**Evaluation Only. Created with Aspose.Words. Copyright 2003-2021 Aspose Pty Ltd.**

**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);

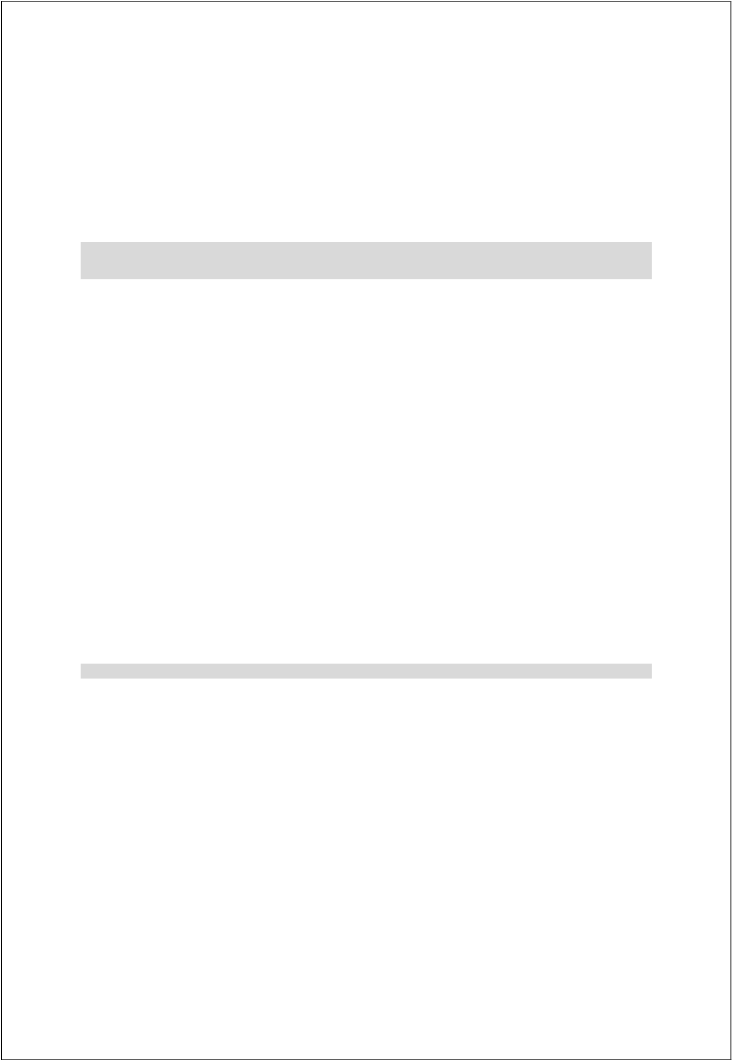
Console.WriteLine("ID Contexto: {0}", Thread.CurrentContext.ContextID);

Console.WriteLine("Dominio de la aplicación actual: {0}", Thread.GetDomain().FriendlyName);

Console.ReadKey();

}

Ejemplos - Threading – Curso Completo de Desarrollo C Sharp – Ángel Arias



} }

**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()

{

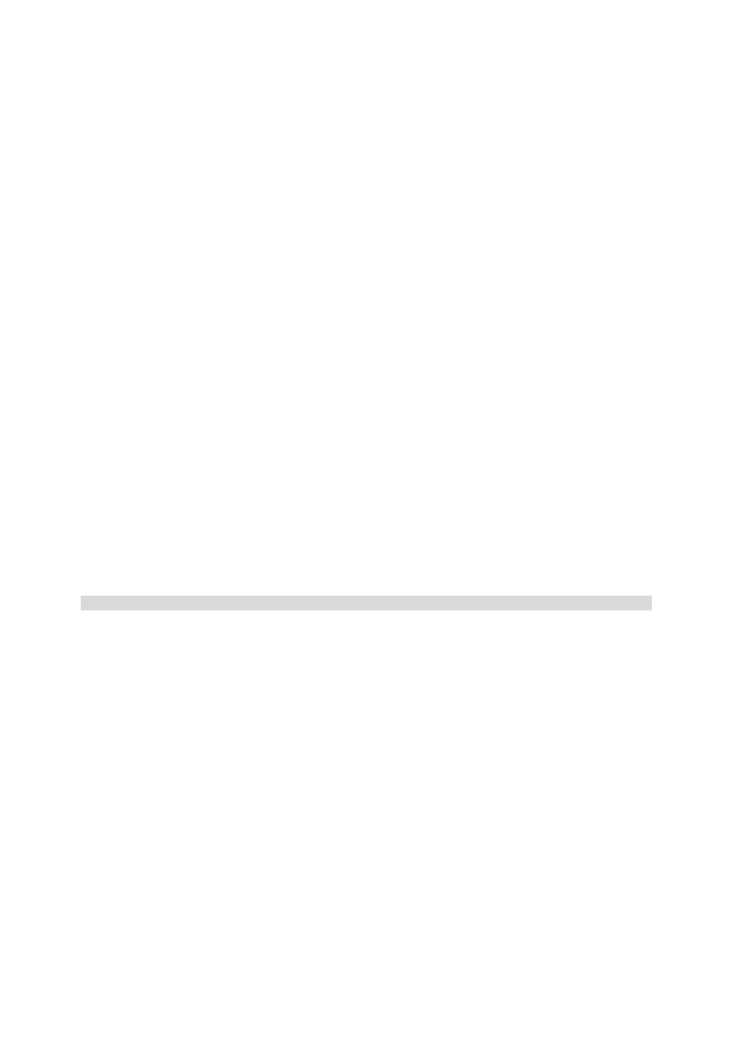
Console.WriteLine("Ejecutando otro Hilo");

}

}

}

Ejemplos - Threading – Curso Completo de Desarrollo C Sharp – Ángel Arias



**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();

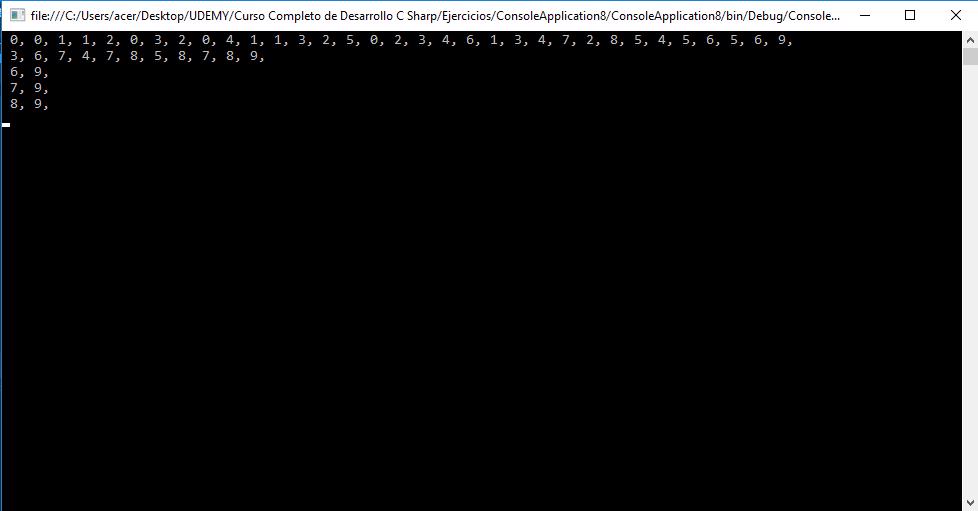
}

}

}

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)

{

**This document was truncated here because it was created in the Evaluation Mode.**