Ejemplos - Threading

Un thread es un flujo de instrucciones independientes en un programa. Un thread es similar a un programa secuencial. Sin embargo, un thread en sí no es un programa, no puede ejecutarse solo, sino que se ejecuta dentro del contexto de un programa.

El uso real de un thread no se trata de un solo thread secuencial, sino más bien el uso de múltiples threads en un solo programa, es decir, de varios threads que se ejecutan al mismo tiempo y que realizan varias tareas se denominan multithreading. Se considera que un thread es un proceso ligero porque se ejecuta dentro del contexto de un programa y aprovecha los recursos asignados para ese programa.

Con el administrador de tareas de Windows, podemos activar la columna Procesos y ver los procesos y la cantidad de threads para cada proceso.

El sistema operativo programa los hilos. Un thread tiene una prioridad y cada thread tiene su propia pila, pero la memoria para el código del programa y el heap se comparten entre todos los threads de un solo proceso.

Un proceso consiste en uno o más hilos de ejecución. Un proceso siempre consta de al menos un thread denominado thread principal (método **Main ()**). Un proceso de un solo thread contiene solo un thread mientras que un proceso de multithreading contiene más de un thread para la ejecución.

En una computadora, el sistema operativo carga e inicia las aplicaciones. Cada aplicación o servicio se ejecuta como un proceso separado en la máquina.

El Espacio de Nombres System.Threading

El espacio de nombres System.Threading nos provee de varios tipos de ayuda para construir aplicaciones multithreading.

Tipo	Descripción
Thread	Representa un thread que se ejecuta en el CLR. Al usar esto podemos crear threads adicionales en la aplicación de dominio.
Mutex	Se usa para la sincronización entre aplicaciones de dominio.
Monitor	Implementa la sincronización de objetos usando Locks y Waihilo.
Semaphore	Nos permite limitar el número de threads que pueden acceder a un recurso concurrentemente.
Interlock	Nos provee de operaciones atómicas para varialbes que están compartidas en multiples threads.
ThreadPool	Te permite interactuar con el CLR manteniendo un pool de threads.
ThreadPriority	Representa el nivel de prioridad como High, Normal, Low.

La Clase System.Threading.Thread

La clase **Thread** nos permite crear y manejar la ejecución de threads en el programa. Estos threas se conocen como threads administrados.

Miembro	Descripción
CurrentThread	Devuelve una referencia al thread que se está ejecutando actualmente.
Sleep	Suspende el thread actual durante un tiempo específico.
GetDomain	Devuelve una referencia del dominio de aplicación actual.
CurrentContext	Devuelve una referencia del contexto actual en el que se está ejecutando el thread actualmente.
Priority	Obtiene o Establece el nivel de prioridad del Thread.
IsAlive	Obtener el estado del thread en forma de un valor true o false.
Start	Indica al CLR que inicie el thread.
Suspend	Suspende el thread.
Resume	Reanuda un thread previamente suspendido.
Abort	Indica al CLR que termine el hilo.
Name	Permite establecer un nombre a un thread.
IsBackground	Indica si un thread se está ejecutando en segundo plano o no.

Ejemplos de Multithreading

Obtención de información del hilo actual

Supongamos que tenemos una aplicación de consola en la que la propiedad **CurrentThread** recupera un objeto **Thread** que representa el hilo que se está ejecutando actualmente.

```
using System;
using System.Threading;

namespace threading
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("**********Información Hilo
Actual**************\n");
            Thread hilo = Thread.CurrentThread;
            hilo.Name = "Hilo Primario";
            Console.WriteLine("Nombre del Hilo: {0}", hilo.Name);
            Console.WriteLine("Estado del Hilo: {0}", hilo.IsAlive);
            Console.WriteLine("Prioridad: {0}", hilo.Priority);
            Console.WriteLine("ID Contexto: {0}",
Thread.CurrentContext.ContextID);
            Console.WriteLine("Dominio de la aplicación actual: {0}",
Thread.GetDomain().FriendlyName);
            Console.ReadKey();
        }
```

```
    }

}
```

Resultado

```
**********Información Hilo Actual***************



Nombre del Hilo: Hilo Primario

Estado del Hilo: True

Prioridad: Normal

ID Contexto: 0

Dominio de la aplicación actual: ConsoleApplication8.vshost.exe
```

Creación de un Thread

En el siguiente ejemplo vemos la implementación de la clase **Thread** en la que el constructor de la clase **Thread** acepta un parámetro delegado. Una vez creado el objeto de clase **Thread**, podemos iniciar el hilo con el método **Start ()** como se muestra a continuación:

```
using System;
using System.Threading;

namespace threading
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread hilo = new Thread(otroHilo);
            hilo.Start();
            Console.WriteLine("Hilo Principal Ejecutándose");
            Console.ReadKey();
        }
        static void otroHilo()
        {
            Console.WriteLine("Ejecutando otro Hilo");
        }
    }

}
```

Resultado

```
Ejecutando otro Hilo

Hilo Principal Ejecutándose
```

El punto importante que se debe tener en cuenta aquí es que, no hay garantía de que la salida sea lo primero, en otras palabras, qué hilo comienza primero. Los hilos son programados por el sistema operativo. Entonces, qué hilo viene primero puede ser diferente cada vez.

Thread en Segundo Plano

El proceso de la aplicación se sigue ejecutando mientras se esté ejecutando al menos un thread en primer plano. Si se está ejecutando más de un thread en primer plano y el método **Main ()** finaliza, el proceso de la aplicación se mantiene activo hasta que todos los threads en primer plano finalicen su trabajo, y al finalizar el thread, todos los threads en segundo plano terminarán de inmediato.

Cuando creas un hilo con la clase Thread, puedes definirlo como un hilo de primer plano o de segundo plano configurando la propiedad **IsBackground**. El método **Main ()** establece esta propiedad del thread "hilo" en false. Después de configurar el nuevo hilo, el hilo principal simplemente escribe en la consola un mensaje final. El nuevo hilo escribe un inicio y un mensaje de finalización, y entre ellos se para durante 2 segundos.

```
using System;
using System.Threading;

namespace threading
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread hilo = new Thread(miHilo);
            hilo.Name = "Hilo1";
            hilo.IsBackground = false;
            hilo.Start();
            Console.WriteLine("Hilo Principal en Ejecución");
            Console.ReadKey();
        }
        static void miHilo()
        {
            Console.WriteLine("El hilo {0} ha comenzado",
Thread.CurrentThread.Name);
            Thread.Sleep(2000);
            Console.WriteLine("El hilo {0} se ha completado",
Thread.CurrentThread.Name);
        }
    }

}
```

Resultado

```
El hilo Hilo1 ha comenzado

Hilo Principal en Ejecución

El hilo Hilo1 se ha completado
```

Si cambiamos la propiedad **isBackground** a true el resultado que nos mostrará la consola como vemos a continuación:

Resultado

```
El hilo Hilo1 ha comenzado

Hilo Principal en Ejecución
```

Race Condition

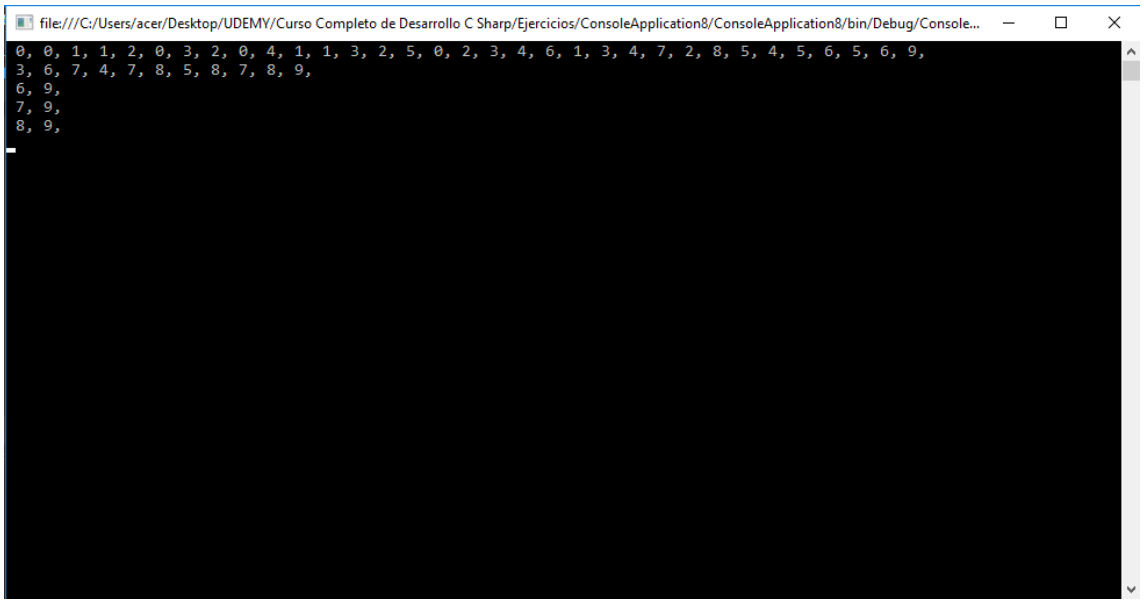
Se produce una race condition si dos o más threads acceden al mismo objeto y el acceso al estado compartido no está sincronizado. Para comprender mejor este concepto, vamos a ver un ejemplo. Esta aplicación utiliza la clase Prueba para imprimir 10 números haciendo una pausa en el hilo actual por un número aleatorio de veces.

```
using System;
using System.Threading;
namespace threading
{
    public class Prueba
    {
        public void Calcular()
        {
            for (int i = 0; i < 10; i++)
            {
                Thread.Sleep(new Random().Next(5));
                Console.Write(" {0},", i);
            }
            Console.WriteLine();
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Prueba prueba = new Prueba();
            Thread[] hilo = new Thread[5];
            for (int i = 0; i < 5; i++)
            {
                hilo[i] = new Thread(new ThreadStart(prueba.Calcular));
                hilo[i].Name = String.Format("Thread Trabajando: {0}", i);
            }
```

```
            //comienza cada thread
            foreach (Thread x in hilo)
            {
                x.Start();
            }
            Console.ReadKey();
        }
    }

}
```

Después de compilar este programa, el thread principal dentro de este dominio de aplicación comienza produciendo cinco threads secundarios. Y a cada hilo de trabajo se le dice que llame al método **Calcular ()** en la misma instancia de la clase Prueba. Por lo tanto, los cinco threads comienzan a acceder al método **Calcular ()** simultáneamente y, como no hemos tomado ninguna precaución para bloquear los recursos compartidos de este objeto; esto conducirá hacia la Race Condition y la aplicación produce resultados impredecibles como podemos ver en los siguientes resultados diferentes:

Resultado



Deadlocks

Tener muchos bloqueos en una aplicación puede hacer que la aplicación tenga problemas. Con **Deadlocks**, al menos dos hilos se esperan el uno al otro para liberar un bloqueo. Cuando ambos threads se esperan, se produce una situación de interbloqueo y los threads esperan interminablemente y el programa deja de responder.

En este ejemplo, ambos métodos cambiaron el estado de los objetos objeto1 y objeto2 al bloquearlos. El método **DeadLock1 ()** primero bloquea el objeto1 y luego el objeto2. El método **DeadLock2 ()** primero bloquea el objeto2 y luego el objeto1. Por lo tanto, se

libera el bloqueo para el objeto1, luego se produce un cambio de hilo y el segundo método se inicia y obtiene el bloqueo para el objeto2. El segundo hilo ahora espera el bloqueo del objeto1. Ambos hilos ahora esperan y no se liberan. Esta es típicamente una situación de punto muerto.

```
using System;
using System.Threading;
namespace threading
{
    class Program
    {
        static object objeto1 = new object();
        static object objeto2 = new object();
        public static void DeadLock1()
        {
            lock (objeto1)
            {
                Console.WriteLine("El hilo 1 quedó bloqueado");
                Thread.Sleep(500);
                lock (objeto2)
                {
                    Console.WriteLine("El hilo 2 quedó bloqueado");
                }
            }
        }
        public static void DeadLock2()
        {
            lock (objeto2)
            {
                Console.WriteLine("El hilo 2 quedó bloqueado ");
                Thread.Sleep(500);
                lock (objeto1)
                {
                    Console.WriteLine("El hilo 2 quedó bloqueado ");
                }
            }
        }
        static void Main(string[] args)
        {
            Thread hilo1 = new Thread(new ThreadStart(DeadLock1));
            Thread hilo2 = new Thread(new ThreadStart(DeadLock2));
            hilo1.Start();
            hilo2.Start();
            Console.ReadKey();
        }
    }

}
```

Resultado

```
El hilo 2 quedó bloqueado

El hilo 1 quedó bloqueado
```

Lock

Lock es la palabra clave en C # que asegurará que un hilo esté ejecutando un fragmento de código a la vez. La palabra clave **Lock** garantiza que un thread no entre en una sección crítica del código mientras que otro thread está en esa sección crítica.

Lock es un método abreviado para adquirir un bloqueo para la pieza de código para un solo hilo.

```
using System;
using System.Threading;

namespace Bloqueo
{
    class Program
    {
        static readonly object objecto = new object();

        static void PruebaLock()
        {

            lock (objecto)
            {
                Thread.Sleep(100);
                // muestra los milisegundos desde que se inició del sistema
                Console.WriteLine(Environment.TickCount);
            }
        }

        static void Main(string[] args)
        {
            for (int i = 0; i < 10; i++)
            {
                ThreadStart inicio = new ThreadStart(PruebaLock);
                new Thread(inicio).Start();
            }

            Console.ReadKey();
        }
    }

}
```

Resultado

```
10397953

10398187

10398359

10398484

10398656
```

```
10398828

10399140

10399328

10399500

10399640
```

Aquí vemos un método estático "**PruebaLock**" que usa la declaración de **lock** en un objeto. Cuando se llama muchas veces al método **PruebaLock** en los nuevos threads, cada invocación del método accede a los primitivos de subprocesamiento implementados por el bloqueo.

El método **Main** crea diez nuevos hilos y luego llama a **Start** en cada uno. El método **PruebaLock** se invoca diez veces, pero el recuento de **ticks** muestra que la región del método protegido se ejecuta de forma secuencial, con una diferencia de 100 milisegundos.

Si otro hilo intenta acceder al código bloqueado, esperará, se bloqueará, hasta que se libere el objeto.

La palabra clave **lock** marca un bloque de instrucción como una sección crítica al obtener el bloqueo de exclusión mutua para un objeto determinado, ejecutar una declaración y luego liberar el Monitor de bloqueo

Monitor

Monitor proporciona un mecanismo que sincroniza el acceso a los objetos. Se puede hacer adquiriendo un bloqueo de modo que solo un hilo pueda acceder a un fragmento de código dado a la vez. El monitor no es diferente del bloqueo, pero la clase del monitor proporciona más control sobre la sincronización de varios threads que intentan acceder al mismo bloqueo de código.

Al usar **Monitor**, se puede garantizar que ningún otro thread pueda acceder a una sección del código de la aplicación que está ejecutando el propietario del bloqueo, a menos que el otro thread esté ejecutando el código utilizando un objeto bloqueado diferente.

La clase Monitor tiene los siguientes métodos para sincronizar el acceso a una sección del código al tomar y liberar un bloqueo:

- Monitor.Enter
- Monitor.TryEnter
- Monitor.Exit

El monitor bloquea objetos, es decir, tipos de referencia, no tipos de valor. Si bien puede pasar un tipo de valor a **Enter** y **Exit**, se encuadra por separado para cada llamada.

Wait libera el bloqueo si se mantiene y espera a que se le notifique. Cuando se notifica la espera, vuelve y obtiene el bloqueo nuevamente. Tanto la señal Pulse como **PulseAll** para el siguiente hilo en la cola de espera para continuar.

Veamos un Ejemplo:

```
using System;
using System.Threading;

namespace Monitor_Bloqueo
{
    class Program
    {
        static readonly object objecto = new object();

        public static void ImprimirNumeros()
        {
            Monitor.Enter(objecto);
            try
            {
                for (int i = 0; i < 5; i++)
                {
                    Thread.Sleep(100);
                    Console.Write(i + ",");
                }
                Console.WriteLine();
            }
            finally
            {
                Monitor.Exit(objecto);
            }
        }

        static void Main(string[] args)
        {


            Thread[] hilos = new Thread[3];
            for (int i = 0; i < 3; i++)
            {
                hilos[i] = new Thread(new ThreadStart(ImprimirNumeros));
                hilos[i].Name = "Hijo " + i;
            }
            foreach (Thread hilo in hilos)
                hilo.Start();

            Console.ReadLine();
        }
    }

}
```

Resultado

```
0,1,2,3,4,
```

```
0,1,2,3,4,

0,1,2,3,4,
```

La clase Monitor es una clase estática y no se puede crear su instancia.

El objeto de clase Monitor utiliza los métodos **Monitor.Enter**, **Monitor.TryEnter** y **Monitor.Exit**. Una vez que tenga un bloqueo en una sección de código, puede usar los métodos **Monitor.Wait**, **Monitor.Pulse** y **Monitor.PulseAll**.

Se asocia con un objeto bajo demanda. Es ilimitado, lo que significa que se puede llamar directamente desde cualquier contexto.

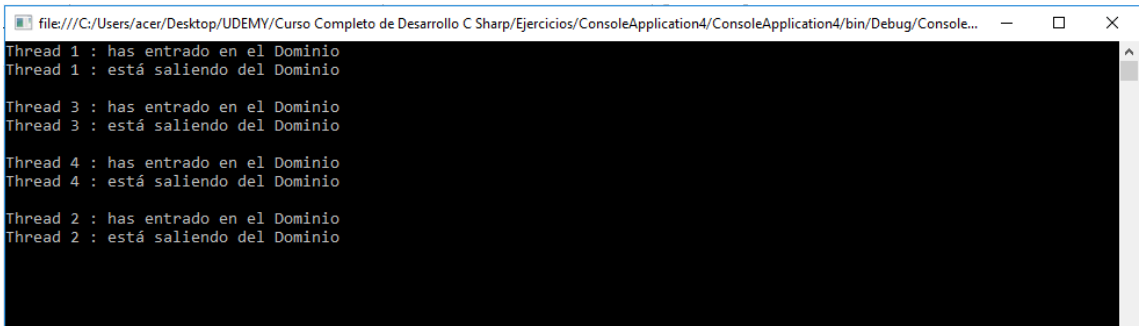
Mutex

Mutex significa **Mutual Exclusion** y nos ofrece sincronización a través de múltiples threads. La clase de **Mutex** se deriva de **WaitHandle**, puede hacer un **WaitOne ()** para adquirir el bloqueo de mutex y ser el propietario del mutex esa vez. El mutex se libera invocando el método **ReleaseMutex ()** como se muestra a continuación.

```
using System;
using System.Threading;

namespace threading
{
    class Program
    {
        private static Mutex mutex = new Mutex();
        static void Main(string[] args)
        {
            for (int i = 0; i < 4; i++)
            {
                Thread hilo = new Thread(new ThreadStart(EjemploMutex));
                hilo.Name = string.Format("Thread {0} :", i + 1);
                hilo.Start();
            }
            Console.ReadKey();
        }
        static void EjemploMutex()
        {
            try
            {
                // Esperar hasta que sea seguro entrar
                mutex.WaitOne();
                Console.WriteLine("{0} has entrado en el Dominio",
Thread.CurrentThread.Name);
                // Esperar hasta que sea seguro entrar
                Thread.Sleep(1000);
                Console.WriteLine("{0} está saliendo del Dominio\r\n",
Thread.CurrentThread.Name);
            }
            finally
```

```
            {
                mutex.ReleaseMutex();
            }
        }
    }
}
```

Resultado



Cuando ejecutamos este programa, se muestra cuando cada nuevo hilo entra por primera vez en su dominio de aplicación. Una vez que ha terminado sus tareas, se libera y comienza el segundo hilo, y así sucesivamente.

Semaphore

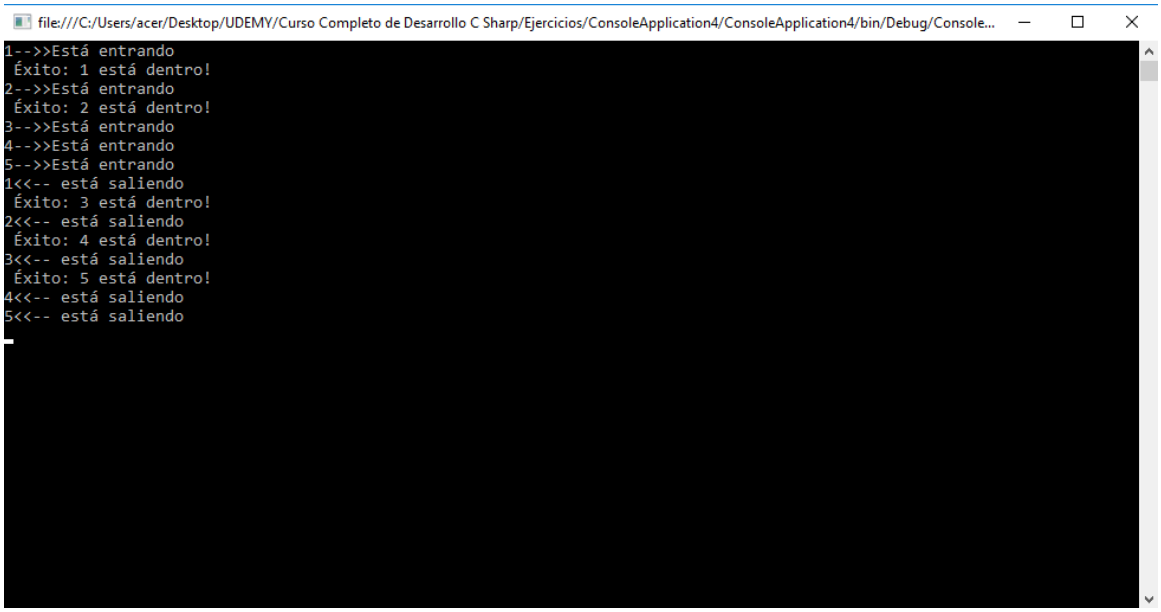
Un **Semaphore** es muy similar a un **Mutex** pero un semáforo puede ser usado por varios hilos a la vez mientras que un **Mutex** solo puede ser usado por uno. Con un semáforo, puedes definir un conteo de cuántos threads pueden acceder a los recursos protegidos por un semáforo simultáneamente.

En el siguiente ejemplo, se crean 5 hilos y 2 semáforos. En el constructor de la clase de semáforo, puedes definir el número de bloqueos que se pueden adquirir con un semáforo.

```
using System;
using System.Threading;

namespace semaforo_ejemplo
{
    class Program
    {
        static Semaphore semaforo = new Semaphore(2, 4);
        static void Main(string[] args)
        {
            for (int i = 1; i <= 5; i++)
            {
                new Thread(semaforoInicio).Start(i);
            }
            Console.ReadKey();
        }
        static void semaforoInicio(object id)
```

```
        {
            Console.WriteLine(id + "-->>Está entrando");
            try
            {
                semaforo.WaitOne();
                Console.WriteLine(" Éxito: " + id + " está dentro!");
                Thread.Sleep(2000);
                Console.WriteLine(id + "<<-- está saliendo");
            }
            finally
            {
                semaforo.Release();
            }
        }
    }

}
```

Resultado



Mientras ejecutamos esta aplicación, se crean inmediatamente 2 semáforos y los demás esperan porque hemos creado 5 threads. Así que 3 threads están en estado de espera. El movimiento, cualquiera de los hilos soltó el resto del creado uno por uno como sigue.

ThreadingPool

El **Thread Pooling** es una colección de threads que se pueden utilizar para no realizar ninguna tarea en segundo plano. Una vez que el thread completa su tarea, se envía al grupo a una cola de threads en espera, donde se puede reutilizar. Esta reutilización evita que una aplicación cree más threads y permite menos consumo de memoria.

Una vez que se completa la tarea en el grupo de threads, .NET framework envía estos trabajos completados a un grupo de threads en lugar de enviar estos threads al recolector de basura. Además, se puede reutilizar para otra nueva tarea.

La agrupación de threads es esencial en aplicaciones multithreading por las siguientes razones.

- El ThreadPool mejora el tiempo de respuesta de una aplicación, ya que los hilos ya están disponibles en la agrupación de hilos en espera de su próxima asignación y no es necesario crearlas desde cero.
- El ThreadPool ahorra al CLR de la sobrecarga de crear un thread completamente nuevo para cada tarea de corta duración y reclamar sus recursos una vez que muere.
- El ThreadPool optimiza los segmentos de tiempo de threads de acuerdo con el proceso actual que se ejecuta en el sistema.
- El ThreadPool nos permite iniciar varias tareas sin tener que establecer las propiedades de cada thread.
- El ThreadPool nos permite indicar información de un objeto para los argumentos de procedimiento de la tarea que se está ejecutando.
- El ThreadPool se puede emplear para fijar el número máximo de threads para procesar una solicitud en particular.

Por lo tanto, trabajar con un grupo de threads nos permite un menor consumo de memoria para una gran cantidad de tareas.

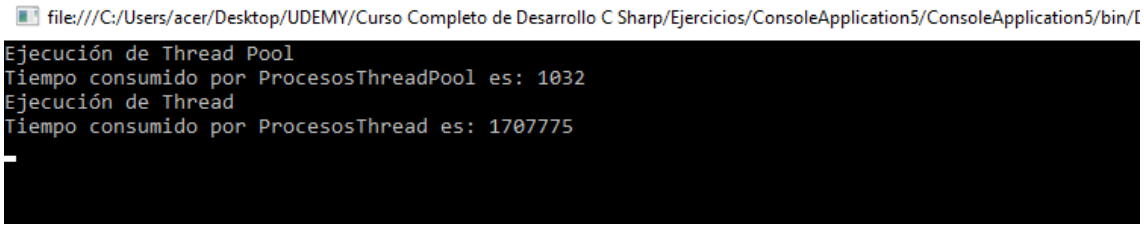
A continuación, vamos a ver un ejemplo comentado:

```
using System;
// Paso 1
// Para implementar la agrupación de threads en una aplicación, necesitamos usar
el espacio de nombres de threads como se muestra en el siguiente código.
using System.Threading;
using System.Diagnostics;

namespace ThreadPooling
{
    class Program
    {
        static void Main(string[] args)
        {
            // Paso 5
            // Ahora solo necesitamos llamar estos métodos a nuestro método
principal, ya que estamos probando el consumo de tiempo, por lo que llamaremos
estos métodos entre el inicio y el final del cronómetro.
            Stopwatch control = new Stopwatch();

            Console.WriteLine("Ejecución de Thread Pool");

            control.Start();
            ProcesosThreadPool();
            control.Stop();

            Console.WriteLine("Tiempo consumido por ProcesosThreadPool es: " +
control.ElapsedTicks.ToString());
            control.Reset();
```

```
            Console.WriteLine("Ejecución de Thread");

            control.Start();
            ProcesosThread();
            control.Stop();

            Console.WriteLine("Tiempo consumido por ProcesosThread es: " +
control.ElapsedTicks.ToString());
            Console.ReadKey();
        }

        // Paso 4
        // Definimos los métodos que llamarán al método Proceso
        static void ProcesosThreadPool()
        {
            for (int i = 0; i <= 10; i++)
            {
                // Paso 2
                // Después debemos llamar a la clase de agrupación de threads
ThreadPool, para usar el objeto de agrupaciones de threads, debemos llamar al
método "QueueUserWorkItem" que nos provee de una función de colas para una
ejecución y una función que ejecuta cuando un thread está disponible desde el
conjunto de threads.
                // ThreadPool.QueueUserWorkItem toma un método sobrecargado
llamado waitcallback que representa una función de devolución de llamada para ser
ejecutada por un threadpool de threads. Si no hay un hilo disponible, esperará
hasta que se libere uno.
                // Donde Proceso () es el método que ejecutará un thread de
threads. En este paso, compararemos cuánto tiempo tarda el objeto de thread y
cuánto tiempo tarda el grupo de threads en ejecutar cualquier método.
                ThreadPool.QueueUserWorkItem(new WaitCallback(Proceso));
            }
        }

        // Paso 4
        // Definimos los métodos que llamarán al método Proceso
        static void ProcesosThread()
        {
            for (int i = 0; i <= 10; i++)
            {
                Thread hilo = new Thread(Proceso);
                hilo.Start();
            }
        }

        // Paso 3
        // Para ello, crearemos dos métodos llamados ProcesosThread() y
ProcesosThreadPool() y dentro crearemos un bucle que itere 10 veces. Estos dos
métodos llamarán a un método simple llamado "Proceso()" que tiene un objeto de
entrada como parámetro ya que waitcallback necesita un parámetro de entrada para
las devoluciones de llamada.
        // Este método se llamarán mediante dos métodos para ejecutar:
ProcesosThread() y ProcesosThreadPool()
        static void Proceso(object callback)
        {

        }


    }
```

```
}
```

Resultado



FIN