



Utilizando Bloqueos

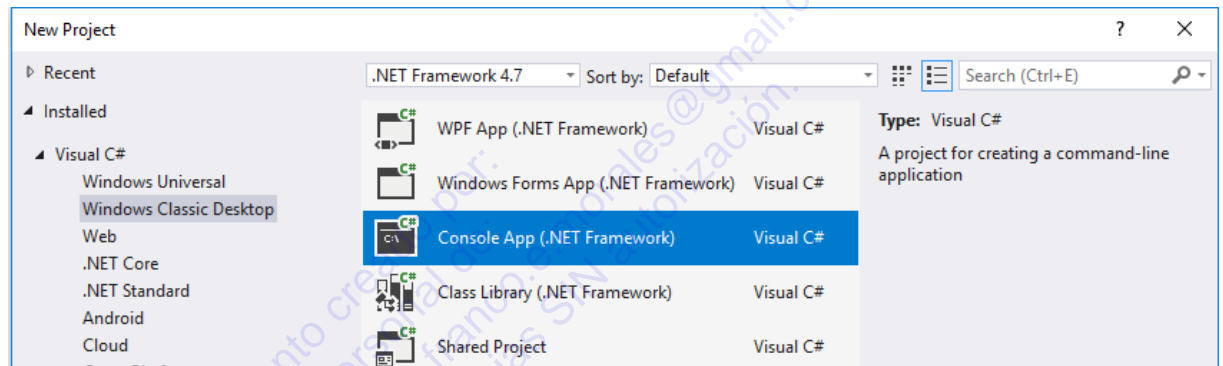
Al introducir multiprocesamiento en nuestras aplicaciones, podemos encontrar a menudo situaciones en las que un recurso es accedido simultáneamente desde múltiples hilos. Si cada uno de esos hilos puede modificar el recurso, este puede terminar en un estado impredecible.

Ejercicio

Utilizando Bloqueos

En este ejercicio conocerás la forma de aplicar un bloqueo de exclusión mutua (mutual-exclusión) a secciones críticas de código.

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



2. Agrega al proyecto un nuevo archivo de clase llamado **Product**. La clase **Product** nos permitirá simular la administración de la existencia de un producto de un almacén ficticio.
3. Modifica el código de la clase **Product** para definir una variable entera que almacene la existencia actual del producto. La clase **Product** también tendrá un constructor que reciba la existencia inicial del producto en el almacén.

```
class Product
{
    int UnitsInStock;
    public Product(int initialUnitsInStock)
    {
        UnitsInStock = initialUnitsInStock;
    }
}
```

Cuando se realice un pedido de una cantidad de un producto, un método se encargará de efectuar el proceso de atención del pedido.

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



```
using System.Threading;
```

5. Agrega el siguiente código a la clase **Product** para definir un método que devuelva **true** en caso de que haya existencia suficiente para atender el pedido y que devuelva **false** en caso contrario.

```
public bool PlaceOrder(int requestedUnits)
{
    bool Accepted = false;

    // UnitsInStock jamás debería ser una cantidad negativa
    if(UnitsInStock < 0)
    {
        throw new Exception("La existencia no puede ser negativa.");
    }

    if(UnitsInStock >= requestedUnits)
    {
        // Simulamos un proceso de larga duración
        Thread.Sleep(1000);
        Console.WriteLine($"Existencia antes del pedido: {UnitsInStock}");
        // Restamos a la existencia la cantidad solicitada.
        UnitsInStock -= requestedUnits;
        Console.WriteLine($"Existencia después del pedido: {UnitsInStock}");
        Accepted = true;
    }
    else
    {
        Console.WriteLine(
            $"Existencia insuficiente: {requestedUnits} de {UnitsInStock}");
    }
    return Accepted;
}
```

Si para cualquier momento dado, un solo hilo puede invocar a este método, todo saldría bien. Sin embargo, supongamos que dos hilos invocan este método al mismo tiempo, la existencia actual del producto podría cambiar entre los pasos de comprobación de existencia y aprobación del pedido o bien entre la aprobación del pedido y la disminución de la existencia, haciendo imposible mantener el registro de la existencia actual del producto por lo que no sabríamos si podemos vender o no el producto deseado.

6. Agrega el siguiente código dentro del método **Main** del archivo **Program.cs** para simular la realización de pedidos.

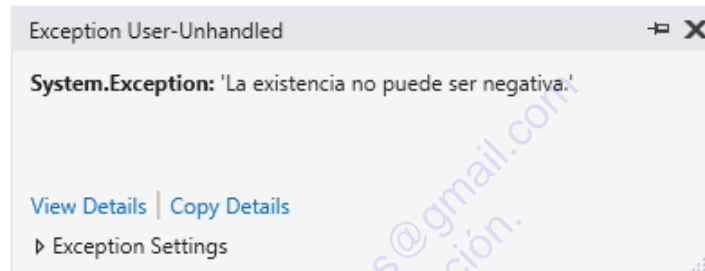
```
static void Main(string[] args)
{
    Product P = new Product(1000);
    Random R = new Random();
    // Simulamos 100 operaciones en paralelo.
    Parallel.For(0, 100, index =>
    {
```



```
        // Intentamos realizar un pedido de 1 a 100 unidades  
        P.PlaceOrder(R.Next(1, 100));  
    });  
    Console.WriteLine("Presione <enter> para finalizar la aplicación...");  
    Console.ReadLine();  
}
```

7. Ejecuta la aplicación. Puedes notar que se genera una excepción que normalmente no debería ocurrir ya que la lógica no disminuye la cantidad existente si no hay existencia suficiente para realizar el pedido.

```
throw new Exception("La existencia no puede ser negativa.");
```



La excepción se genera debido al acceso concurrente a la sección de código crítica del método **PlaceOrder**.

Para resolver este problema, podemos utilizar la palabra clave **lock** para aplicar un bloqueo de **exclusión mutua (mutual-exclusion)** a secciones críticas de código, en nuestro caso, dentro del método **PlaceOrder**. Un bloqueo de **exclusión mutua** hace que cuando un hilo esté accediendo a la sección crítica, todos los demás hilos sean bloqueados.

Para aplicar un bloqueo de **exclusión mutua**, utilizamos la palabra clave **lock**. El bloqueo se aplica a un objeto debido a que el bloqueo trabaja asegurado que un objeto solo pueda ser accedido por un único hilo en cualquier momento dado. El objeto debería ser privado y no debería servir para otro fin en nuestra lógica, sólo para ser bloqueado.

8. Agrega el siguiente código a nivel de la clase **Product**, por ejemplo, antes de la declaración del método **PlaceOrder**.

```
private object ObjectToLock = new object();
```

El propósito de este objeto es proporcionar *algo* que la instrucción **lock** pueda hacer mutuamente exclusivo, bloqueándolo.

La sección crítica de código debe ir dentro del bloque **lock**. Mientras la instrucción **lock** esté en el ámbito de ejecución, sólo un hilo podrá entrar a la sección crítica en un momento dado.

9. Modifica el código del método **PlaceOrder** para proteger la sección crítica del código.

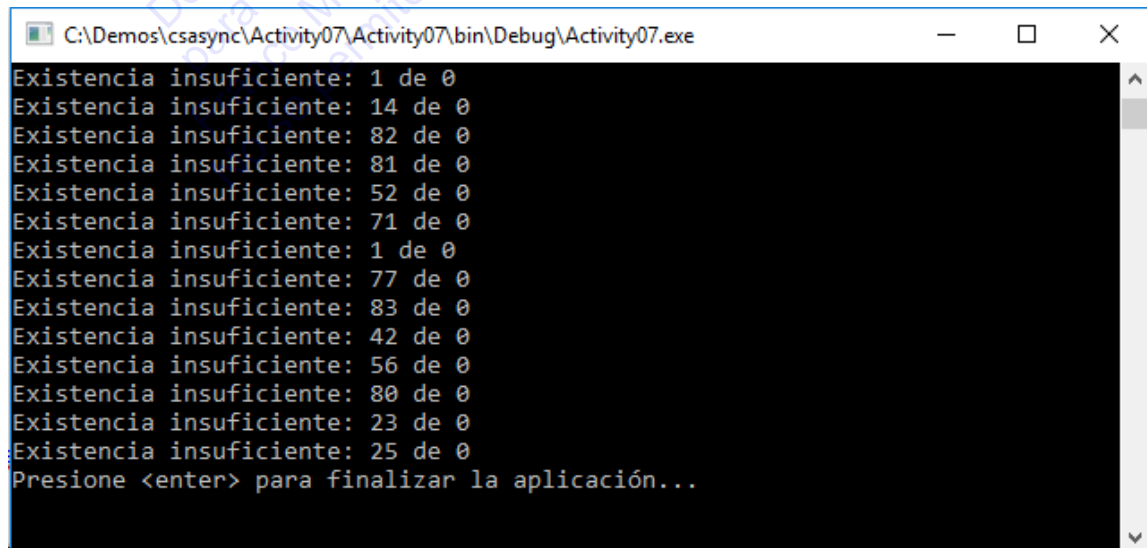


```
private object ObjectToLock = new object();
public bool PlaceOrder(int requestedUnits)
{
    bool Accepted = false;

    // UnitsInStock jamás debería ser una cantidad negativa
    if (UnitsInStock < 0)
    {
        throw new Exception("La existencia no puede ser negativa.");
    }

    lock (ObjectToLock)
    {
        if (UnitsInStock >= requestedUnits)
        {
            // Simulamos un proceso de larga duración
            Thread.Sleep(1000);
            Console.WriteLine($"Existencia antes del pedido: {UnitsInStock}");
            // Restamos a la existencia la cantidad solicitada.
            UnitsInStock -= requestedUnits;
            Console.WriteLine($"Existencia después del pedido: {UnitsInStock}");
            Accepted = true;
        }
        else
        {
            Console.WriteLine(
                $"Existencia insuficiente: {requestedUnits} de {UnitsInStock}");
        }
    }
    return Accepted;
}
```

10. Ejecuta la aplicación. Podrás notar que la aplicación se ejecuta y finaliza sin problema.



```
C:\Demos\csasync\Activity07\Activity07\bin\Debug\Activity07.exe
Existencia insuficiente: 1 de 0
Existencia insuficiente: 14 de 0
Existencia insuficiente: 82 de 0
Existencia insuficiente: 81 de 0
Existencia insuficiente: 52 de 0
Existencia insuficiente: 71 de 0
Existencia insuficiente: 1 de 0
Existencia insuficiente: 77 de 0
Existencia insuficiente: 83 de 0
Existencia insuficiente: 42 de 0
Existencia insuficiente: 56 de 0
Existencia insuficiente: 80 de 0
Existencia insuficiente: 23 de 0
Existencia insuficiente: 25 de 0
Presione <enter> para finalizar la aplicación...
```

11. Regresa a Visual Studio y detén la ejecución.



Es importante mencionar que internamente, la instrucción **lock** utiliza otro mecanismo de bloqueo mediante los métodos **Monitor.Enter** y **Monitor.Exit** para aplicar un bloqueo de exclusión mutua a una sección crítica de código.

La instrucción **lock**, encapsula a **Monitor.Enter**, el bloque **try-finally** y a **Monitor.Exit** dentro del bloque **finally**.

12. Modifica el código del método **PlaceOrder** para ejemplificar la forma en que **lock** realiza el bloqueo de exclusión mutua encapsulando a **Monitor.Enter**, el bloque **try-finally** y a **Monitor.Exit** dentro del bloque **finally**.

```
public bool PlaceOrder(int requestedUnits)
{
    bool Accepted = false;

    // UnitsInStock jamás debería ser una cantidad negativa
    if (UnitsInStock < 0)
    {
        throw new Exception("La existencia no puede ser negativa.");
    }

    //lock (ObjetoToLock)
    Monitor.Enter(ObjectToLock);
    try
    {
        if (UnitsInStock >= requestedUnits)
        {
            // Simulamos un proceso de larga duración
            Thread.Sleep(1000);
            Console.WriteLine($"Existencia antes del pedido: {UnitsInStock}");
            // Restamos a la existencia la cantidad solicitada.
            UnitsInStock -= requestedUnits;
            Console.WriteLine($"Existencia después del pedido: {UnitsInStock}");
            Accepted = true;
        }
        else
        {
            Console.WriteLine(
                $"Existencia insuficiente: {requestedUnits} de {UnitsInStock}");
        }
    }
    finally
    {
        Monitor.Exit(ObjectToLock);
    }
    return Accepted;
}
```

13. Ejecuta la aplicación. Podrás notar que la aplicación se ejecuta y finaliza sin problema.



```
C:\Demos\csasync\Activity07\Activity07\bin\Debug\Activity07.exe
Existencia insuficiente: 10 de 0
Existencia insuficiente: 30 de 0
Existencia insuficiente: 36 de 0
Existencia insuficiente: 68 de 0
Existencia insuficiente: 58 de 0
Existencia insuficiente: 49 de 0
Existencia insuficiente: 31 de 0
Existencia insuficiente: 13 de 0
Existencia insuficiente: 79 de 0
Existencia insuficiente: 93 de 0
Existencia insuficiente: 19 de 0
Existencia insuficiente: 85 de 0
Existencia insuficiente: 34 de 0
Existencia insuficiente: 33 de 0
Existencia insuficiente: 56 de 0
Existencia insuficiente: 81 de 0
Presione <enter> para finalizar la aplicación...
```

En este ejercicio mostramos el uso de la palabra clave **lock** para aplicar un **bloqueo de exclusión mutua** (*mutual-exclusion*) a secciones críticas de código.



Para obtener más información sobre la instrucción **lock**, se recomienda consultar el siguiente enlace:

lock Statement (C# Reference)

<http://go.microsoft.com/fwlink/?LinkID=267848>

Thread Synchronization

<http://go.microsoft.com/fwlink/?LinkID=267849>