Lecture 14 – Tasks

Learning Goals

- The difference between **Task** and **Thread**
- Explain **IO Bound** Operations
- How to **create** and **execute** Tasks
- How to **Synchronize** Tasks

Task vs. Thread

- Threads are the most low level constructs in multithreading
  - Cannot return values
  - Use shared variables amongst the threads to pass values
- Tasks are higher level abstraction of multithreading
  - capable of returning values
  - can be chained
  - can use thread pool
  - useful for IO bound operations

IO Bound Operation

Imagine processes in which not only local CPU is being used, but there are calls from the process to other servers for some external inputs. eg.

- Our local process calls a service in AWS Cloud to make some computation for us
- Our local process calls google cloud to do some search for us
- Our local process queries an external cloud DB

IO Bound Operation

IO Bound Operations:

- Out-of-process calls
- Operations can take an indeterminate amount of time
- Releasing the local resources while waiting for a response

Task Class

- Namespace: `System.Threading.Tasks`
- Constructor: `Task(Action)`
  - Param 1: specific action
  - There are more constructor overloads. See here
- Properties:
  - `Id`: Get the id of this task
  - `IsCompleted`: Gets a value which indicate if the task is completed
  - `Status`: Get the status of the task
  - `CurrentID`: Get the Id of the currently executing task
  - `Factory:` Provides access to factory methods for creating and configuring Task instances

Task Class

- Common Methods:
  - `Start()`: Start the task in a task pool
  - `Wait()`: Wait for the task to complete
  - `Dispose()`:Release all resources used by the current taskTask
  - `Delay(Int32)`: Start the task with a delay in seconds
  - More methods here

Instantiate and Start a Task

```
Task task = new Task(MethodA);          // Instantiate a new task
task.Start();                           // Start the Task

private static void MethodA()
 {
        Console.WriteLine("Hello World");
  }
```

Task with Returning Value

- Namespace: `System.Threading.Tasks`
- Constructor: `Task<TResult>(Func<TResult>)`
  - Param 1: A function which returns some results of type TResult
  - There are more constructor overloads. See here
- Properties:
  - `Id`: Get the id of this task
  - `IsCompleted`: Gets a value which indicate if the task is completed
  - `Status`: Get the status of the task
  - `CurrentID`: Get the Id of the currently executing task
  - `Factory:` Provides access to factory methods for creating and configuring Task<TResults> instances

Task with Returning Value

- Common Methods:
  - `Start()`: Start the task
  - `Wait()`: Wait for the task to complete
  - `Dispose()`:Release all resources used by the current taskTask
  - `Delay(Int32)`: Start the task with a delay in seconds

Task with Returning Value

```
Task<string> task = new Task<string>(MethodB);   // Instantiate a new task
task.Start();                                    // Start the task
task.Wait();                                     // Wait till the task is done
Console.WriteLine(task.Result);                  // Output the result in the console

private static string MethodB()                  // A method which returns string
 {
        return "Return me to the main thread";
  }
```

Task IO Bound Operation

```
Task<string> task = Task.Factory.StartNew<string>
            (() => GetPosts("https://jsonplaceholder.typicode.com/posts"));
SomethingElse();
task.Wait();
Console.WriteLine(task.Result);
//methods
private static void SomethingElse(){ //Implementation of a function  }
private static string GetPosts(string url)
   {  using (var client = new System.Net.WebClient())
        {
           return client.DownloadString(url);
        }
    }
```

Task IO Bound Operation

Takeaways:

- `Task.Factory.StartNew<string>()`:
  - Using Factory property of the Task; more efficient than `Task.Start<string>()`
  - It is better for synchronization between tasks. Instead of two steps, one step is used, i.e.
    - `Task<string> task = new Task<string>()`
    - `task.Start()`
- `Task<string>:` the task expects a string returned from the function
- `task.Wait()`: Using the wait method on task before calling the Results property
- `task.Result:` Using the Result property of task

Task Chaining

- In asynchronous programming, we aim to invoke a subsequent operation after the completion of the previous operation
- Continuations allow decedent operation to consume the results of the first operation.
- A continuation task (also known just as a continuation) is an asynchronous task that's invoked by another task, known as the antecedent, when the antecedent finishes.

Task Chaining Cont.

- Continuations are relatively easy to use, powerful, and flexible
- A continuation is a task that is created in the WaitingForActivation state. It is activated automatically when its antecedent task or tasks complete.
- Calling Task.Start on a continuation in user code throws an System.InvalidOperationException exception

Task Chaining Benefits

- Pass data from the antecedent to the continuation
- Specify the precise conditions under which the continuation will be invoked or not invoked
- Cancel a continuation either before it starts or cooperatively as it is running.
- Invoke multiple continuations from the same antecedent

Task Chaining Benefits Cont.

- Invoke one continuation when all or any one of multiple antecedents complete
- Chain continuations one after another to any arbitrary length
- Use a continuation to handle exceptions thrown by the antecedent

Task Chaining

```
Task<string> antecedent = Task.Run(() =>
{
    return DateTime.Today.ToShortDateString();
});

Task<string> continuation = antecedent.ContinueWith(x =>
{
    return "Today is " + antecedent.Result;
});
```

Task Chaining

Takeaways:

- `Task.Run()`: Instead of using Task.Start() or Task.Factory.StartNew(). This is a task method introduced in .Net 4.5 and after. For ease of use with Tasks
- `antecedent.ContinueWith()`
  - Waits until `antecedent` task finished execution and then continue with its returned value
- `() =>{return DateTime.Today.ToShortDateString();}`
  - A lambda expression which takes no input value and returns a string
- `x =>{return "Today is " + antecedent.Result;}`
  - A lambda expression which takes an input value and returns a string
- `antecedent.Result`
  - Property of a task. Return the task value

Task Creation and Execution Separation

- The Task class also provides constructors that initialize the task but that do not schedule it for execution.
- For performance reasons, the `Task.Run` or `TaskFactory.StartNew` methods are the preferred mechanism for creating and scheduling computational tasks
- For scenarios where creation and scheduling must be separated, you can use the constructors and then call the `Task.Start` method to schedule the task for execution at a later time.

Homework

Clone the Lecture Example repository and exercise with different ways to:

- Create tasks
- Execute tasks
- Synchronize tasks

Look at all the examples in this link