `Task.Result`
  - Note: This property is blocks when task is not yet completed!

Cancelling Tasks

- Running tasks can be requested cancelled
  - Signal token created by `CancellationTokenSource` class
  - Other code signal token supplied to task
- Task method then
  - Checks if cancellation is requested
  - Throws `OperationCanceledException` to accept cancellation

```
task = Task.Factory.StartNew( () =>
{  ...
   if( token.IsCancellationRequested )
   {
      throw new OperationCanceledException( token );
   }
}
```

- Check task running status via `Task.Status`

The `Parallel` Class

- The `Parallel` class leverages data parallelism
- `Parallel.`
  - `Invoke()` invokes actions in parallel
  - `For()` is a parallel `for`-loop
  - `ForEach()` is a parallel `foreach`-loop

```
Parallel.For( 0, 1000, i =>
   Console.WriteLine( $"Executing number {i,4}..." )
);
```

- Developer's responsibility that iterations are in fact independent

Additional Parallel Options

- Options and refinements are provided through various overloads
  - The `ParallelLoopState` and `ParallelLoopResult` classes

```
ParallelLoopResult result = Parallel.For( 0, 1000, ( i, state ) =>
{
   if( i == 87 )
   {
      state.Break();
   }
   ...
}
```

  - The `ParallelOptions` class
    - `MaxDegreeOfParallelism`
    - `CancellationToken`

Parallel LINQ

- PLINQ = Parallel LINQ
  - `ParallelEnumerable` class is defined in `System.Linq` namespace
- `ParallelEnumerable`
  - `AsParallel<T>()`
  - `AsSequential<T>()`
  - `WithCancellation<T>()`
  - `WithDegreeOfParallelism<T>()`

```
var even = from i in numbers.AsParallel()
                            .WithCancellation( token )
                            .WithDegreeOfParallelism( 2 )
           where i % 2 == 0
           select i;
```

Agenda

- Introducing Task Parallel Library
- Combining and Handling Tasks
- Threading Issues
- Lab 7
- Discussion and Review

Combining Tasks

- Tasks can be combined using `Task.ContinueWith()`

```
Task<DateTime> t1 = new Task<DateTime>( () =>
   DateTime.Now );
Task<string> t2 = t1.ContinueWith( previous =>
   $"The time is {previous.Result}!" );

t1.Start();
Console.WriteLine( t2.Result );
```

- Combinators include
  - `Task.WhenAll()` Completes when all tasks have completed
  - `Task.WhenAny()` Completes when any of the tasks completes
  - `Task.Delay()` Completes after a specified time span

- `TaskCreationOptions` allows the creation of child tasks

TaskContinuationOptions

- The behavior of `Task.ContinueWith()` and `Task<T>.ContinueWith()` can be refined

- `TaskContinuationOptions` enumeration supplied in overloads
  - `None`
  - `OnlyOnCanceled`
  - `OnlyOnFaulted`
  - `OnlyOnRanToCompletion`
  - `NotOnCanceled`
  - `NotOnFaulted`
  - `NotOnRanToCompletion`
  - …

Task Exceptions

- Task exceptions are thrown when
  - Waiting for task
  - Getting result for task
- `AggregateException` instances are thrown
  - Consists of a number of inner exceptions
  - **`Flatten()`** is important!

```
try
{
   t.Wait();
}
catch ( AggregateException ae )
{
   foreach( Exception e in ae.InnerExceptions )
   {
      Console.WriteLine( e.Message );
   }
}
```

C# 5.0 **`await`** Operator

- C# 5.0 introduces **`await`** keyword for methods returning `Task` or `Task<T>`
  - Yields control until awaited task completes
  - Results gets returned
- Allows you to program just like for synchronous programming…!

```
WebClient client = new WebClient();
string result = await client.DownloadStringTaskAsync( ... );

Console.WriteLine( result );
```

- Really complex control flow under the hood is made stunningly simple by compiler

C# 5.0 **async** Modifier

- C# 5.0 introduces **async** keyword
  - Marks method or lambda as asynchronous
  - Note: Methods making use of **await** must be marked "**async**"
- You can now easily define your own asynchronous methods

```
async static void DoStuff()
{
   // ...

   string result = await client.DownloadStringTaskAsync( ... );

   // ...
}
```

- Can create async methods returning **void**, **Task**, or **Task<T>**

Exceptions Thrown by Tasks and Awaitable Methods

- Observe and catch exceptions "as usual" when awaiting tasks

```
try
{
   string data = await client.DownloadStringTaskAsync( ... );
}
catch ( WebException ex ) { ... }
```

- Note that
  - **Task.WaitXxx()** throws an **AggregateException**
  - **Task.Result** throws an **AggregateException**
  - Awaiting a **Task** throws exceptions "as usual", however!

Unobserved Task Exceptions

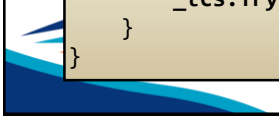
- Subscribe to unobserved exceptions through the `TaskScheduler.UnobservedTaskException` event

```
TaskScheduler.UnobservedTaskException +=
   ( object s, UnobservedTaskExceptionEventArgs ute ) => {
      foreach( Exception e in ute.Exception.InnerExceptions )
      {
          ...
      };
```

TaskCompletionSource<T>

- Any occurrence or computation can be transformed into a `Task<T>` using `TaskCompletionSource<T>`

```
public partial class Form1 : Form
{
    TaskCompletionSource<DateTime> _tcs =
       new TaskCompletionSource<DateTime>();
    ...
    async private void OnClick(object sender, EventArgs e)
    {
        DateTime dt = await _tcs.Task;
        ...
    }
    private void OnMouseEnter(object sender, EventArgs e)
    {
        _tcs.TrySetResult(DateTime.Now);
    }
}
```



Three Approaches to Asynchrony

- Synchronous calls
  - `Xxx()` methods
- .NET Asynchronous Programming Model (APM) consisting of
  - `BeginXxx()` methods
  - `EndXxx()` methods
- Event-based Asynchronous Pattern (EAP) consisting of
  - `XxxAsync()` methods
  - `XxxCancelAsync()` methods
  - `XxxCompleted` events
- Task-based Asynchronous Pattern
  - `XxxAsync()` or `XxxTaskAsync()` methods

Tasks and Asynchronous Programming Model

- The "traditional" .NET Asynchronous Programming Model consists of
  - `BeginXxx()` methods
  - `EndXxx()` methods
- Tasks encapsulate this model using `TaskFactory.FromAsync()`

```
HttpWebResponse response =
   await Task<WebResponse>.Factory.FromAsync(
      request.BeginGetResponse,
      request.EndGetResponse,
      request )
   as HttpWebResponse;
```

When to Use What?

- Thread
  - Avoid if possible!
  - Only for "eternal" processing
- ThreadPool
  - Use for very quick, small, unordered computations
  - Usually callbacks
- Task
  - Use for "task parallelism": computational independence or I/O-bound work
- Parallel
  - Use for "data parallelism": processing sets of independent data

Agenda

- Introducing Task Parallel Library
- Combining and Handling Tasks
- Threading Issues
- Lab 7
- Discussion and Review

Synchronizing Tasks

- Processor and operating system schedule tasks in and out repeatedly
  - Thread context switch can occur at any time
    - Even in the middle of assignments and increments etc.
- Hence computations need to be computationally safe
  - Some operations must be performed indivisibly!
  - Race conditions should be avoided
- Basically two solutions
  - Synchronizing access to critical regions of code
  - Signaling between threads

The **Monitor** Class

- The **Monitor** class is a light-weight mechanism for use within a single process
  - `Monitor.Enter` static
  - `Monitor.TryEnter` static
  - `Monitor.Exit` static
- The **lock** keyword in C# is based on **Monitor** and **try-finally**

```
object syncObject = new object();
...
lock( syncObject )
{
   _counter++;
}
```

- Note: **lock** can only lock reference types…!

Wait Handles and Events

- The `WaitHandle` class
  - Facilitates waiting on certain handles (or "flags" being raised)
- `WaitHandle` methods
  - `WaitOne()` static
  - `WaitAny()` static
  - `WaitAll()` static
- `WaitHandle`-based classes
  - `ManualResetEvent`
  - `AutoResetEvent`
  - `Mutex`
  - `Semaphore`
  - …

Concurrent Collections

- Thread-safe collection alternatives are provided in the `System.Collections.Concurrent` namespace
  - `ConcurrentQueue<T>`
  - `ConcurrentStack<T>`
  - `ConcurrentDictionary<K,V>`
  - `ConcurrentBag<T>`

```
ConcurrentQueue<int> queue = new ConcurrentQueue<int>();

Task producer = Task.Factory.StartNew( () => { ...
   queue.Enqueue( DateTime.Now.Milliseconds );
   ...
}

Task consumer = Task.Factory.StartNew( () => { ...
   int number;
   if( queue.TryDequeue( out number ) ) { ... }
}
```

BlockingCollection<T>

- BlockingCollection<T>
  - Concurrent collection
  - Optional bounded capacity
  - Blocking operations

```
BlockingCollection<int> bc = new BlockingCollection<int>( 5 );
...
string result = string.Format( $"Successfully took {0}",
   await Task.Run<int>( () => bc.Take() );
```

- Implement your own concurrent collection using
  - IProducerConsumerCollection<T>

Quiz: Asynchronous Programming – Right or Wrong?


```
await Console.WriteLine( "Hello, World" );
```

```
WebClient client = new WebClient();
await client.DownloadFile(
   "http://www.wincubate.net/BusinessCard.jpg"
);
```

```
WebClient client = new WebClient();
await client.DownloadFileTaskAsync(
   "http://www.wincubate.net/BusinessCard.jpg"
);
```

```
static void FetchImage( string url, string localFileName )
{
   using ( WebClient client = new WebClient() )
   {
      await client.DownloadFileTaskAsync( url, localFileName );
   }
}
```

Lab 7: Creating and Controlling Tasks and Threads

- Lab 7.1 – 7.3

Discussion and Review

- Introducing Task Parallel Library
- Combining and Handling Tasks
- Threading Issues

WINCUBATE


WINCUBATE
Jesper Gulmann Henriksen
PhD, MCT, MCSD, MCPD
Phone : +45 22 12 36 31
Email : jgh@wincubate.net
WWW : http://www.wincubate.net
Hasselvangen 243
8355 Solbjerg
Denmark