Seminario de Lenguajes (.NET)

Práctica 10

Ejercicios obligatorios para el coloquio: Los **ejercicios 8, 9, 13, 14** y **15** de esta práctica forman parte del conjunto de ejercicios de programación obligatorios que el alumno debe resolver y exponer de manera oral sobre máquina el día del coloquio hacia final de la cursada.

1) Dado el método **`Saludar`**

```
static void Saludar()
{
    Console.WriteLine("Hola Mundo!");
}
```

a) Instanciar un objeto **`Task`** utilizando su constructor **`Task(Action)`** para ejecutar asincrónicamente el método **`Saludar`**. Iniciar la tarea y esperar a que se complete para ver el resultado en la consola antes de que finalice el programa.

b) Ídem al inciso a) pero en lugar de utilizar un constructor de **`Task`** utilizar el método estático **`Run`** de esta clase.

c) Ídem al inciso b) pero reemplazando el método **`Saludar`** por una expresión lambda que haga lo mismo. Intentar resolver todo en una única línea de código, aún la espera por la finalización de la tarea.

d) Ídem al inciso c) pero reemplazando la expresión lambda por un método anónimo.

2) Dado el método **`Imprimir`**

```
static void Imprimir()
{
    int idTrhead = System.Threading.Thread.CurrentThread.ManagedThreadId;
    int? idTarea = Task.CurrentId;
    Console.WriteLine($"Tarea: {idTarea}  Thread: {id Thread}");
}
```

Ejecutarlo asincrónicamente 100 veces de manera concurrente. Para ello, instanciar 100 objetos **`Task`** en un vector. Se debe esperar a la finalización de todas las tareas utilizando el método estático **`WaitAll`** de la clase **`Task`**. Observar que, por cuestiones de rendimiento, se van a generar muchos menos threads que tareas.

3) Dado el siguiente método **`Procesar`** que simula algún trabajo útil que consume ciclos de CPU

```
static void Procesar()
{
    for (int i = 0; i < 1000; i++)
    {
        string st = i.ToString();
    }
    Console.WriteLine("Fin del procesamiento");
}
```

- Crear 4 tareas para ejecutar este método concurrentemente de manera asincrónica.
- Tomar el tiempo de cuánto dura esta ejecución hasta que las 4 tareas se completen.
- Compararlo con el tiempo requerido para ejecutar este método 4 veces seguidas de manera secuencial. Es de esperar que la sobrecarga necesaria para la administración de los subprocesos sea mayor a la ganancia obtenida por la ejecución simultánea en una arquitectura multicore. Por lo tanto seguramente en este caso la ejecución secuencial será mucho más rápida
- Probar con distintos valores para la cantidad de ciclos del **`for`** en el método **`Procesar`**, dependiendo de la arquitectura de la máquina donde esté corriendo, para una cantidad grande (quizá cercana a los 100 millones) la solución asincrónica será más eficiente. Observar el uso de la CPU (con alguna herramienta del SO sobre el que se esté ejecutando) mientras corre el programa.

4) Dado el siguiente método **`Imprimir`** utilizar el constructor **`Task(Action<Object>, Object)`** para correr asincrónicamente 1000 tareas con el objetivo de imprimir los números del 1 al 1000 en la consola (la impresión no será ordenada).

```
static void Imprimir(object o)
{
    Console.Write($"{o} - ");
}
```

5) No se puede usar el método estático **`Task.Run`** para crear e iniciar una tarea con un delegado genérico **`Action<object>`**. Sin embargo puede usarse el método **`StartNew`** de un objeto de tipo **`TaskFactory`**. Modificar el ejercicio 4 utilizando la propiedad **`Task.Factory`** (que devuelve un objeto de tipo **`TaskFactory`**) y el método **`StartNew`** para crear e iniciar las tareas en una sóla operación.

6) Reemplazar el método **`Imprimir`** en el ejercicio anterior por una expresión lambda equivalente.

7) Codificar el método **`static void Sumatoria(int n)`** que calcula la sumatoria de **`1`** hasta **`n`** y que imprime el resultado en la consola, completar el siguiente código para invocar asincrónicamente el método **`Sumatoria`** con argumentos que van desde **`1`** hasta **`10`**. Se debe utilizar una expresión lambda para la definición de las tareas.

```
List<Task> tareas = new List<Task>();
for (int n = 1; n <= 10; n++)
{
   . . .
}
Task.WaitAll(tareas.ToArray());
```

Posible salida por consola
El orden es no determinístico

```
suma desde 1 hasta 4 = 10
suma desde 1 hasta 2 = 3
suma desde 1 hasta 1 = 1
suma desde 1 hasta 3 = 6
suma desde 1 hasta 7 = 28
suma desde 1 hasta 5 = 15
suma desde 1 hasta 6 = 21
suma desde 1 hasta 8 = 36
suma desde 1 hasta 9 = 45
suma desde 1 hasta 10 = 55
```

8) Codificar el método **`static void Sumatoria(int a, int b)`** que calcula la sumatoria desde **`a`** hasta **`b`** y que imprime el resultado en la consola. Completar el siguiente código para invocar asincrónicamente el método **`Sumatoria`** para los distintos valores de **`a`** y de **`b`** que se especifican en las instrucciones **`for`**. Se debe utilizar una expresión lambda para la definición de las tareas.

```
List<Task> tareas = new List<Task>();
for (int a = 1; a <= 3; a++)
{
    for (int b = a + 2; b <= a + 4; b++)
    {
        . . .
    }
}
Task.WaitAll(tareas.ToArray());
```

Posible salida por consola
El orden es no determinístico

```
suma desde 1 hasta 5 = 15
suma desde 1 hasta 3 = 6
suma desde 1 hasta 4 = 10
suma desde 2 hasta 4 = 9
suma desde 3 hasta 5 = 12
suma desde 3 hasta 6 = 18
suma desde 2 hasta 5 = 14
suma desde 2 hasta 6 = 20
suma desde 3 hasta 7 = 25
```

9) Dado el método **`static int Sumatoria(int n)`** definido de la siguiente manera:

```
static int Sumatoria(int n)
{
    int suma = 0;
    for (int i = 1; i <= n; i++)
    {
        suma += i;
    }
    return suma;
}
```

Completar el siguiente código para invocar asincrónicamente el método **`Sumatoria`** e imprimir el entero que devuelve cada tarea

```
List<Task<int>> tareas = new List<Task<int>>();
for (int n = 1; n <= 10; n++)
{
     . . .
}
. . . // Imprimir el resultado devuelto por  cada tarea
```

Salida por consola

a) Resolverlo utilizando un constructor de la clase **`Task`**
b) Resolverlo utilizando el método **`StartNew`** de una instancia de **`TaskFactory`**

10) Se debe tener cuidado con el acceso a variables compartidas por distintas tareas. Observar el comportamiento del siguiente programa

```
class Program
{
    static string leyenda = "Valores procesados: ";
    public static void Main()
    {
        List<Task> tareas = new List<Task>();
        for (int n = 1; n <= 10; n++)
        {
            Task t = new Task((o) => Procesar(o), n);
            tareas.Add(t);
            t.Start();
        }
        Task.WaitAll(tareas.ToArray());
        Console.WriteLine(leyenda);
    }
    static void Procesar(object obj)
    {
        // hace algún trabajo y accede a una variable compartida;
        leyenda += obj + " ";
    }
}
```

El programador pretenden que, a medida que las tareas se vayan completando, se vaya concatenando el valor procesado (recibido como parámetro) en en la variable **`leyenda`**.
a) Ejecutar varias veces el programa y observar la salida en la consola.
b) Investigar en la documentación de .Net la utilidad de la instrucción **`lock`**. Corregir el código para que el programa haga lo que se pretende.

11) Tomando como base el ejercicio 8), transformar el método **`static void Sumatoria(int a, int b)`** en un método asincrónico (usando **`async/await`**) que pueda ser utilizado de la siguiente manera:

```
List<Task> tareas = new List<Task>();
for (int a = 1; a <= 3; a++)
{
    for (int b = a + 2; b <= a + 4; b++)
    {
        tareas.Add(SumatoriaAsync(a, b));
    }
}
Task.WaitAll(tareas.ToArray());
```

Posible salida por consola
El orden es no determinístico

```
suma desde 1 hasta 5 = 15
suma desde 1 hasta 3 = 6
suma desde 1 hasta 4 = 10
suma desde 2 hasta 4 = 9
suma desde 2 hasta 6 = 20
suma desde 3 hasta 7 = 25
suma desde 2 hasta 5 = 14
suma desde 3 hasta 6 = 18
suma desde 3 hasta 5 = 12
```

12) Tomando como base el ejercicio 9), transformar el método `static int Sumatoria(int n)` en un método asincrónico (usando `async/await`)

13) Codificar un método asincrónico que devuelva un `Task<string>` con el contenido de un archivo de texto cuyo nombre se pasa como parámetro.

14) Codificar un método asincrónico que utilice el método codificado en el ejercicio anterior y que devuelva un `Task<int>` con la cantidad de palabras contenidas en un archivo de texto cuyo nombre se pasa como parámetro.

15) Codificar un método asincrónico que utilice el método codificado en el ejercicio anterior y que devuelva un `Task<int[]>` con la cantidad de palabras contenidas en cada uno de los archivos de texto cuyos nombres se pasan como parámetro en un `string[]`. Este método debe invocar varias veces al método definido en el ejercicio anterior lo que generará varias tareas que deben esperarse asincrónicamente. Para esperar varias tareas de manera asincrónica se puede usar `Task.WenAll(...)` que crea una tarea que finalizará cuando se hayan completado todas las tareas proporcionadas, por lo tanto se puede usar `await Task.WenAll(...)`.