



Primitivas de Sincronización Comunes: Barrier

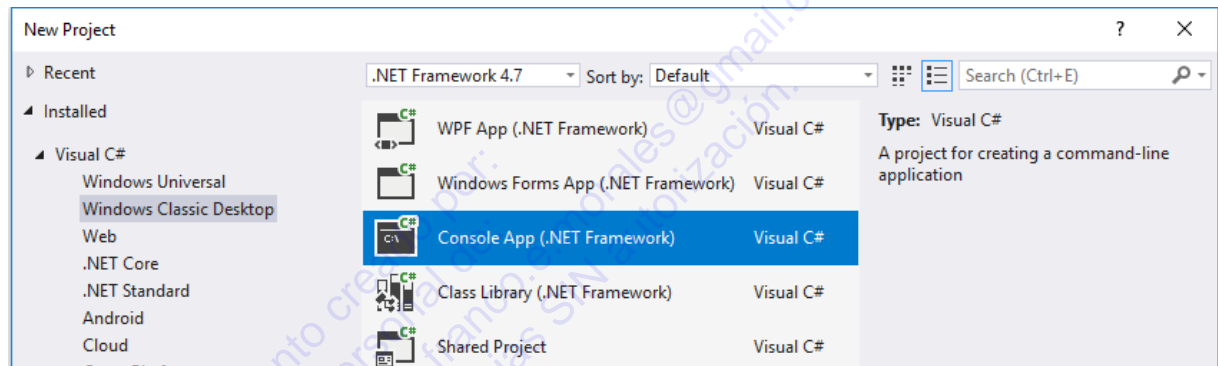
Otra de las primitivas de sincronización comunes que podemos utilizar con la biblioteca *Task Parallel* es a través de la clase **Barrier**. La clase *Barrier* permite detener temporalmente la ejecución de varios hilos hasta que todos ellos hayan llegado a un punto en particular.

Ejercicio

Utilizando Barrier con Task Parallel Library

En este ejercicio conocerás el uso de la primitiva de sincronización *Barrier* 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 **Barrier**.

```
static void UseBarrier()  
{  
}
```

Supongamos que tenemos que realizar un proceso de instalación de un sistema que se divide en cuatro fases:

- Fase 1: Copiar archivos comprimidos desde el dispositivo origen.
- Fase 2: Extraer archivos de instalación.
- Fase 3: Instalar los módulos del Sistema.
- Fase 4: Configurar los módulos instalados



Cada fase debe completarse antes de pasar a la siguiente fase. No podemos empezar a extraer archivos si aún no se han copiado todos los archivos fuente comprimidos, tampoco podemos empezar a instalar los módulos si estos aún no han sido extraídos y no podemos configurarlos si aún no han sido instalados. Debido a que el proceso a realizar es de larga duración, deseamos agilizar el proceso lanzando 4 hilos de forma simultánea para que realicen todo el proceso.

4. Agrega el siguiente código dentro del método **UseBarrier** para simular la ejecución del proceso de instalación del sistema utilizando 4 hilos en paralelo.

```
static void UseBarrier()
{
    Action Install = () =>
    {
        // Fase 1

        // Fase 2

        // Fase 3

        // Fase 4
    };
    Parallel.Invoke(Install, Install, Install, Install);
}
```

El problema que tenemos ahora es que debemos sincronizar cada hilo antes de iniciar una fase. La clase **Barrier** permite detener temporalmente la ejecución de varios hilos hasta que todos ellos hayan llegado a un punto en particular, en nuestro caso, en el punto en que hayan terminado una fase.

Cuando creamos un objeto **Barrier**, especificamos un número inicial de hilos participantes, en nuestro caso, queremos 4 hilos participando.

5. Agrega el siguiente código al inicio del método **UseBarrier** para crear un objeto **Barrier** indicando que hay 4 hilos participantes.

```
Barrier B = new Barrier(4);
```

El objeto **Barrier** cuenta con los métodos **AddParticipant** y **RemoveParticipant** para agregar o eliminar un hilo participante. También cuenta con los métodos **AddParticipants** y **RemoveParticipants** para agregar o eliminar un grupo de hilos participantes.

Cuando un hilo alcance el punto de sincronización deseado, por ejemplo, al finalizar una fase, el hilo debe invocar al método **SignalAndWait** del objeto **Barrier**.

6. Agrega el siguiente código para indicar que el hilo actual ha alcanzado el fin de la fase 1.



```
// Fase 1  
B.SignalAndWait();
```

SignalAndWait decrementa el contador *Barrier* y también bloquea el hilo actual hasta que el contador llegue a cero. Cuando el contador llega a cero, a todos los hilos se les permite continuar. La clase *Barrier* es utilizada frecuentemente en escenarios donde varias tareas realizan cálculos interrelacionados en un periodo de tiempo y después esperan a que todas las otras tareas alcancen el mismo punto antes de realizar cálculos interrelacionados en el siguiente periodo de tiempo.

7. Modifica el método **UseBarrier** para indicar los puntos en que es finalizada cada fase del proceso.

```
static void UseBarrier()  
{  
    Barrier B = new Barrier(4);  
  
    Action Install = () =>  
    {  
        // Fase 1  
        B.SignalAndWait();  
  
        // Fase 2  
        B.SignalAndWait();  
        // Fase 3  
        B.SignalAndWait();  
        // Fase 4  
        B.SignalAndWait();  
    };  
    Parallel.Invoke(Install, Install, Install, Install);  
}
```

Una fase se considera terminada cuando todos los hilos participantes hayan reportado que han terminado la fase invocando el método **SignalAndWait**.

8. Agrega el siguiente código al inicio del método **UseBarrier** para poder contar el número de veces que un hilo invoca al método **SignalAndWait**.

```
int Counter = 0;
```

9. Agrega el siguiente código para incrementar el contador al finalizar cada fase. La clase **Interlocked**, proporciona operaciones atómicas para variables que son compartidas por múltiples hilos.

```
// Fase 1  
Interlocked.Increment(ref Counter);  
B.SignalAndWait();  
// Fase 2  
Interlocked.Increment(ref Counter);  
B.SignalAndWait();
```



```
// Fase 3
Interlocked.Increment(ref Counter);
B.SignalAndWait();
// Fase 4
Interlocked.Increment(ref Counter);
B.SignalAndWait();
```

También es posible indicar al objeto *Barrier* que ejecute una acción cada vez que una fase haya sido finalizada.

10. Modifica la declaración del objeto *Barrier* para que envíe a la consola un mensaje indicando el momento en que cada fase es finalizada.

```
Barrier B = new Barrier(4, (BarrierObject) =>
{
    string Message =
        $"Fase {BarrierObject.CurrentPhaseNumber} finalizada. " +
        $"{Counter} notificaciones";
    Console.WriteLine(Message);
});
```

Es una buena práctica invocar al método **Dispose** cuando hayamos terminado de utilizar el objeto *Barrier*.

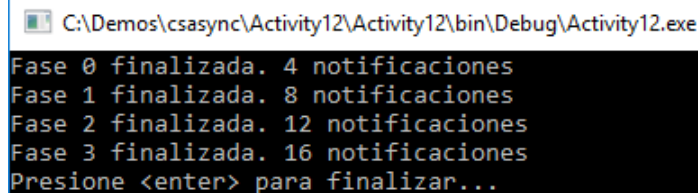
11. Agrega el siguiente código al final del método **UseBarrier** para invocar al método **Dispose** del objeto *Barrier*.

```
B.Dispose();
```

12. Agrega el siguiente código dentro del método **Main** para invocar al método **UseBarrier**.

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

13. Ejecuta la aplicación. Puedes notar que por cada fase finalizada se reciben 4 notificaciones tal y como era esperado, una notificación por cada uno de los 4 hilos, 16 notificaciones en total por las 4 fases.



```
C:\Demos\csasync\Activity12\Activity12\bin\Debug\Activity12.exe
Fase 0 finalizada. 4 notificaciones
Fase 1 finalizada. 8 notificaciones
Fase 2 finalizada. 12 notificaciones
Fase 3 finalizada. 16 notificaciones
Presione <enter> para finalizar...
```



Muchas de las clases de sincronización, permiten establecer tiempos de espera (timeouts) en términos del número de giros (*Spins*). Cuando un hilo está esperando un evento, se dice que está girando (*Spin*). La cantidad de tiempo que toma un *Spin* depende de la computadora que está ejecutando el hilo. Por ejemplo, si utilizamos la clase *ManualResetEventSlim*, podemos especificar el número máximo de *Spins* como un argumento del constructor. Si un hilo está esperando al objeto *ManualResetEventSlim* para señalizar y este alcanza el número máximo de *Spins*, el hilo es suspendido y deja de utilizar los recursos del procesador. Esto ayuda a asegurar que las tareas en espera no consuman tiempo excesivo de procesador.

En este ejercicio mostramos el uso de la primitiva de sincronización **Barrier** que junto con las clases **ManualResetEventSlim**, **SemaphoreSlim**, **CountdownEvent** y **ReaderWriterLockSlim** son las primitivas de sincronización de hilos más comunes soportadas por la biblioteca *Task Parallel*.



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

Barrier Class

<http://go.microsoft.com/fwlink/?LinkID=267854>