Laboratorio

Ejecutando tareas en paralelo

Versión: 1.0.0
Septiembre de 2017

Documento creado por:
TI Capacitacion para uso personal de:
Franco Morales franco.e.morales@gmail.com
No se permiten copias SIN autorización.

CONTENIDO

Documento creado por:
TI Capacitacion para uso personal de:
Franco Morales franco.e.morales@gmail.com
No se permiten copias SIN autorización.

Introducción

Un tema importante relacionado con el manejo de tareas, es sin lugar a dudas, la ejecución de tareas en paralelo. La biblioteca ***Task Parallel*** incluye una clase estática llamada ***Parallel***. La clase ***Parallel*** proporciona diversos métodos que se pueden utilizar para ejecutar tareas simultáneamente.

En este laboratorio, utilizarás la biblioteca de clases ***Task Parallel*** para a ejecutar tareas en paralelo, enlazar tareas y manejar las excepciones que puedan generarse durante la ejecución de tareas.

Objetivos

Al finalizar este laboratorio, los participantes serán capaces de:

- Ejecutar tareas en paralelo.
- Ejecutar iteraciones de ciclo en paralelo.
- Utilizar *Parallel LINQ*.
- Enlazar tareas.
- Manejar excepciones en tareas.

Documento creado por:
TI Capacitación para uso personal de:
Franco Morales franco.e.morales@gmail.com
No se permiten copias SIN autorización.

Requisitos

Para la realización de este laboratorio es necesario contar con lo siguiente:

- Un equipo de desarrollo con Visual Studio. Los pasos descritos en este laboratorio fueron diseñados con Visual Studio Enterprise 2017 sobre una máquina con Windows 10 Pro.

Tiempo estimado para completar este laboratorio: **60 minutos**.

Ejercicio 1: Ejecutando un conjunto de tareas de forma simultánea

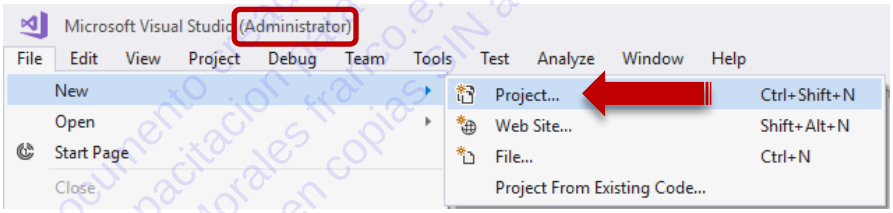
En este ejercicio, utilizarás la biblioteca *Task Parallel* para ejecutar tareas en paralelo, ejecutar iteraciones de ciclo en paralelo y utilizar consultas en paralelo con *Parallel LINQ*.

Tarea 1. Crear una aplicación de Consola.

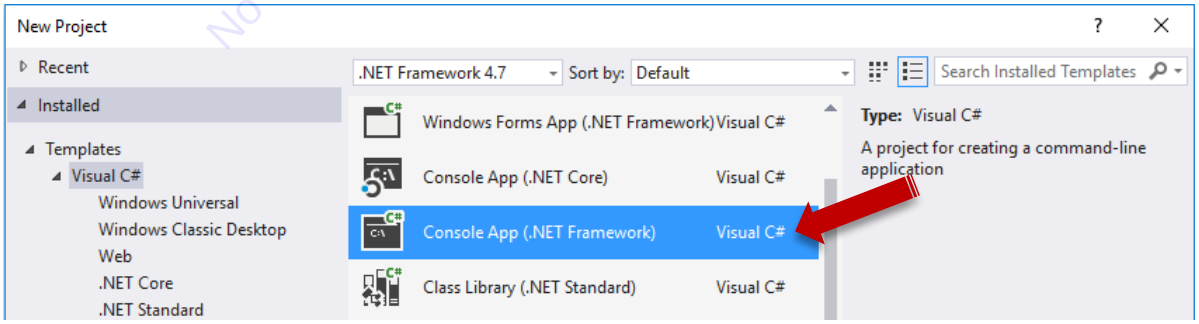
En esta tarea, crearás una aplicación de consola que será utilizada para realizar cada una de las tareas de este laboratorio.

Realiza los siguientes pasos para crear una aplicación de Consola.

1. Abre Visual Studio en el contexto del Administrador.
2. Selecciona la opción ***File*** > ***New*** > ***Project***.



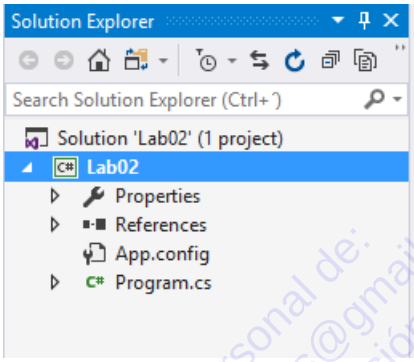
3. En la ventana ***New Project*** selecciona la plantilla ***Console App (.NET Framework)*** para crear una nueva aplicación de Consola.



4. Asigna un nombre al proyecto y haz clic en ***OK*** para crear la solución.

Documento diseñado por: TI Capacitación. Para uso personal de: Franco Morales franco.e.morales@gmail.com No se permiten copias SIN autorización.

El *Explorador de Soluciones* será similar al siguiente.



Documento creado por: Ti Capacitación para uso personal de: Franco Morales franco.morales@gmail.com No se permiten copias sin autorización.

Tarea 2. Ejecutar tareas simultaneas.

La biblioteca *Task Parallel* incluye una clase estática llamada *Parallel* que proporciona diversos métodos que pueden utilizarse para ejecutar tareas simultáneas.

En esta tarea agregarás código a la clase ***Program*** para ejecutar tareas en paralelo.

1. Haz doble clic sobre el archivo ***Program.cs*** para abrirlo en el editor de código.

2. Agrega el siguiente código al inicio del archivo para importar el espacio de nombres ***System.Threading***.

```
using System.Threading;
```

3. Agrega el siguiente código dentro de la clase ***Program*** para definir el método ***RunParallelTasks***.

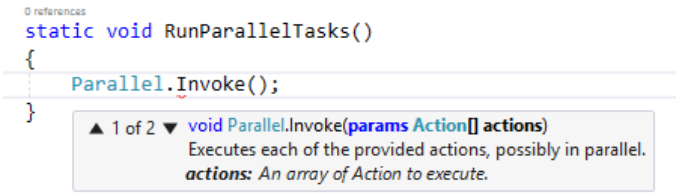
```
static void RunParallelTasks()
{

}
```

Para ejecutar un conjunto fijo de tareas en paralelo, podemos utilizar el método ***Invoke*** de la clase ***Parallel***.

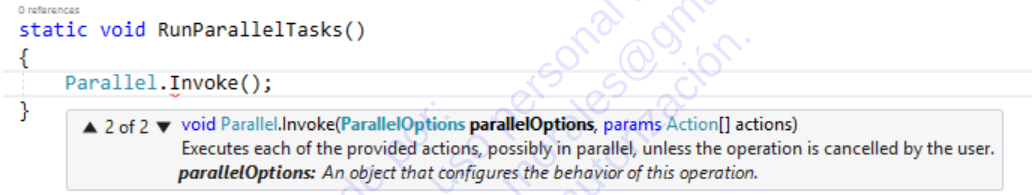
4. Agrega el siguiente código dentro del método *RunParallelTasks* para invocar al método ***Invoke*** de la clase ***Parallel***.

```
static void RunParallelTasks()
{
    Parallel.Invoke();
}
```

Una sobrecarga del método *Invoke* recibe como parámetro una serie de objetos ***Action***.



La otra sobrecarga tiene dos parámetros.



El primer parámetro permite configurar el comportamiento de la operación y el otro parámetro permite proporcionar la serie de objetos ***Action*** que deseamos ejecutar.

Al invocar a estos métodos, podemos utilizar expresiones lambda para especificar las tareas que queremos ejecutar de forma simultánea.

5. Agrega el siguiente código para especificar 3 tareas que deseamos ejecutar en paralelo.

```
static void RunParallelTasks()
{
    Parallel.Invoke(
        () => { },
        () => { },
        () => { }
        );
}
```

No necesitamos crear explícitamente cada tarea, las tareas son creadas implícitamente a partir de los delegados que proporcionamos al método *Parallel.Invoke*.

6. Agrega el siguiente código a la clase ***Program*** para definir un método que nos permitirá ejemplificar la ejecución de tareas en paralelo.

```
static void WriteToConsole(string message)
{
  Console.WriteLine($"{message}. {Thread.CurrentThread.ManagedThreadId}");
  Thread.Sleep(5000); // Simular un proceso de larga duración
  Console.WriteLine($"Fin de la tarea {Thread.CurrentThread.ManagedThreadId}");
}
```

7. Modifica el código del método *RunParallelTasks* para hacer que las tareas ejecuten al método *WriteToConsole*.

```
static void RunParallelTasks()
{
    Parallel.Invoke(
        () => { WriteToConsole("Tarea 1"); },
        () => { WriteToConsole("Tarea 2"); },
        () => { WriteToConsole("Tarea 3"); }
        );
}
```

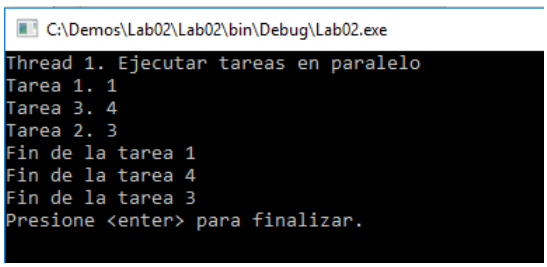
8. Agrega el siguiente código al inicio del método *RunParallelTasks* para indicar el momento en que se invocan las tareas en paralelo.

```
static void RunParallelTasks()
{
    Console.WriteLine(
        $"Thread {Thread.CurrentThread.ManagedThreadId}. Ejecutar tareas en
paralelo");
```

9. Agrega el siguiente código dentro del método *Main* para ejecutar al método *RunParallelTasks.*

```
    RunParallelTasks();
    Console.Write("Presione <enter> para finalizar.");
    Console.ReadLine();
```

10. Ejecuta la aplicación. Podrás notar que el mensaje "**Presione <enter> para finalizar.**" se muestra después de los demás mensajes, esto es, después de que la ejecución de las tareas en paralelo finaliza. El método ***Invoke*** no retorna hasta que todas las tareas hayan finalizado, ya sea de forma normal o por alguna excepción.



Puedes notar también que no hay garantía en el orden en que se ejecutan las tareas.

Documento creado por: Capacitación para uso personal de: Pablo Moráles francisco.e.morales@gmail.com No se permiten copias SIN autorización.

Tarea 3. Ejecutar Iteraciones de Ciclo en Paralelo.

La clase ***Parallel*** de la biblioteca ***Task Parallel***, también proporciona métodos que podemos utilizar para ejecutar iteraciones de ciclos ***for*** y ***foreach*** en paralelo. Es importante mencionar que la ejecución de iteraciones de ciclos en paralelo, no puede ser siempre apropiada. Por ejemplo, si deseamos comparar valores secuenciales, debemos ejecutar secuencialmente las iteraciones del ciclo. Sin embargo, si cada iteración del ciclo representa una operación independiente, ejecutar las iteraciones del ciclo en paralelo, nos permitirá maximizar el uso del poder de procesamiento disponible.

En esta tarea examinaremos los beneficios de la ejecución de iteraciones de ciclo en paralelo.

1. Agrega el siguiente código a la clase ***Program*** para definir un método que nos permita ejemplificar la iteración de ciclo en paralelo.

```
static void ParallelLoopIterate()
{

}
```

   Para ejecutar las iteraciones de un ciclo ***for*** en paralelo, podemos utilizar el método ***Parallel.For***. Este método tiene varias sobrecargas para satisfacer a muchos escenarios diferentes. En su forma más simple, el método ***Parallel.For*** toma tres parámetros:

   - Un parámetro ***Int32*** que representa el índice de inicio de la operación. El valor de este parámetro es el primer valor que se toma para el índice de la iteración.
   - Un parámetro ***Int32*** que representa el índice final de la operación. El ciclo se ejecuta mientas el valor del índice actual sea menor al valor de este parámetro.
   - Un delegado ***Action<Int32>*** que es ejecutado una vez por cada iteración.

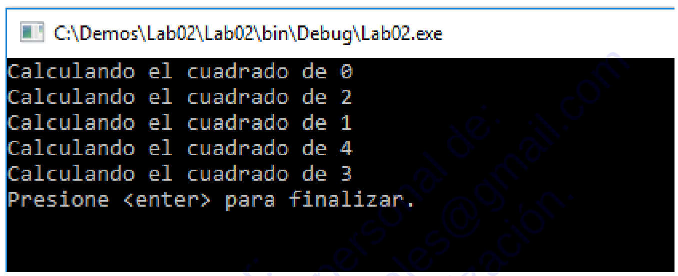
2. Agrega el siguiente código dentro del método *ParallelLoopIterate* para calcular el cuadrado de los números 0, 1, 2, 3 y 4. Los resultados obtenidos serán almacenados en un arreglo.

```
static void ParallelLoopIterate()
{
    int[] SquareNumbers = new int[5];
    Parallel.For(0, 5, i =>
    {
        SquareNumbers[i] = i * i;
        Console.WriteLine($"Calculando el cuadrado de {i}");
    });
}
```

3. Modifica el código del método ***Main*** para ejecutar el método ***ParallelLoopIterate***.

```
static void Main(string[] args)
{
    //RunParallelTasks();
    ParallelLoopIterate();

    Console.Write("Presione <enter> para finalizar.");
    Console.ReadLine();
}
```

Ejecuta la aplicación y observa el resultado. Puedes notar que no existe necesariamente un orden de iteración del ciclo. El cuerpo del ciclo es ejecutado simultáneamente por varias tareas.



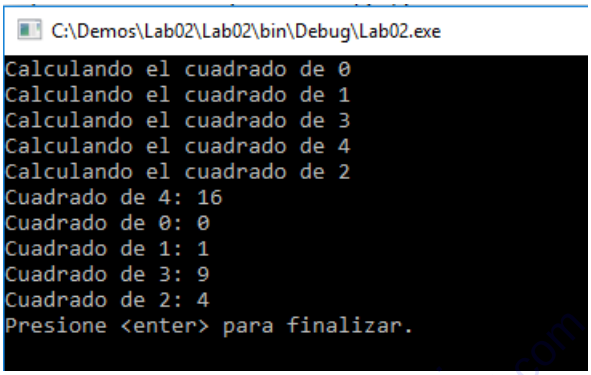
Para ejecutar las iteraciones del ciclo ***foreach*** en paralelo, podemos utilizar el método ***Parallel.Foreach***. Al igual que el método ***Parallel.For***, el método ***Parallel.ForEach*** incluye diferentes sobrecargas. En su forma más simple, el método ***Parallel.ForEach*** toma dos parámetros:

- Una colección ***IEnumerable<TSource>*** sobre la cual deseamos iterar.
- Un delegado ***Action<TSource>*** que es ejecutado una vez por iteración.

4. Agrega el siguiente código al método ***ParallelLoopIterate*** para mostrar los valores del arreglo ***SquareNumbers***.

```
static void ParallelLoopIterate()
{
    int[] SquareNumbers = new int[5];
    Parallel.For(0, 5, i =>
    {
        SquareNumbers[i] = i * i;
        Console.WriteLine($"Calculando el cuadrado de {i}");
    });

    Parallel.ForEach(SquareNumbers, n =>
    {
        Console.WriteLine(
            $"Cuadrado de {Array.IndexOf(SquareNumbers, n)}: {n}");
    });
}
```

5. Ejecuta la aplicación y observa el resultado. Puedes notar que no hay un orden especifico de cómo se fue iterando sobre los elementos del arreglo. Varias tareas fueron ejecutadas para iterar sobre los elementos del arreglo.



Para obtener más información y ejemplos acerca de la ejecución de las operaciones de datos en paralelo, se recomienda consultar los siguientes enlaces:

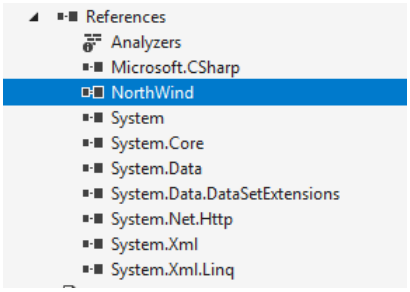
Data Parallelism (Task Parallel Library)
http://go.microsoft.com/fwlink/?LinkID=267839

Parallel.For Method (Int32, Int32, Action<Int32>)
http://msdn.microsoft.com/es-mx/library/dd783539%28v=vs.110%29.aspx

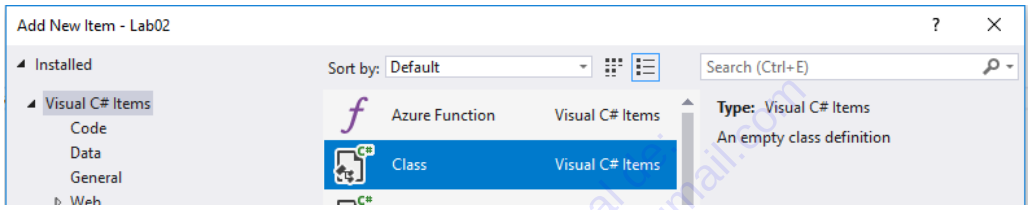
Tarea 4. Utilizando Parallel LINQ.

Parallel LINQ (PLINQ) es una implementación de ***Language - Integrated Query (Lenguaje Integrado de Consultas (LINQ))*** que soporta operaciones en paralelo. En esta tarea examinaremos el uso de ***PLINQ***.

1. Agrega una referencia del ensamblado ***NorthWind.dll*** a la aplicación de consola. El ensamblando ***NorthWind.dll*** se encuentra adjunto a este documento y contiene datos ficticios que utilizarás para ejemplificar el uso de ***PLINQ***. Después de agregar la referencia, el nodo ***References*** del proyecto será similar al siguiente.



2. Agrega al proyecto consola un nuevo archivo de clase llamado ***ProductDTO***.



La clase *ProductDTO* nos permitirá almacenar el resultado de una consulta.

3. Modifica la clase ***ProductDTO*** para agregarle un constructor y unas propiedades.

```
class ProductDTO
{
    public ProductDTO()
    {
        // Simular un proceso de larga duración
        System.Threading.Thread.Sleep(1);
    }
    public int ProductID { get; set; }
    public string ProductName { get; set; }
    public decimal? UnitPrice { get; set; }
    public decimal? UnitsInStock { get; set; }
}
```

4. Agrega el siguiente código dentro de la clase ***Program*** para definir un método que realice una consulta *LINQ* para obtener una lista de objetos ***ProductDTO*** a partir del conjunto de Productos que expone el ensamblado ***NorthWind***.

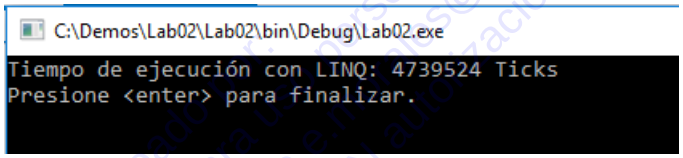
```
static void RunLINQ()
{
    // Declarar una variable para medir el tiempo de ejecución.
    var S = new System.Diagnostics.Stopwatch();
    S.Start();
    var DTOProducts = NorthWind.Repository.Products.Select(p =>
    new ProductDTO
    {
        ProductID = p.ProductID,
        ProductName = p.ProductName,
        UnitPrice = p.UnitPrice,
        UnitsInStock = p.UnitsInStock
```

```
    }).ToList();
    S.Stop();
    Console.WriteLine($"Tiempo de ejecución con LINQ: {S.ElapsedTicks} Ticks");
```

5. Modifica el método ***Main*** para invocar únicamente al método ***RunLINQ***.

```
static void Main(string[] args)
{
    //RunParallelTasks();
    //ParallelLoopIterate();

    RunLINQ();
    Console.Write("Presione <enter> para finalizar.");
    Console.ReadLine();
}
```

Ejecuta la aplicación y observa el tiempo de ejecución en Ticks de la consulta LINQ. Un **Tick** es un intervalo de 100 nanosegundos equivalente a una diez millonésima de segundo. Cada milisegundo está compuesto por 10,000 ticks.



En la mayoría de los casos, la sintaxis ***PLINQ*** es idéntica a la sintaxis *LINQ* tradicional. Al escribir una expresión *LINQ*, podemos optar por *PLINQ* invocando al método de extensión ***AsParallel*** en el origen de datos ***IEnumerable***.

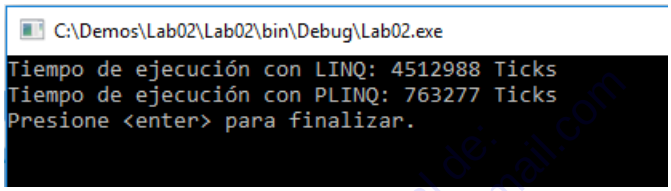
6. Agrega el siguiente código dentro de la clase ***Program*** para definir un método que realice una consulta *PLINQ* para obtener una lista de objetos ***ProductDTO*** a partir del conjunto de Productos que expone el ensamblado ***NorthWind***.

```
static void RunPLINQ()
{
    var S = new System.Diagnostics.Stopwatch();
    S.Start();
    var DTOProducts = NorthWind.Repository.Products.AsParallel().Select(p =>
    new ProductDTO
    {
        ProductID = p.ProductID,
        ProductName = p.ProductName,
        UnitPrice = p.UnitPrice,
        UnitsInStock = p.UnitsInStock
    }).ToList();
    S.Stop();
    Console.WriteLine($"Tiempo de ejecución con PLINQ: {S.ElapsedTicks} Ticks");
}
```

7. Modifica el método ***Main*** para invocar al método ***RunPLINQ***.

```
static void Main(string[] args)
{
    //RunParallelTasks();
    //ParallelLoopIterate();

    RunLINQ();
    RunPLINQ();
    Console.Write("Presione <enter> para finalizar.");
    Console.ReadLine();
}
```

8. Ejecuta la aplicación y observa el resultado.



Puedes notar que el tiempo de ejecución de la consulta con PLINQ fue mucho menor que el tiempo de ejecución de la consulta con LINQ. En el ejemplo que se muestra en la imagen anterior, la diferencia en Ticks es: 4,512,988 – 763,277 = 3,749,711 ticks. Casi 6 veces más rápido con PLINQ.

Para obtener más información sobre ***PLINQ***, se recomienda consultar el siguiente enlace:

Parallel LINQ (PLINQ)

http://go.microsoft.com/fwlink/?LinkID=267840

Documento creado por el Centro de Capacitación para uso personal de Franco Morales franco.e.morales@gmail.com. No se permiten copias SIN autorización.

Ejercicio 2: Enlazando Tareas

En la programación asíncrona es muy común que cuando una operación asíncrona finalice, una segunda operación sea invocada proporcionándole algún tipo de información. Adicionalmente, una tarea puede desencadenar otras tareas si el trabajo que necesita realizar es también de múltiples hilos por naturaleza.

En este ejercicio exploraremos la forma de enlazar la ejecución de tareas.

Tarea 1. Crear Tareas de Continuación.

Las ***Tareas de Continuación (Continuation Task)*** permiten encadenar varias tareas juntas para que se ejecuten una tras otra. La tarea que, al finalizar invoca a otra tarea, es conocida como ***Antecedente*** y la tarea que esta invoca, es conocida como ***Continuación***. Podemos pasar datos desde la tarea ***Antecedente*** hacia la tarea de ***Continuación*** y podemos controlar la ejecución de la cadena de tareas de varias formas.

En esta tarea ejemplificaremos el uso de las ***Tareas de Continuación***.

1. Agrega el siguiente código a la clase ***Program*** para simular la realización de una operación de larga duración.

```
static List<string> GetProductNames()
{
    // Simular un proceso de larga duración.
    Thread.Sleep(3000);
    return NorthWind.Repository.Products.Select(p => p.ProductName).ToList();
}
```

   El método devuelve una lista con los nombres de los productos del repositorio de datos NorthWind.

2. Agrega el siguiente código dentro de la clase ***Program*** para definir un método que creé una tarea para invocar al método *GetProductNames*.

```
static void RunContinuationTasks()
{
    var FirstTask =
        new Task<List<string>>(GetProductNames);
}
```

   Si la tarea termina exitosamente, podríamos requerir que una segunda tarea procese los datos obtenidos o, si la tarea falla, podríamos requerir que una segunda tarea realice algún proceso de recuperación.

3. Agrega el siguiente código a la clase ***Program*** para definir un segundo proceso que será ejecutado cuando la primera tarea finalice.

```
static int ProcessData(List<string> productNames)
{
    // Simular procesamiento de datos
    foreach(var ProductName in productNames)
    {
        Console.WriteLine(ProductName);
    }
    return productNames.Count;
}
```

Una tarea que se ejecuta solamente cuando haya terminado una tarea anterior es llamada ***Continuación***. Este enfoque permite construir una secuencia de operaciones en segundo plano. Las *Tareas de Continuación* (*Continuation Task*) permiten encadenar varias tareas juntas para que se ejecuten una tras otra. La tarea que, al finalizar, invoca a otra tarea, se conoce como el ***Antecedente***. La tarea que es invocada por la tarea *Antecedente* se conoce como la ***Continuación***.

Podemos pasar datos desde el *Antecedente* hacia la *Continuación* y podemos controlar la ejecución de la cadena de tareas. Para crear una *Continuación* básica, utilizamos el método ***Task.ContinueWith***.

4. Agrega el siguiente código en el método *RunContinuationTasks* para definir una segunda tarea que será ejecutada al finalizar la ejecución de la primera tarea.

```
static void RunContinuationTasks()
{
    var FirstTask =
        new Task<List<string>>(GetProductNames);

    var SecondTask = FirstTask.ContinueWith(antecedent =>
    {
        return ProcessData(antecedent.Result);
    });
}
```

Puedes notar que cuando creamos la tarea de *Continuación*, proporcionamos el resultado de la primera tarea como un argumento (*antecedent*) del delegado de la segunda tarea. Utilizamos la primera tarea para recopilar algunos datos y después invocamos la tarea de *Continuación* para procesar los datos. Las *Tareas de Continuación* no necesitan devolver el mismo tipo de resultado que sus tareas antecedentes. En este ejemplo, el resultado de la primera tarea es de tipo ***List<string>*** mientras que el resultado de la segunda tarea es ***int***.

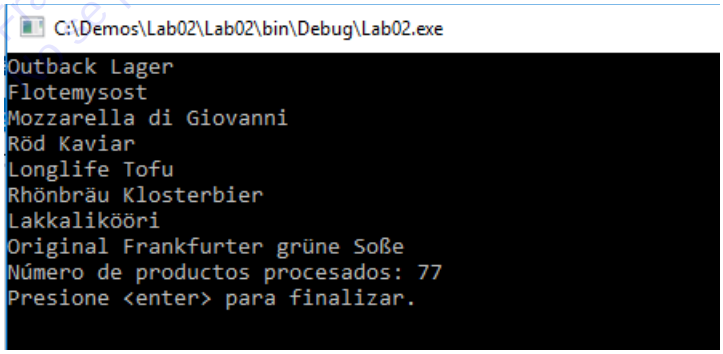
5. Agrega el siguiente código para iniciar la primera tarea y obtener el resultado de la segunda tarea.

```
static void RunContinuationTasks()
{
    var FirstTask =
        new Task<List<string>>(GetProductNames);

    var SecondTask = FirstTask.ContinueWith(antecedent =>
    {
        return ProcessData(antecedent.Result);
    });

    FirstTask.Start();
    Console.WriteLine($"Número de productos procesados: {SecondTask.Result}");
}
```

6. Modifica el código del método ***Main*** para que invoque al método ***RunContinuationTasks***.

```
static void Main(string[] args)
{
    //RunParallelTasks();
    //ParallelLoopIterate();
    //RunLINQ();
    //RunPLINQ();

    RunContinuationTasks();

    Console.Write("Presione <enter> para finalizar.");
    Console.ReadLine();
}
```

7. Ejecuta la aplicación y observa el resultado. Puedes notar que se muestran los nombres de los productos y el total de nombres procesados.



Documento creado por: Franco Morales franco.e.morales@gmail.com
Capacitación para uso personal de: No se permiten copias SIN autorización.

Las ***Continuaciones*** permiten implementar el patrón ***Promise***. Esta es una técnica común que muchos ambientes asíncronos utilizan para garantizar que las operaciones se realizan en una secuencia garantizada.

Para mayor información acerca de las ***Tareas de Continuación***, se recomienda consultar el siguiente enlace.

Chaining Tasks by Using Continuation Tasks
http://go.microsoft.com/fwlink/?LinkID=267841

Tarea 2. Crear Tareas Anidadas.

Una tarea anidada es simplemente una tarea que creamos dentro del delegado de otra tarea. Cuando creamos tareas de esta manera, la tarea anidada y la tarea externa son esencialmente independientes. La tarea externa no necesita esperar a que la tarea anidada finalice para poder finalizar también.

En esta tarea ejemplificaremos el uso de las ***Tareas Anidadas***.

1. Agrega el siguiente código a la clase ***Program*** que ejemplifica la creación y ejecución de una tarea anidada.

```
static void RunNestedTasks()
{
    var OuterTask = Task.Factory.StartNew(()=>
    {
        Console.WriteLine("Iniciando la tarea externa...");
        var InnerTask = Task.Factory.StartNew(() =>
        {
            Console.WriteLine("Iniciando tarea anidada...");
            // Simular un proceso de larga duración
            Thread.Sleep(3000);
            Console.WriteLine("Finalizando la tarea anidada...");
        });
    });
    OuterTask.Wait();
    Console.WriteLine("Tarea externa finalizada");
}
```

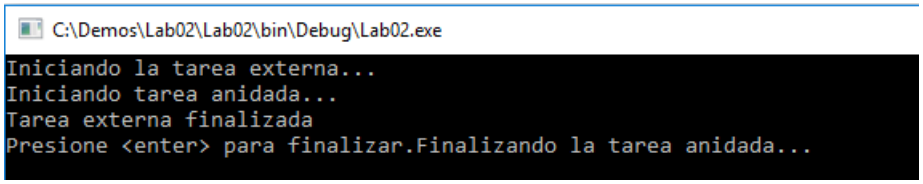
2. Modifica el código del método ***Main*** para ejecutar el método ***RunNestedTasks***.

```
static void Main(string[] args)
{
    //RunParallelTasks();
    //ParallelLoopIterate();
    //RunLINQ();
    //RunPLINQ();
```

Documento creado por: Capacitación para uso personal de: Franco Morales franco.e.morales@gmail.com No se permiten copias SIN autorización

```
        // RunContinuationTasks();
        RunNestedTasks();

        Console.Write("Presione <enter> para finalizar.");
        Console.ReadLine();
    }
```

3. Ejecuta la aplicación y observa el resultado.



Puedes observar que cuando creamos tareas anidadas, la tarea anidada y la tarea externa son esencialmente independientes. La tarea externa no necesita esperar a que la tarea anidada sea completada para poder finalizar. En este ejemplo, debido al retardo en la tarea anidada, la tarea externa es completada antes de la tarea anidada como lo demuestran los mensajes.

Tarea 3. Crear Tareas Hijas.

La tarea principal (la tarea que dio lugar a la nueva tarea o tarea anidada) puede esperar a que las tareas anidadas sean completadas antes de finalizar ella misma o puede retornar y dejar que las tareas anidadas continúen su ejecución de forma asíncrona. Las tareas que causan que la tarea principal espere, son llamadas ***Tareas Hijas*** (***Child Tasks***). Una tarea hija es un tipo de *Tarea Anidada*, excepto que especificamos la opción ***AttachedToParent*** cuando creamos una *Tarea Hija*. En este caso, la *Tarea hija* y la *Tarea Padre* se convierten en tareas estrechamente conectadas. El estatus de la *Tarea Padre* depende del estatus de las *Tareas hijas*. En otras palabras, una *Tarea Principal* no se puede completar hasta que todas sus *Tareas hijas* se hayan completado. La *Tarea Principal* también propaga cualquier excepción que sus *Tareas hijas* lancen.

En esta tarea ejemplificaremos el uso de las ***Tareas Hijas***.

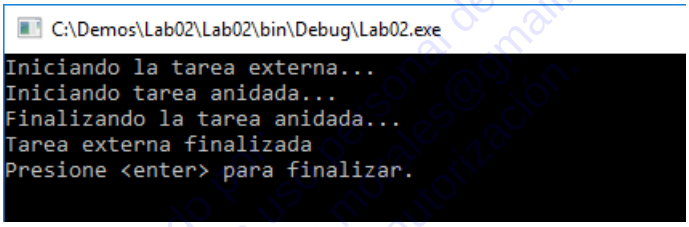
1. Modifica el código del método *RunNestedTasks* para especificar que la tarea anidada es una tarea hija.

```
static void RunNestedTasks()
{
    var OuterTask = Task.Factory.StartNew(()=>
    {
        Console.WriteLine("Iniciando la tarea externa...");
        var InnerTask = Task.Factory.StartNew(() =>
        {
            Console.WriteLine("Iniciando tarea anidada...");
            // Simular un proceso de larga duración
            Thread.Sleep(3000);
```

```
                Console.WriteLine("Finalizando la tarea anidada...");
            }, TaskCreationOptions.AttachedToParent);
        });

        OuterTask.Wait();
        Console.WriteLine("Tarea externa finalizada");
    }
```

Notemos que este ejemplo es esencialmente idéntico al ejemplo de *Tarea Anidada*, excepto que, en este caso, la *Tarea hija* es creada mediante la opción ***AttachedToParent***. Como resultado, en este caso, la *Tarea padre* esperará a que la *Tarea hija* se complete antes de completarse ella misma.

2. Ejecuta la aplicación y observa el resultado. Puedes notar que la tarea principal esperó a que su tarea hija finalizará antes de finalizar ella misma.



Las *Tareas Anidadas* son útiles debido a que nos permiten desglosar operaciones asíncronas en unidades más pequeñas que pueden ser distribuidas a través de los *threads* disponibles. Por el contrario, es más útil utilizar *Tareas padre e hija* cuando necesitemos controlar la sincronización mediante el aseguramiento de que ciertas *Tareas hijas* sean completadas antes de que la *Tarea padre* retorne.

Documento creado para uso personal de:
TI Capacitación
Franco Martínez Flores e.martinez@gmail.com
No se permite copias sin autorización.

Ejercicio 3: Manejo de excepciones en Tareas

Al igual que en otras aplicaciones, cuando desarrollamos aplicaciones con múltiples hilos, podemos encontrar casos donde se pueden originar excepciones. Las excepciones no controladas que son disparadas por el código que es ejecutado dentro de una tarea son propagadas hacia el hilo que inició dicha tarea.

Una tarea que origina una excepción podría estar enlazada a otras tareas a través de *Tareas Hijas* o de *Continuación*, por lo que múltiples excepciones podrían ser lanzadas. Para asegurarnos de que todas las excepciones se propaguen hacia el hilo de inicio, la biblioteca *Task Parallel* empaqueta el conjunto de excepciones en un objeto ***AggregateExeption***.

En este ejercicio exploraremos la forma de manejar excepciones originadas por las tareas.

Tarea 1. Atrapar excepciones de Tareas.

Para capturar las excepciones en el hilo principal, debemos esperar a que la tarea se complete. Hacemos esto invocando al método *Task.Wait* en un bloque *Try* y atrapamos una excepción ***AggregateException*** en el bloque *catch* correspondiente.

1. Agrega el siguiente código dentro de la clase ***Program*** para simular un proceso de larga duración que pueda ser cancelado.

```
static void RunLongTask(CancellationToken token)
{
    for (int i = 0; i < 5; i++)
    {
        // Simular un proceso de larga duración
        Thread.Sleep(2000);
        // Lanzar un OperationCanceledException si se solicita una cancelación
        token.ThrowIfCancellationRequested();
    }
}
```

2. Agrega el siguiente código dentro de la clase ***Program*** para definir un método que nos permitirá manejar excepciones de tareas.

```
static void HandleTaskExceptions()
{
}
```

Documento creado por: Capacitación para uso personal de Franco Morales franco.morales@gmail.com No se permite copia sin autorización.

3. Agrega el siguiente código dentro del método *HandleTaskExceptions* para obtener un token que nos permita cancelar una tarea de larga duración.

```
// Obtener un Token de cancelación
var CTS = new CancellationTokenSource();
var CT = CTS.Token;
```

Cuando una tarea genera una excepción, la excepción se propaga hacia el hilo que inició la tarea. El hilo que inicia la tarea que causa la excepción es conocido como ***Joining Thread***. Para nuestro ejemplo, el hilo principal de la aplicación lanzará una tarea que disparará una excepción, en consecuencia, el hilo principal de la aplicación será el ***Joining Thread***.

4. Agrega el siguiente código dentro del método ***HandleTaskExceptions*** para crear e iniciar la tarea de larga duración.

```
static void HandleTaskExceptions()
{
    // Obtener un Token de cancelación
    var CTS = new CancellationTokenSource();
    var CT = CTS.Token;

    var LongRunningTask =
        Task.Run(() => RunLongTask(CT), CT);
}
```

La tarea podría estar enlazada a otras tareas a través de *Tareas hijas* o de *Continuación*, por lo que múltiples excepciones podrían ser lanzadas.

Para capturar las excepciones en el *Joining Thread*, debemos esperar a que la tarea finalice. Hacemos esto, invocando al método ***Task.Wait*** en un bloque **Try**.

5. Agrega el siguiente código dentro del método ***HandleTaskExceptions*** para esperar a que la tarea finalice su ejecución.

```
try
{
    LongRunningTask.Wait();
}
```

Para asegurarnos de que todas las excepciones se propaguen hacia el hilo de inicio (*Joining Thread*), la biblioteca *Task Parallel* empaqueta el conjunto de excepciones en un objeto ***AggregateExeption***, así que debemos atrapar una excepción ***AggregateException*** en un bloque ***catch***.

6. Agrega el siguiente código dentro del método ***HandleTaskExceptions*** para atrapar la excepción ***AggregateException*** y la excepción general ***System.Exception***.

Documento creado por: TI Capacitación para uso personal de: Francisco Morales francisco.e.morales@gmail.com No se permiten copias SIN autorización.

```
static void HandleTaskExceptions()
{
    // Obtener un Token de cancelación
    var CTS = new CancellationTokenSource();
    var CT = CTS.Token;

    var LongRunningTask =
        Task.Run(() => RunLongTask(CT), CT);

    try
    {
        LongRunningTask.Wait();
    }
    catch(AggregateException ae)
    {

    }
    catch(Exception ex)
    {
        Console.WriteLine($"Excepción: {ex.Message}");
    }
}
```

A través de una colección ***InnerExceptions***, el objeto ***AggregateException*** expone todas las excepciones que se han generado.

Un escenario común de manejo de excepciones es atrapar la excepción ***TaskCanceledException*** que es disparada cuando cancelamos una tarea.

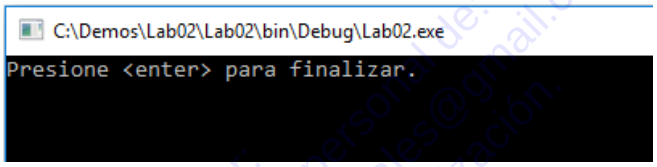
7. Agrega el siguiente código dentro del bloque ***catch(AggregateException ae)*** para mostrar los mensajes de las excepciones que se hayan generado.

```
catch(AggregateException ae)
{
    foreach(var Inner in ae.InnerExceptions)
    {
        if(Inner is TaskCanceledException)
        {
            Console.WriteLine("La tarea fue cancelada.");
        }
        else
        {
            // Aquí procesamos las excepciones distintas a la cancelación.
            Console.WriteLine(Inner.Message);
        }
    }
}
```

8. Modifica el código del método ***Main*** para ejecutar al método ***HandleTaskExceptions***.

```
static void Main(string[] args)
{
    //RunParallelTasks();
    //ParallelLoopIterate();
    //RunLINQ();
    //RunPLINQ();
    // RunContinuationTasks();
    // RunNestedTasks();
    HandleTaskExceptions();

    Console.Write("Presione <enter> para finalizar.");
    Console.ReadLine();
}
```

9. Ejecuta la aplicación y examina el resultado. Puedes notar que después de unos 10 segundos, la aplicación finaliza correctamente, sin ninguna excepción.

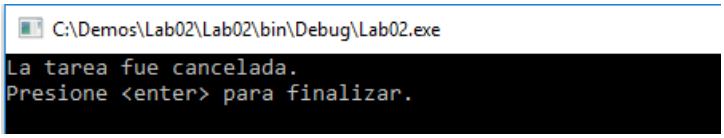


10. Presiona la tecla ***Enter*** para finalizar la ejecución y regresar a Visual Studio.

11. Agrega la siguiente instrucción dentro del método ***HandleTaskExceptions*** para cancelar la tarea.

```
var LongRunningTask =
    Task.Run(() => RunLongTask(CT), CT);

CTS.Cancel();

try
{
    LongRunningTask.Wait();
}
```

12. Ejecuta la aplicación y examina el resultado. Puedes notar que se genera una ***AggregateException*** y que se muestra el mensaje indicando que la tarea fue cancelada.



Documento creado por TI Capacitación para uso personal de: Franco Nicolás Moraes e.moraes@gmail.com No se permiten copias sin autorización.

Resumen

En este laboratorio aprendiste a utilizar la biblioteca de clases *Task Parallel* para ejecutar tareas en paralelo, enlazar tareas y manejar las excepciones que pueden ser generadas durante la ejecución de tareas.

Documento creado por:
TI Capacitacion para uso personal de:
Franco Morales franco.e.morales@gmail.com
No se permiten copias SIN autorización.