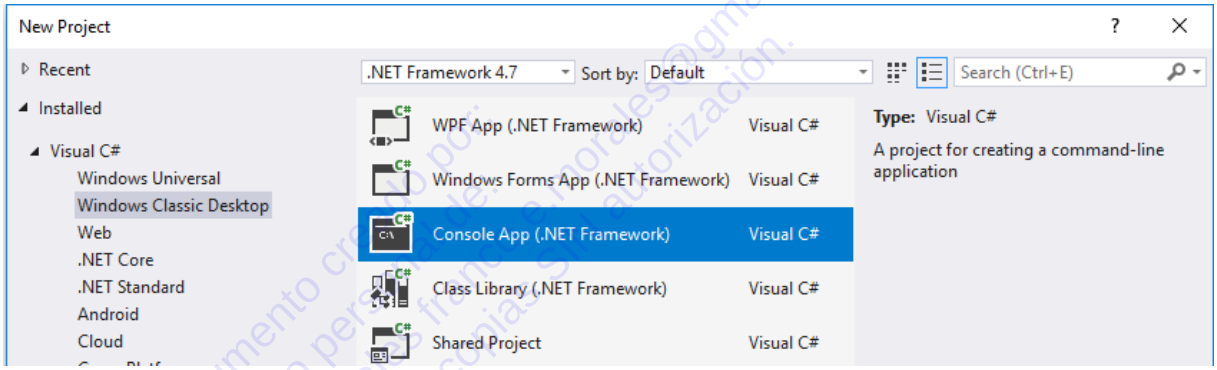
Primitivas de Sincronización Comunes: SemaphoreSlim

Otra de las primitivas de sincronización comunes que podemos utilizar con la biblioteca *Task Parallel* es a través de la clase ***SemaphoreSlim***. La clase *SemaphoreSlim* nos permite restringir el número de hilos que pueden acceder a un recurso o grupo de recursos de forma concurrente, esto es, al mismo tiempo.

Ejercicio
Utilizando SemaphoreSlim con Task Parallel Library

En este ejercicio conocerás el uso de la primitiva de sincronización *SemaphoreSlim* soportada por la biblioteca ***Task Parallel***.

1. Crea una aplicación de ***Consola*** utilizando la plantilla ***Console App (.NET Framework)***.



2. Agrega la siguiente instrucción al inicio del archivo ***Program.cs*** para importar el espacio de nombre de la clase ***Thread***.

```
using System.Threading;
```

3. Agrega el siguiente código dentro de la clase ***Program*** para definir un método que nos permita ejemplificar el uso de la primitiva ***SemaphoreSlim***.

```
static void UseSemaphoreSlim()
{
}
```

La clase *SemaphoreSlim* utiliza un contador entero para registrar el número de hilos que están accediendo actualmente a un recurso o a un grupo de recursos. Cuando creamos un objeto *SemaphoreSlim*, especificamos un valor inicial y opcionalmente el valor máximo de accesos concurrentes que serán permitidos.

4. Agrega el siguiente código dentro del método ***UseSemaphoreSlim*** para crear una instancia de *SemaphoreSlim* que permita dos accesos concurrentes.

```
SemaphoreSlim S = new SemaphoreSlim(2);
```

Cuando un hilo desea acceder a los recursos protegidos, este invoca al método ***Wait*** del objeto *SemaphoreSlim*. El método *Wait* bloquea el hilo hasta que pueda tener acceso.

5. Agrega el siguiente código dentro del método ***UseSemaphoreSlim*** para solicitar la continuación de ejecución esperando un máximo de 10,000 milisegundos.

```
bool Entered = S.Wait(10000);
```

El método *Wait* devolverá ***true*** si pasó exitosamente el semáforo o ***false*** en caso contrario, esto es, transcurrió el tiempo máximo de espera.

6. Agrega el siguiente código dentro del método ***UseSemaphoreSlim*** para mostrar el valor devuelto.

```
Console.WriteLine($"El hilo no tuvo que esperar 10000 milisegundos {Entered}");
```

Si el valor actual de accesos concurrentes permitidos del objeto *SemaphoreSlim* es mayor que cero, el contador es decrementado y al hilo le es otorgado el acceso para continuar su ejecución. En este ejemplo, debido a que el valor inicial del contador del objeto *SemaphoreSlim* es 2, el hilo no tendrá que esperar y continuará sin problema. El valor que devolverá el método *Wait* en este punto es ***true***.

7. Agrega el siguiente código dentro del método ***UseSemaphoreSlim*** para solicitar la continuación de ejecución esperando un máximo de 10,000 milisegundos.

```
Entered = S.Wait(10000);
Console.WriteLine($"Segundo acceso exitoso: {Entered}");
```

En este punto, después de 2 accesos exitosos, el contador de accesos permitidos por *SemaphoreSlim* llegó cero. Si un hilo invoca al método *Wait* y el contador es cero, el hilo es bloqueado.

8. Agrega el siguiente código dentro del método ***UseSemaphoreSlim*** para solicitar la continuación de ejecución esperando un máximo de 10,000 milisegundos.

```
Console.WriteLine("Aquí ya no hay accesos y el hilo tendrá que esperar...");
Entered = S.Wait(10000);
Console.WriteLine($"Tercer acceso exitoso: {Entered}");
```

En este ejemplo, el valor que será mostrado es **false** ya que habrá transcurrido el tiempo máximo de espera (10,000 milisegundos) sin haber conseguido el permiso de *SemaphoreSlim.*

Cuando el hilo haya finalizado su ejecución, debe invocar al método ***Release*** del objeto *SemaphoreSlim*. Con esto, el contador es incrementado indicando que hay un acceso disponible.

9. Agrega el siguiente código dentro del método ***UseSemaphoreSlim*** para indicar que el hilo finalizó la ejecución.

```
S.Release();
```

Si un hilo invoca al método *Wait* y el contador es cero, el hilo es bloqueado hasta que otro hilo invoque al método *Release* o hasta que transcurra el tiempo máximo de espera.

10. Agrega el siguiente código dentro del método ***UseSemaphoreSlim*** para comprobar que después de ejecutar el método *Release*, el acceso es permitido nuevamente.

```
Entered = S.Wait(10000);
Console.WriteLine($"Después del Release el acceso es exitoso: {Entered}");
```

11. Agrega el siguiente código dentro del método ***UseSemaphoreSlim*** para lanzar una tarea asíncrona que libere el semáforo después de 100 milisegundos.

```
var T = Task.Run(() =>
{
    Thread.Sleep(100);
    Console.WriteLine("Tarea liberando el semáforo...");
    S.Release();
});
```

12. Agrega el siguiente código dentro del método ***UseSemaphoreSlim*** para mostrar el número actual de accesos concurrentes permitidos.

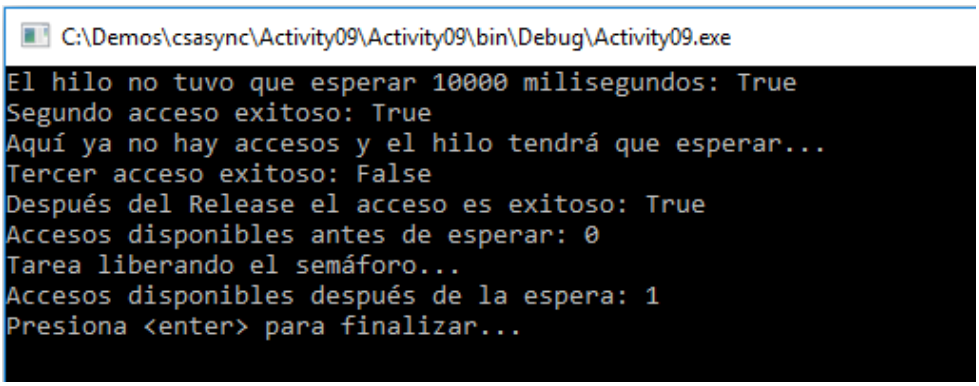
```
Console.WriteLine($"Accesos disponibles antes de esperar: {S.CurrentCount}");
T.Wait();
Console.WriteLine($"Disponibles después de la espera: {S.CurrentCount}");
S.Dispose();
```

13. Agrega el siguiente código dentro del método ***Main*** para invocar al método ***UseSemaphoreSlim***.

```
static void Main(string[] args)
{
    UseSemaphoreSlim();
    Console.Write("Presiona <enter> para finalizar...");
    Console.ReadLine();
}
```

14. Ejecuta la aplicación. Puedes notar el resultado con la secuencia de mensajes esperada.

Documento cedido por: Franco Mendes franco.e.morales@gmail.com para uso personal de: No se permite copias SIN autorización.



En este ejercicio mostramos el uso de la primitiva de sincronización ***SemaphoreSlim*** soportada por la biblioteca ***Task Parallel.***

Para obtener más información sobre la clase ***SemaphoreSlim***, se recomienda consultar el siguiente enlace:

SemaphoreSlim Class
http://go.microsoft.com/fwlink/?LinkID=267851

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