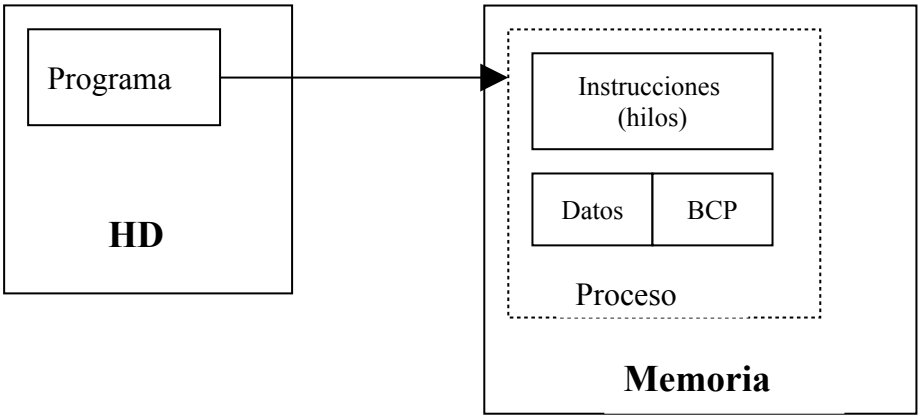
Multitasking and multithreading (Multitarea y multihilo)

Introducción y repaso

Un proceso es un conjunto de recursos que usa el sistema operativo para disponer de un programa en ejecución. Está compuesto por el ejecutable principal, una serie de librerías y la memoria que necesite. Además el SO dispone de una serie de datos (el Bloque de Control de Proceso) sobre el propio proceso (Identificador único, zonas de memoria ocupadas, etc...).



Como ya se vio el curso pasado, el administrador de tareas (*taskmanager*) , el comando *tasklist* o *wmic process* en la consola de windows nos muestra los BCPs de los procesos que hay en estos momentos en ejecución. En el caso de un sistema UNIX se puede ver con el comando *ps -ef* o con *top.*

Los procesos deben ser independientes y el fallo de uno no debiera afectar al mal funcionamiento de otros.

En la informática actual disponemos de equipos de gran potencia computacional, mucho más de lo que se necesita para el trabajo diario de una única aplicación típica. Esto nos lleva a querer dividir el tiempo de uso de CPU entre varios procesos de forma que parezca al usuario que todos están ejecutándose al mismo tiempo además de aprovechar mejor los recursos que tenemos. Además algunos procesadores disponen de varios núcleos (o gpus u ordenadores con varios procesadores) lo que propicia esta ejecución simultánea de aplicaciones. Esto es lo que permite la **programación multiproceso** o concurrente.

Pero yendo un poco más allá, también es cierto que en una aplicación podemos desear realizar varias tareas "a la vez" o sería mejor decir de forma concurrente. Es decir, que si disponemos de varias funciones, que estas se puedan estar ejecutando de forma paralela. Incluso una única función puede estar ejecutándose varias veces con distintos parámetros. Esto lo logramos utilizando **programación multihilo.**

Un proceso dispone de al menos un hilo o thread principal (primary thread). Se puede realizar un programa de forma que disponga de varios hilos ejecutándose en paralelo.

No se debe confundir la modularidad en funciones con la creación de hilos. Una función se puede ejecutar de forma secuencial detrás de otra (es lo que venimos haciendo hasta el momento) o se puede lanzar en paralelo con otra (creando al menos un hilo o subproceso además del principal) o incluso consigo misma (piensa en varios archivos descargándose de un torrent).

Cuando se usa un único hilo no tenemos problemas de seguridad en el acceso a los datos dentro de la aplicación pues sólo los va a manejar un subproceso. El problema es que esto puede causar que un trabajo que lleve tiempo (impresión, cálculo intensivo, etc.) lleve al bloqueo temporal del programa. En estos casos es mejor sacar subprocesos y que unos se encarguen de cierto trabajo en background y otros de la interfaz de usuario y otros menesteres. Así numerosos subprocesos (o procesos) dan la ilusión de estar realizando varias tareas a la vez.

Veamos algunos conceptos que es necesario comprender para el desarrollo de este tema:

Concurrency (Concurrencia): la ejecución de varios procesos al mismo tiempo compartiendo una o varias cpus, o varios núcleos de una cpu. Además estos procesos pueden estar interactuando entre sí lo que requiere cierto nivel de sincronismo.

Paralelism (Paralelismo): El paralelismo es la descomposición de un programa en partes que son independientes y que se pueden ejecutar de forma simultánea en distintos procesadores o núcleos.

El concepto se diferencia de la concurrencia en que el paralelismo exige ejecución simultanea de al menos dos tareas y por tanto un sistema multikernel o multiprocesador mientras la concurrencia simplemente exige que el tiempo de ejecución de dos tareas se solape, pero puede ser en un sistema monoprocesador.

Multitask (Multitarea): característica de los sistemas operativos que indica que pueden ejecutar software concurrentemente.

Multithread (Multihilo): Varias tareas de un proceso se pueden separar en hilos que se ejecutan concurrentemente. Un sistema que permite esto es un sistema multithread. Los hilos comparten zona de memoria de la aplicación.

Asynchronous programming (Programación asíncrona): La programación que venimos haciendo hasta el momento es síncrona (y más o menos secuencial), es decir, para que empiece la ejecución de cierta parte de código tiene que haber terminado la anterior. En la asíncrona se permite que una parte del código se ejecute cuando todavía no ha terminado de ejecutarse otra parte.

Es sencillo comenzar a trabajar con concurrencia: crear threads, sincronizarlos de formas sencilla, etc... la dificultad está en que existen muchos problemas subyacentes que hacen de la concurrencia un campo complejo cuando trabajamos con varios hilos a un tiempo. Se pueden producir esperas excesiva e incluso bloqueos de programas por una mala sincronización de hilos (deadlock). Por tanto aunque sea una disciplina que da buenos frutos, hay que planificarla y pensarla realmente bien.

Ejemplo de problema de sincronización (**Race condition**):

1. El **Proceso 1** Accede a la **zona A** de memoria y **lee un dato** que ha de cambiar y almacenar más tarde (ej: valor=5).
2. Se acaba el tiempo de CPU para el Proceso 1 (ha leído, pero no ha guardado el dato cambiado).
3. Entra el **proceso 2** y **modifica** los datos de la **zona A** (ej: 5 → 7).
4. Vuelve el **proceso 1** y va a **guardar** el **dato modificado** (ej: 5 → 6)

El dato final es 6, pero se perdió la modificación del Proceso 2 (que debería ser 7, o 8 si se hubieran aplicado ambos cambios).

Esto es una **Race Condition** o Condición de Carrera: el resultado final depende del orden temporal en que los procesos acceden a datos compartidos, llevando a resultados inconsistentes y no deterministas.

Por esto es necesario **sincronizar el acceso a recursos comunes.**

En el caso anterior, la solución sería:

1. Proceso 1 accede a la zona A (**sección crítica**) y la bloquea (zona de **exclusión mutua**).
2. Proceso 1 es interrumpido (pero mantiene el bloqueo de la zona A).
3. Proceso 2 intenta acceder a la zona A, pero el SO lo bloquea (wait).
4. Vuelve el Proceso 1, completa su trabajo y libera la zona A.
5. El Proceso 2 puede entonces acceder a la zona A.

Incluso se podría producir el bloqueo de varios procesos porque esperan a que otro haga algo y si entran en círculo el sistema se bloquea (**deadlock o abrazo mortal**).

Un código o componente es **thread-safe** cuando puede ser utilizado por múltiples hilos simultáneamente sin causar condiciones de carrera (race conditions), corrupción de datos o comportamientos inconsistentes.

Otras características

- Para trabajar en multihilo o multiproceso se debe dividir el tiempo de CPU entre los distintos procesos y subprocesos. En Windows, los time-slice (fracciones de tiempo dedicado a un proceso o subproceso) son de decenas de milisegundos mientras que el switching entre hilos es del orden de unos pocos microsegundos, de ahí que interese el uso de multitarea/multihilo por perder muy poco tiempo.

- En un ordenador con un único procesador se usa sólo time-slicing, es decir, se divide el tiempo de cpu entre los procesos o hilos. Si hay varios procesadores o núcleos tenemos una mezcla de división temporal (time slicing) y concurrencia real.

- Un proceso no sabe cuando lo van a sacar de la cpu (sistema preemptive).

- Los hilos y los procesos son muy similares. La principal diferencia es que los procesos son entes aislados mientras que los hilos comparten una zona de memoria (**heap**) con otros hilos dentro del mismo proceso lo que da mucha versatilidad y eficiencia(por ejemplo un hilo puede estar cargando datos del disco duro en memoria y otro presentando esos datos actualizados en pantalla).

- Cuando se trabaja con hilos hay poco control sobre los mismos. En el momento que lanzamos un hilo, lo único que hacemos es decirle al SO que lo ejecute tan pronto como pueda, pero no sabemos en qué momento se va a ejecutar. Podemos establecer prioridades pero es no asegura un control fuerte del mismo.

¿Por qué es necesaria conocer la concurrencia?

- Aprovechar hardware multi-núcleo/GPUs para eficiencia.
- Permitir múltiples apps simultáneas.
- Interfaces responsivas (no bloqueadas por tareas).
- Servicios multi-cliente (FTP, P2P, BBDD).
- Simulaciones realistas (juegos, ciencia).

Programación multithreading o multihilo

Hasta el momento vimos como tener cierto control sobre procesos ejecutados en el sistema y también vimos la posibilidad de ejecutar nuevos procesos. Pero la importancia de este tema está sobre todo en la gestión de múltiples subprocesos (threads, hebras, hilos,...) de una aplicación de forma que **un único programa pueda realizar muchas tareas de forma concurrente** sobre todo de cara al usuario o de cara a posibles clientes en una arquitectura cliente-servidor.

Esta gestión de procesos la vamos a realizar en principio mediante la clase **Thread** de .Net.

Posteriormente veremos **Task,** que es una alternativa moderna a Thread, más sencilla y eficiente.

Task facilita la programación asíncrona y el manejo de excepciones.

Clase Thread

La complejidad en el desarrollo de aplicaciones multithread no está en la creación de hilos, que veremos que es una tarea simple, si no en la gestión del acceso concurrente a los datos de la aplicación que están compartidos por sus múltiples hilos. Esto lo vimos en el ejemplo del principio del tema.

Hay que tener especial cuidado porque en .Net la mayoría de las operaciones son **no atómicas y no thread-safe** lo que hace que sean peligrosas y hay que tener gran cuidado con ellas.

```
// PROBLEMA: operación no atómica (múltiples pasos que pueden ser interrumpidos)

contadorCompartido++; // No es thread-safe!

// Esto equivale a:
       // 1. Leer valor
       // 2. Incrementar
       // 3. Escribir valor
       // Entre estos pasos puede entrar otro hilo
```

Ejemplo de una aplicación con dos hilos.

Es necesario tener en using al espacio de nombres System.Threading:

```
static void Main(string[] args)
{
    Thread thread = new Thread(charA);
    thread.Start();

    //char B
    for (int i = 1; i < 1000; i++)
    {
        Console.Write('B');
    }
    Console.ReadKey();
}

static void charA()
{
    for (int i = 1; i < 1000; i++)
    {
        Console.Write('A');
    }
}
```

En este ejemplo el constructor de la clase Thread tiene como parámetro un delegado por lo que admite que se le pase una función como parámetro. Una vez que se ejecuta el Start, tenemos **dos hilos** corriendo: el **programa principal** y la función **charA**.

Prueba a ejecutar el programa varias veces y verás que como se dijo anteriormente una de las características del trabajo con hilos es que es el SO el encargado de gestionarlos, nosotros no podemos decir quién entra en cada momento salvo que usemos alguna herramienta de sincronización.

Control de Excepciones

El control de excepciones debe ser dentro de un hilo. Es decir, **si hago un try y dentro lanzo el hilo, la excepción que se produzca en el hilo no se controla**.

```
// INCORRECTO – no captura excepciones del hilo
try {
    thread.Start();
} catch {
    // ¡No captura excepciones del hilo!
}

// Lo adecuado es manejar excepciones dentro del hilo
```

Prioridad

Cambiando la prioridad (**Priority**) mediante el enumerado correspondiente no asegura un control preciso sobre cuando se va a ejecutar un hilo. Simplemente es información para el CLR de forma que le da una ayuda de qué hilos deberían entrar antes en la cpu o disponer de más tiempo. Por defecto la prioridad es Normal.

Prueba a hacer antes de hilo.Start() lo siguiente:

```
thread.Priority = ThreadPriority.Highest;
thread.Start();
```

Compara las ejecuciones. Quizá no veas mucha diferencia, pero fíjate que en sucesivas ejecuciones podrás observar que el hilo con prioridad Higher imprime más caracteres 'A' que 'B' en el mismo período.

Jerarquía de prioridades de menor a mayor:

```
Lowest→BelowNormal→Normal→AboveNormal→Highest
```

La prioridad del hilo principal es la del programa que se establece con *Process.PriorityClass.* Por ejemplo:

```
Process.GetCurrentProcess().PriorityClass = ProcessPriorityClass.High;
```

Más información de la clase Process en el Apéndice IV y en:

https://docs.microsoft.com/es-es/dotnet/api/system.diagnostics.process.priorityclass?view=netcore-3.1

Sincronización: bloqueos, pausas y esperas.

Como hemos visto, crear hilos individuales es sencillo. Sin embargo, en aplicaciones reales los hilos suelen **depender entre sí**: comparten datos, esperan resultados mutuos, o necesitan acceder a recursos limitados.

En este apartado aprenderemos a sincronizar hilos mediante mecanismos de espera y bloqueo. Para entender estos conceptos, es fundamental conocer los estados en los que puede encontrarse un hilo:

Running (En ejecución): Ejecutándose activamente en la CPU.

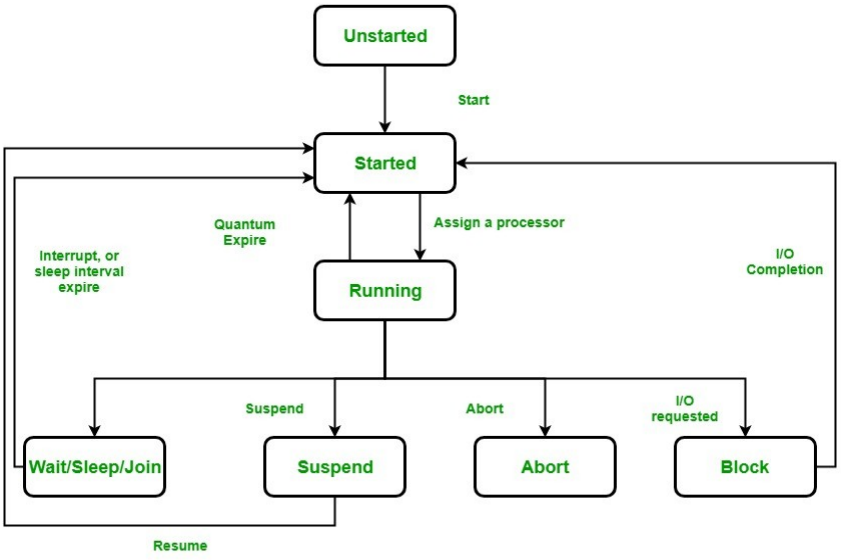
Ready/Preparado: Listo para ejecutarse, esperando su turno en la CPU

WaitSleepJoin: Bloqueado esperando por:

- Un recurso (Wait)
- Una pausa temporal (Sleep)
- La finalización de otro hilo (Join)

Blocked/Bloqueado: No puede ejecutarse porque le falta un recurso crítico

Terminated/Finalizado: Ha completado su ejecución



Puedes ver la explicación pormenorizada del esquema anterior en: https://www.geeksforgeeks.org/lifecycle-and-states-of-a-thread-in-c-sharp/

Lock

Mediante lock se va a permitir bloquear el acceso de hilos a una zona común mientras otro hilo la esté utilizando.

Veamos un ejemplo de por qué es necesario. Escribe el siguiente código:

```
class Program
{
    static bool running = true; // Booleana compartida para controlar los bucles

    static void Main(string[] args)
    {
        Thread thread = new Thread(writeDown);
        thread.Start();

        // writeUp
        int i = 1;
        while (running)
        {
            Console.SetCursorPosition(1, 1);
            Console.Write("{0,4}", i);
            i++;
            if (i >= 1000)
            {
                running = false;
            }
        }
        Console.ReadKey();
    }

    static void writeDown()
    {
        int i = 1;
        while (running)
        {
            Console.SetCursorPosition(1, 20);
            Console.Write("{0,4}", i);
            i++;
            if (i >= 1000)
            {
                running = false;
            }
        }
    }
}
```

¿Por qué crees que aparecen mal los números? Prueba a meter un `Thread.Sleep(10)` dentro de cada bucle para ver mejor lo que sucede.

Estamos ante una Race condition teniendo como recurso común la consola.

Se verá que en muchas ocasiones no coloca los números donde le decimos. Eso es porque después de ejecutar un SetCursorPosition salta al otro hilo y ejecuta el Write del otro hilo lo que hace que quede colocado en mala posición. Esto no tiene por que ocurrir siempre.

Para solucionarlo se usa un objeto único de referencia denominado **l** de forma que si un thread está bloqueando dicho objeto (mediante la sentencia **lock***),* no deja ejecutar a nadie más que quiera acceder a dicho objeto.

De esta forma queda así el programa:

```
class Program
{
    static bool running = true;
    static readonly object l = new object();

    static void Main(string[] args)
    {
        Thread thread = new Thread(writeDown);
        thread.Start();

        // writeUp
        int i = 1;
        while (running)
        {
            lock (l)
            {
                Console.SetCursorPosition(1, 1);
                Console.Write("{0,4}", i);
                i++;
                if (i >= 1000)
                {
                    running = false;
                }
            }
        }
        Console.ReadKey();
    }

    static void writeDown()
    {
        int i = 1;
        while (running)
        {
            lock (l)
            {
                Console.SetCursorPosition(1, 20);
                Console.Write("{0,4}", i);
                i++;
                if (i >= 1000)
                {
                    running = false;
                }
            }
        }
    }
}
```

Si está en el primero y salta al segundo, cuando intenta ejecutar uno de los comandos dentro del **lock(l)** del segundo no puede por estar cogido por el primero, así que vuelve al primero a terminar de ejecutarlo y cuando se libera si puede ir el segundo.

El objeto l actúa como un testigo físico (como la llave de un baño único). Solo el hilo que tiene el testigo puede ejecutar la sección crítica. Si otro hilo intenta adquirir el lock mientras está ocupado, queda bloqueado hasta que el testigo sea liberado.

Si la coge, entra en el recurso. Si el SO lo quita de la CPU, sale del recurso pero se lleva la llave de forma que si viene otro hilo que esté obligado a usar la misma llave no puede entrar. El *lock* es el comando que intenta "coger" el testigo. Si no puede, el hilo queda en bloqueo.

El recurso compartido por varios hilos es lo que se denomina sección crítica (**critical section**) y los algoritmos que evitan los problemas de acceso a la misma se denominan algoritmos de exclusión mutua (**Mutual Exclusion or Mutex**). En el ejemplo anterior sería la consola pero en otras ocasiones será una o varias variables, objetos, arvchivos...

C# incluye el modificador **volatile**, que resuelve problemas específicos de visibilidad entre hilos, pero **no es un reemplazo de lock** para la mayoría de casos de sincronización. Lo omitimos para centrarnos en conceptos universales de multihilo. Más información en :

https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/volatile

Sleep

Este comando permite que el hilo en el que se ejecuta entre en pausa durante el tiempo indicado en milisegundos.

Por ejemplo:

```
Thread.Sleep(500); // Pausa de 500ms (0.5 segundos)

// Casos especiales
Thread.Sleep(0); // Cede el resto del quantum al mismo priority
Thread.Sleep(1); // Cede a hilos de igual o menor prioridad
```

Características:

- Solo afecta al hilo actual - no bloquea otros hilos.
- El hilo entra en estado WaitSleepJoin durante la pausa.
- No consume tiempo de CPU mientras está suspendido.

Ojo porque **si el hilo está en una zona bloqueada con lock, esta permanece bloqueada** durante el sleep.

Consejos prácticos en el uso del *lock*

Finalmente vamos a resumir algunas recomendaciones a seguir cuando se realiza programación con hilos:

- Uso de **banderas en los while**: la forma de hacer que un hilo acabe de forma natural es gestionar adecuadamente las banderas booleanas en los bucles de repetición del hilo. Dichas banderas son accesibles por distintos hilos así desde uno se puede parar otro. Evidentemente por esto deben ser cambiadas dentro de un *lock*.
- Uso estructura **while-lock-if** para evitar una vuelta final de más en un bucle similar entre varios hilos. Esto en principio puede parecer absurdo pero vamos a entenderlo con un ejemplo.

  Tomemos el primer ejemplo donde se escribía A y B en la pantalla. Si resulta que queremos que cuando acabe uno, también termine el otro, lo mejor es escribir el código con bucles while dependiendes de una booleana. Veamos una versión un poco ampliada para que se vea mejor. Se usan distintos colores y se muestra también el número de veces que se ha escrito cada letra.

```
static bool running = true;
static readonly object l = new();

static void Main(string[] args)
{
    Thread threadA = new Thread(charA);
    Thread threadB = new Thread(charB);
    threadA.Start();
    threadB.Start();

    Console.ReadKey();
}
```

```
static void charA()
{
    int contA = 1;
    while (running)
    {
        lock (l)
        {
            Console.ForegroundColor = ConsoleColor.Green;
            Console.Write($" A:{contA}");
            contA++;
            if (contA > 1000)
            {
                running = false;
            }
        }
    }
}

static void charB()
{
    int contB = 1;
    while (running)
    {
        lock (l)
        {
            Console.ForegroundColor = ConsoleColor.Red;
            Console.Write($" B:{contB}");
            contB++;
            if (contB > 1000)
            {
                running = false;
            }
        }
    }
}
```

Como ves ahora para acabar se usar una variable boleeana común.

¿Qué utilidad consideras que tiene el lock? Prueba a quitarlo.

Bueno, si lo ejecutas en principio parece que funciona pero realmente en cuanto acaba uno de los hilos, el otro aún da una vuelta más. Esto es porque está esperando en el lock y ya se ha pasado el while. Entonces cuando se libera el lock da una vuelta más.

Esto es por usar la booleana común (recurso común) fuera de lock, pero no podemos meter el bucle dentro del lock (prueba a hacerlo). Por tanto la solución debe tener esta forma:

```
static void charA()
{
    int contA = 1;
    while (running)
    {
        lock (l)
        {
            if (running)
            {
                Console.ForegroundColor = ConsoleColor.Green;
                Console.Write($" A:{contA}");
                contA++;
                if (contA > 1000)
                {
                    running = false;
                }
            }
        }
    }
}

static void charB()
{
    int contB = 1;
    while (running)
    {
        lock (l)
        {
            if (running)
            {
                Console.ForegroundColor = ConsoleColor.Red;
                Console.Write($" B:{contB}");
                contB++;
                if (contB > 1000)
                {
                    running = false;
                }
            }
        }
    }
}
```

Es decir, se valora el recurso común (en este caso la booleana) otra vez dentro del lock, de forma que la última vuelta, el hilo que es´ta esperando ya no ejecuta lo que hay en su interior.

Ojo, esto no hay que usarlo siempre, pero sí si se desea que en cuanto un hilo acabe otros también.

- Uso del **sleep normalmente debe ir fuera del *lock***. El uso interno es cuando se quiere simular trabajo bloqueante, si no se pone fuera.

- Ten cuidado también a meter **todo un bucle en un lock**, pues puede bloquear el proceso si el bucle dura mucho.

```
// Bloqueo prolongado. EVITAR.
lock (l)
{
	for (int i = 0; i < 100000; i++)
	{
		// Procesamiento intensivo
	}
}
```

- **No usar *lock* anidados** del mismo testigo, no tiene sentido.

- En casos un poco más complejos se debe tener cuidado con el uso de varios testigos, ya que si un hilo tiene un testigo A y otro tiene un segundo testigo B pero el primer hilo necesita B y el segundo A puede aparecer un **deadlock**.

```
static void Hilo1()
{
	lock (recursoA)
	{
		lock (recursoB) // Espera si Hilo2 tiene recursoB
		{
		// ...
		}
	}
}

static void Hilo2()
{
	lock (recursoB)
	{
		lock (recursoA) // Espera si Hilo1 tiene recursoA
		{
		// ...
		}
	}
}
```

Prueba a meter un lock(k) en el primer hilo y lock(l) tras el SetCursorPosition. En el segundo hilo al revés (lock(l) a todo el cuerpo del bucle y lock(k) solo después del SetCursorPosition.

Esperar a otro hilo con Join

Se puede hacer que un hilo espere a que otro acabe. Para eso se usa la función **Join**. Introduce el **Join** después del **Start** de la siguiente forma:

```
(…)
thread.Start();
thread.Join();
for (int i = 1; i < 100; i++){
      (…)
```

En este caso el contaje del programa principal no comienza hasta que acaba el secundario.

Se puede establecer un **Timeout** para la espera de tiempo máxima de un hilo.

```
if (hiloSecundario.Join(1000)) // Espera máximo 1 segundo
{
      Console.WriteLine("Hilo terminado a tiempo");
}
else
{
      Console.WriteLine("Timeout - el hilo sigue ejecutándose, yo continuo, no espero más.");
}
```

Por supuesto se puede esperar a varios hilos con varios Joins.

```
Thread hilo1 = new Thread(Trabajo1);
Thread hilo2 = new Thread(Trabajo2);

hilo1.Start();
hilo2.Start();

hilo1.Join(); // Espera al primer hilo
hilo2.Join(); // Espera al segundo hilo

Console.WriteLine("Ambos hilos han terminado");
```

Hilos en background y Foreground

En .NET existen dos tipos de hilos que determinan el comportamiento al finalizar la aplicación:

Foreground (Primer Plano)

- La aplicación NO termina hasta que todos los hilos foreground finalicen.
- Comportamiento por defecto al crear un Thread.
- Garantizan que el trabajo se complete.

Background (Segundo Plano / Demonio)

- Se finalizan automáticamente cuando todos los hilos foreground terminan
- No evitan el cierre de la aplicación
- Ideales para trabajos auxiliares no críticos

Se puede probar con este ejemplo en el que el hilo secundario a pesar de ser de mayor prioridad se cierra en cuanto termina el primer hilo:

```
static object l = new object();
static void Main(string[] args) // Hilo principal, foreground por defecto.
{
    Thread thread = new Thread(writeDown);
    thread.IsBackground = true; // Punto clave.  Prueba a cambiarlo a false.
    thread.Start();

    for (int i = 1; i < 50; i++)
    {
        lock (l)
        {
            Console.SetCursorPosition(1, 1);
            Console.Write("{0,4}", i);
        }
        Thread.Sleep(50);
    }
} // Cuando acaba el programa se cierra interrumpiendo el hilo background.

static void writeDown()
{
    for (int i = 1; i < 50; i++)
    {
        lock (l)
        {
            Console.SetCursorPosition(1, 20);
            Console.Write("{0,4}", i);
        }
        Thread.Sleep(200);
    }
}
```

Paso de parámetros a un hilo

Hasta el momento hemos tratado los hilos como fragmentos de código que tienen sus propios datos (variables locales) o que pueden tener datos compartidos (propiedades de clase o instancia) a la hora de realizar sus tareas.

Pero a fin de cuentas un hilo se ejecuta a partir de una función y algo inherente a las funciones es la posibilidad de pasarle parámetros.

Para ejecutar un hilo se puede usar una sobrecarga de la función Start a la cual se le puede pasar **un único parámetro**. Este parámetro se le pasará a su vez a la función del hilo. La condición de parámetro único no limita pues es de **tipo object**, y por tanto se puede pasar cualquier cosa.

La diferencia es que cuando llamo a *Start,* entonces le paso el parámetro (es una sobrecarga). Veamos un ejemplo:

```
static void Main(string[] args)
{
    Thread[] hilos = new Thread[10];
    for (int i = 0; i < 10; i++)
    {
        hilos[i] = new Thread(writeALotOfChars);
        char c = (char)('A' + i);
        hilos[i].Start(c);
    }
    for (int i = 1; i < 1000; i++)
    {
        Console.Write("-Main-");
    }
    Console.ReadKey();
}

static void writeALotOfChars(object a)
{
    for (int i = 1; i < 1000; i++)
    {
        Console.Write((char)a);
    }
}
```

Este ejemplo es similar al del principio del tema pero lanzando una ristra de 10 hilos y cada uno de ellos mostrando una letra en pantalla. Se puede observar como divide el sistema el tiempo de cpu entre cada uno de los procesos. Da la impresión de no seguir un patrón y efectivamente no podemos fiarnos de cuanto tiempo dispone un subproceso.

Gestionando las prioridades se puede hacer que unos hilos acaben antes que otros.

En el caso de querer **pasar varios parámetros** podría encapsularlos en un único objeto hecho a medida o en un array o colección.

También es posible usar expresiones lambda para simular el paso de uno o varios parámetros a una función sobre todo si esta no es de tipo object. Veamos un ejemplo:

```
static void addition(int a, int b)
{
    Console.WriteLine("* Thread {0} thinking: ", Thread.CurrentThread.Name);
    for (int i = 0; i < 18; i++) //Working simulation loop
    {
        Thread.Sleep(100);
        Console.Write("*");
    }
    Console.WriteLine("\nResult {0}: {1}", Thread.CurrentThread.Name, a + b);
}

static void factorial(int n)
{
    long result = 1;
    Console.WriteLine(". Thread {0} thinking: ", Thread.CurrentThread.Name);
    for (int i = 2; i <= n; i++)
    {
        Console.Write('.');
        Thread.Sleep(100);  //Working simulation Sleep
        result *= i;
    }
    Console.WriteLine("\nResult {0}: {1}", Thread.CurrentThread.Name, result);
}

static void factorialWrapper(object number)
{  // Need the wrapper to have an object param
    if (number is int)
    {
        factorial((int)number);
    }
}

static void Main(string[] args)
{
    Thread thread1 = new Thread(()=>factorial(20));
    Thread thread2 = new Thread(factorialWrapper);
    Thread thread3 = new Thread(() => addition(44, 23));
    thread1.Name = "Factorial";
    thread2.Name = "FactorialWrapper";
    thread3.Name = "Addition";
    thread1.Start();
    thread2.Start(20);
    thread3.Start();
    Console.ReadKey();
}
```

En ciertas situaciones hay que tener cuidado con las variables usadas como parámetros en las expresiones lambda pues puede haber algún tipo de conflicto que en principio parece extraño, pero que tiene su explicación.

Veamos un ejemplo:

```
for (char i = 'A'; i < 'D'; i++)
 {
     char oneChar = i;

     new Thread(() =>
     {
         for (int j = 0; j < 5; j++)
             Console.Write(oneChar);
     }).Start();

 }
```

Prueba a cambiar el Write(oneChar) por Write(i) y verás un comportamiento extraño.

La explicación es que es necesario usar oneChar porque si uso **i** estoy usando el mismo **i** compartido para todas las funciones por lo que el mismo hilo puede estar mostrando distintas letras al evolucionar **i**. Sin embargo como oneChar se declara dentro del bucle, cada hilo tiene el suyo propio y no hay conflicto.

Para leer más sobre el problema comentado en los comentarios del código echa un ojo al artículo siguiente:

https://blogs.msdn.microsoft.com/ericlippert/2009/11/12/closing-over-the-loop-variable-considered-harmful/

Sincronizacion con Task, async y await devolviendo valor.

Un caso que no hemos visto hasta el momento es la posibilidad de esperar a una función que devuelva un valor. Sí que disponemos de una forma sencilla de espera con Join (o puedes ver una algo más compleja con Wait y Pulse en el apéndice II). Pero ¿y si queremos recibir un resultado devuelto por una función pero esta puede tardar y mientras queremos seguir haciendo cosas?

Es muy típico sobre todo en conexiones a red, bases de datos, etc. que mientras esperamos a que nos manden algo se podría seguir haciendo otras tareas o simplemente que el IU siga respondiendo.

Esto se puede realizar a través de la clase Task que, aunque no es de tan bajo nivel como Thread, es de gran utilidad y hay similares en muchos lenguajes.

Veamos un ejemplo. Crea una aplicación Windows Forms y en el formulario coloca 2 botones (uno denominado **btnColor** y otro **btnSearch**) y una etiqueta.

En el **Click** de **btnColor** simplemente añade estas lineas:

```
Random g = new Random();
this.BackColor = Color.FromArgb(255, g.Next(256), g.Next(256), g.Next(256));
```

Añade la función SlowFind que encuentra el índice de un valor en un vector. Está realizada a propósito de forma muy poco eficiente para que tarde bastante en realizar la tarea y se perciban bien los tiempos.

```
public int SlowFind(int num, int[] vector)
{
    Random g = new Random();
    int position;

    do
    {
        position = g.Next(vector.Length); // Random search :-/
    } while (num != vector[position]);

    return position;
}
```

Una segunda función (**Init**) va a buscar la posición del 1 en un vector muy grande donde el resto de los elementos son 0. Ejecútala en el C**lick** de **btnSearch**.

```
public void Init()
{
     Random g = new Random();
    int[] v = new int[100000000]; // Ojo, esto ocupa 400MB
    v[g.Next(v.Length)] = 1;

    Stopwatch time = new Stopwatch(); // Timing control, no necesario.
    time.Start();

    int result = SlowFind(1, v);

    time.Stop();

    Debug.WriteLine($"Time Elapsed: {time.ElapsedMilliseconds} ms.");
    Debug.WriteLine("I couldn't do anything during the search :-(");
    label1.Text=$"The 1 is in the position {result} of the vector.";
}
```

Si lo pruebas verás que puede tardar varios segundos (depende del ordenador y de la "suerte". Y durante ese tiempo no se puede hacer nada. No funciona el otro botón (de hecho la aplicación se queda como colgada).

Por tanto nos interesa llamar a dicha función, pero mientras tanto poder seguir realizando otras cosas que no requieran el valor devuelto (por ejemplo cambiar colores).

Para ello lo primero es convertir la función en una tarea que permite devolver un valor, en este caso entero, mediante la clase Task:

```
public async void Init()  // async → "Puedo usar await dentro"
{
    // Cada hilo crea su propia instancia de Random
    // Evita problemas de thread-safety sin necesidad de locks
    Random g = new Random();

    // Código síncrono normal
    int[] v = new int[100000000];
    v[g.Next(v.Length)] = 1;

    // Operación asíncrona - NO BLOQUEA
    Task<int> task = Task.Run(() => SlowFind(1, v));

    // Mientras se hace la operación previa, la UI responde.
    Debug.WriteLine("Doing things while waiting for the result. :-D");

    // await → "espera sin bloquear, luego continúa aquí".
    int result = await task;

    label1.Text = $"The 1 is in the position {result} of the vector.";
}
```

La llamada a Run de la clase Task encapsula la función lenta para poder ejecutarla de forma asíncrona (es decir, como hilo). Se usa una **lambda para cumplir el delegado Func<T>.**

Cuando hacemos:

```
Task.Run(() => SlowFind(num, vector));
```

Task.Run, a diferencia de new Thread() que crea un hilo dedicado, encola el trabajo en el **ThreadPool** de .NET. El ThreadPool mantiene un conjunto de hilos pre-instanciados que se reutilizan para múltiples tareas (denominados hilos worker o worker threads), optimizando así la gestión de recursos al evitar el costo constante de crear y destruir hilos. Tras la creación también lo ejecuta.

Existen otras diferencias con Thread además del valor devuelto en cuanto a optimización de recursos que por el momento no veremos.

Cambiamos la función Init de forma que comience por la clausula **async**. La espera se realiza con **await.**

La palabra clave async le dice al compilador: 'Este método va a esperar operaciones largas de forma no bloqueante usando await'. No hace el trabajo asíncrono por sí misma, solo permite usar await dentro del método.

No se puede hacer un await si la función no tiene async.

Un ultimo consejo en referencia a este ejemplo: **no tengas un único generador de números aleatorios**, ya que la clase Random no es segura ante hilos y no debe ser compartida salvo con uso de locks. Pero en este caso (y muchas veces) es más simple tener varios objetos Random, uno por tarea.

Intercambio de datos

Se puede aprovechar esta característica para realizar esperas típicas como la de que llegue alguna información por red, de una base de datos o que cargue un archivo grande, y al mismo tiempo ir realizando otras tareas.

Un ejemplo del primer caso es esperar que llegue una página web pero la aplicación no se bloquee.

Para ello puedes añadir un botón más, dos textbox (la segunda multilínea) y el siguiente código:

```
// Inicializar HttpClient;
private readonly HttpClient _httpClient = new HttpClient();

// Ojo con el async en la función del evento
private async void button3_Click(object sender, EventArgs e)
{
    try // Se puede hacer control de excepciones aquí.
    {
        textBox2.Text = await _httpClient.GetStringAsync(textBox1.Text);
    }
    catch (HttpRequestException ex)
    {
        textBox2.Text = $"Error de conexión: {ex.Message}";
    }
    catch (Exception ex)
    {
        textBox2.Text = $"Error inesperado: {ex.Message}";
    }
}
```

En .net es muy habitual ver **funciones denominadas NombreAsync**, que es una versión Tarea de la misma función sin el Async. Por ejemplo `File.ReadAllTextAsync` es la versión Task (asíncrona) de `File.ReadAllText`. Disponible desde C#7.1 en .Net Core (No en .Net Framework).

Puedes probar a descargar algunos de los JSON siguientes:

https://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/2.5_day.geojson

https://sample.json-format.com/?smp_download=1&key=employees/json/employees_5MB.json

Si ves que se sigue bloqueando un poco, es por la renderización del textbox. Para minimizar este efecto puedes usar un RichTextBox que es más eficiente con textos grandes. También hay soluciones más complejas que no son objetivo de este tema.

Nota: Puedes usar una IA para plantear estas soluciones. Un prompt válido sería:

Estoy realizando pruebas de programación multihilo en C#. En concreto estoy revisando la clase Task y me encuentro con el siguiente problema. En el siguiente código:

```
private readonly HttpClient _httpClient = new HttpClient();


private async void button3_Click(object sender, EventArgs e)
{
   textBox2.Text = await _httpClient.GetStringAsync(textBox1.Text);
}
```

la renderización del texto por parte del TextBox es lenta y bloquea la IU. Un solución es sutituirla por un RichTextBox, pero quisiera que me dieras algunas soluciones utilizando TextBox. Puedes hacerlo con asignación o usando AppendText.

Por supuesto, las soluciones que te plantee la IA debes probarlas porque es probable que algune no funcione correctamente pero siempre te puede ayudar indicándole el problema.

Características que se deben tener en cuenta:

- La sincronización a traves de **Task** es siempre en **Background**. Si el hilo llamante acaba los llamados también.
- El uso de Random debe ser independiente por hilo. Cada hilo con su Random.
- Al revés que en con Thread, **sí se puede hacer control de excepciones** en la llamada a un Task con await.
- Las funciones de sistema que tienen versión asíncrona se les añade la palabra **Async** al final y devuelven un Task<T>. Por ejemplo File.ReadAllTextAsync devuelve un Task<string>.

- Si quieres parar una tarea, por ejemplo para simular carga de trabajo **NO uses Sleep**. Puede causar distintos problemas como bloqueos en el ThreadPool.

  Se recomienda usar, por ejemplo para 1000ms:

  ```
  Task.Delay(1000).Wait();
  ```

  Esto es para una **espera síncrona** clásica, como el sleep. El hilo actual se bloquea. Es como si dijera: *'Congélate 1 segundo, no hagas nada'*

  Si lo que se desea es una **espera asíncrona**, por ejemplo para simular una carga de trabajo, se puede usar await (queda a la espera pero el hilo no bloquea):

  ```
  await Task.Delay(1000);
  ```

  En este caso es como si dijera: *'Vuelve aquí después de 1 segundo, pero mientras tanto sé útil'*

- Al igual que en los Thread, si varias tareas acceden a recurso común debe gestionarse con **lock**.

  En el caso de Task, hay otras alternativas más modernos para el bloqueo como Interlocked, las colecciones concurrentes y SemaphoreSlim, pero el concepto fundamental de protección de recursos compartidos permanece igual, por lo que no las veremos.

Espera de Múltiples Tareas: WhenAll y WhenAny

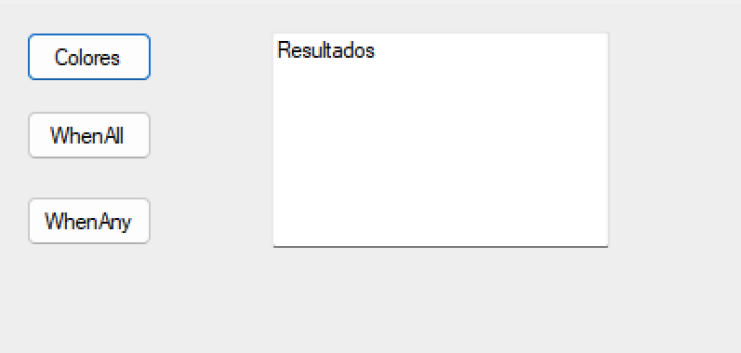
Cuando trabajamos con múltiples tareas en ocasiones es necesario sicronizar la finalización de las mismas. Para ello Task dispone de varias funciones estáticas que me permimten sincronizar esta espera. Veremos las siguientes:

`Task.WhenAll(varias tareas):` Espera a que acaben todas las tareas antes de continuar. Por ejemplo si descargamos distintos datos de distintas URLs para procesar, necesitamos que hayan acabado todos para continuar.

`Task.WhenAny(varias tareas):` Espera a que acabe la tarea más rápida. Por ejmeplo si necesitamos un dato lo más rápido posible podemos pedirlo a distintas fuentes y continuamos cuando se descarga desde la más rápida.

Estas funciones tienen varias sobrecargas de forma que las distintas tareas se les pueden pasar como parámetros (params), como array, como colecciones, etc.

Veamos esto con un ejemplo. Crea una aplicación WindowsForms con tres botones y un textbox multilínea. El aspecto puede ser el siguiente:



Al click de Colores añádele el código que ya teníamos del ejemplo anterior, es decir:

```
Random g = new Random();
this.BackColor = Color.FromArgb(255, g.Next(256), g.Next(256), g.Next(256));
```

Copia las siguientes funciones de generación y búsqueda. Las búsquedas se hacen mediante algoritmos lentos para ver mejor el efecto. Deberías entenderlas sin problemas:

```
public int SlowFind(int num, byte[] vector)
{
    Random g = new Random();
    int position;
    // Busqueda mediante índices aleatorios
    // La velocidad depende de la suerte
    do
    {
        position = g.Next(vector.Length);
    } while (num != vector[position]);

    return position;
}

public static int SlowFindSleep(int num, byte[] vector)
{
    Random g = new Random();
    int position = 0;

    // Búsqueda lineal haciendo pausa (liberando la CPU)
    // en cada vuelta del bucle
    while (position < vector.Length && num != vector[position++])
    {
        Task.Delay(0).Wait();
    }

    return position;
}

// Genera vector con 100000000 de ceros y un 1 en una posición aleatoria
// Se realiza de tipo byte y no int para ocupar 100MB en lugar de 400MB
public byte[] generaVector()
{
    Task.Delay(1).Wait(); // Evita mismo new Random() en sucesivas llamadas.
    Random g = new Random();
    byte[] v = new byte[100000000];
    v[g.Next(v.Length)] = 1;
    return v;
}
```

A continuación programamos el comportamiento del botón **WhenAll**. Para ello hacemos la función **InitAll** de búsqueda del valor 1 en tres vectores distintos y nos dará el resultado cuando acabe de buscar en los tres.

Dicha función será llamada desde el click del botón WhenAll.

Nota: Se ha quitado el control de excepciones a lo largo del código para no alargarlo.

El código queda así:

```
// Función que busca el 1 en todos los vectores
// de forma paralela usando WhenAll para la espera.
public async void InitAll()
{

    txtResultados.Text = $"Buscando en todos los vectores";
    btnAll.Enabled = false;
    byte[] v1 = generaVector();
    byte[] v2 = generaVector();
    byte[] v3 = generaVector();

    Task<int> task1 = Task.Run(() => SlowFind(1, v1));
    Task<int> task2 = Task.Run(() => SlowFind(1, v2));
    Task<int> task3 = Task.Run(() => SlowFind(1, v3));

    // En esta línea se libera el bloqueo para hacer otras tareas
    // y solo se continua cuando han acanbado las 3 tareas.
    int[] tiempos = await Task.WhenAll(task1, task2, task3);

    btnAll.Enabled = true;
    txtResultados.Text = $"Resultados WhenAll{Environment.NewLine}";
    for (int i = 0; i < tiempos.Length; i++)
    {
        txtResultados.AppendText($"Posición en v[{i}]: {tiempos[i]}" +
                                 $"{Environment.NewLine}");
    }
}

//Función asociada al evento Click
private void btnAll_Click(object sender, EventArgs e)
{
    InitAll();
}
```

Pruébalo y deberías comprobar que aunque se estén buscando los índices (y puede tardar un buen rato), la IU sigue funcionando sin ningun problema. Puedes cambiar colores, tamaño del formulario.

El botón se inhabilita para evitar que se le de varias veces durante la búsqueda.

Finalmente vemos el código del botón WhenAny:

```
// Función que busca la posición del 1
// de un vector mediante el método más rápido
public async Task<string> InitAny()
{
    txtResultados.Text = $"Buscando el método maás rápido";
    Random g = new Random();

    byte[] v = generaVector(); ;

    Task<string> task1 = Task.Run(() => $"SlowFind: {SlowFind(1, v)}");
    Task<string> task2 = Task.Run(() => $"SlowFindSleep: {SlowFindSleep(1, v)}");

    // Si a IndexOf le pasas 1 sin casting busca un integer que no existe.
    Task<string> task3 = Task.Run(() => $"IndexOf: {Array.IndexOf(v, (byte)1)}");

    // Hay doble await porque WhenAny es del tipo Task<Task<string>>
    // - Task interior: representa "la tarea que terminó primero"
    // - Task exterior: representa "El dato que contiene esa tarea"
    Task<string> tareaFinal = await Task.WhenAny(task3, task2, task1);
    return await tareaFinal;
}

private async void btnAny_Click(object sender, EventArgs e)
{
    string ganador = await InitAny();
    txtResultados.Text = $"Resultados WhenAny{Environment.NewLine}{ganador}";
}
```

Este puede parecer un poco más complejo porque se hacen dos await, pero el motivo es que WhenAny devuelve Task<Task<T>> por como está construido internamente, entonces el primer await realmente es el que espera por la tarea y el segundo es un requisito sintáctico del lenguaje C#. Aunque sabemos que la tarea ya está completada y no hay espera real, el compilador exige usar await para convertir Task<T> en T.

El proceso puedes visualizarlo en este esquema:

```
Task.WhenAny(task1, task2, task3)
     ↓ (retorna)
Task<Task<int>>  ← "Promesa de una tarea"
     ↓ (await - espera a que alguna termine)
Task<int>        ← ¡La tarea que ganó! (task1 por ejemplo)
     ↓ (await - obtiene el resultado de ESA tarea, no hay espera real)
string           ← “SlowFind: 22” (resultado final ejemplo)
```

Como nota final, indicar que cuando se usan funciones de sistema con nombre acabado en **Async** (y alguna otra), estas ya devuelven una tarea y **no es necesario ejecutarlas con Task.Run()**. Revisa siempre lo que devuelven.

Veamos dos ejemplos:

Ejemplo 1: Espera a que acaben varias tareas (ejemplo incompleto y no funcional en .Net Framework pues no existe ReadAllTextAsync):

```
// Operaciones que YA tienen versión async
// ¡Sin Run!, ya devuelve Task<T>
var t1 = File.ReadAllTextAsync("archivo1.txt");  // Lee de archivo
var t2 = _httpClient.GetStringAsync(url);        // Lee URL
var t3 = _dbContext.Users.ToListAsync();         // Lee de BD

await Task.WhenAll(t1, t2, t3);
```

Ejemplo 2: TimeOut con WhenAny

```
public async Task<string> DescargarConTimeout(string url, int timeoutMs)
{
    var httpClient = new HttpClient();
    var tareaDescarga = httpClient.GetStringAsync(url); // Ya es Task.
    var tareaTimeout = Task.Delay(timeoutMs);           // Ya es Task.

    // Esperar a que termine la descarga O el timeout
    var tareaCompletada = await Task.WhenAny(tareaDescarga, tareaTimeout);

    if (tareaCompletada == tareaTimeout)
    {
        throw new TimeoutException("La descarga tardó demasiado");
    }

    return await tareaDescarga; // Ya está completada
}
```

Cuestiones pendientes y bibliografía

El tema multiproceso/multihilo es realmente muy amplio, de hecho existen libros completos que hablan exclusivamente de esta multiprogramación. Nosotros hemos hecho una introducción a este mundo pero quedan muchos elementos por estudiar que por falta de tiempo no vamos a profundizar en ellos a menos que nos sea de interés en otros temas.

Algunos de estos campos son el uso de Wait y Pulse (en apéndice) o los EventHandlers para tratar señalización, las AppDomains que mejoran el uso de memoria y cpu por parte de los procesos, la sincronización con delegados, thread pooling, la clase Task y un largo etc.

Como documento interesante sobre este tema y que puede ayudar a completarlo en gran medida está el siguiente recurso web que ha sido usado como base para algunos de los puntos aquí vistos:

http://www.albahari.com/threading/

En dicha Web te puedes bajar el documento Threading C# en PDF.

También se ha usado para este documento la bibliografía siguiente donde se puede completar conocimientos:

Concurrent programming on Windows. Joe Duffy. Addison-Wesley 2009.
Pro C# 2008 and de .net 3.5 platform. Andrew Troelsen. Apress 2007.
Accelerated C# 2010. Trey Nash. Apress 2010.
C# for Java programmers. Bagnall, Chen, Goldberg. Syngress 2002.
Professional C# 4 and .Net 4. Nagel, Evjen, Glynn, Watson, Skinner. Wrox 2010.
Core C# and .Net. Stephen C. Perry. Prentice Hall 2005.
Peer-to-peer with VB.NET. Mattew MacDonald. Apress 2003.

Pero cualquier otro libro completo sobre C# seguramente traiga algún capítulo sobre Threading con el que se pueda completar conocimiento.

Apéndice I: Introducción a los hilos en Java

El concepto de hilos va más allá del lenguaje de programación y en diversos lenguajes existen formas de realizar múltiples hilos. Con el objeto de ampliar conocimiento, lo que hemos visto en c# se puede hacer de forma muy similar en Java y para muestra veremos como crear varios hilos y realizar una sincronización básica de forma que puedan llevarse estos conocimientos a otras materias como programación en Android.

La principal diferencia es que en Java no tenemos delegados y la forma en la que se trata de hacer hilos es heredando la clase Thread la cual implementa la Interfaz Runnable que obliga a implementar el método run(). Es precisamente en ese método donde meteremos el código que nos interese ejecutar como hilo. En el siguiente esquema se ven otros métodos de la clase Thread que pueden ser de interés:

A continuación un ejemplo simple similar al primero visto en C#.

```
class HiloLetraA extends Thread {

    @Override
    public void run(){
        for (int i=0; i<1000; i++)
            System.out.print("A");
    }
}

public class Hilos {
    public static void main(String[] args) {
         HiloLetraA hiloA= new HiloLetraA();
         hiloA.start();
         for (int i=0; i<1000; i++)
            System.out.print("B");

    }
}
```

Se puede ver por tanto que para crear el hilo es necesario crear una clase aparte y dentro de esa clase se establece al menos el método run() que será el que se ejecute a realizar el start. Se pueden por supuesto añadir propiedades a la clase para disponer de distintos datos con los que trabajará el hilo.

Puedes ver los estados de los hilos en Java en:

https://www.codejava.net/java-core/concurrency/understanding-thread-states-thread-life-cycle-in-java

Apéndice II: Señalización entre procesos con Wait y Pulse

En ocasiones lo que realiza un proceso puede depender de lo que hagan otros procesos. Para ello tiene que existir la posibilidad de que se avisen entre ellos. Una primera forma puede ser usando los métodos estáticos **Wait** y **Pulse** de la clase **Monitor** junto con el ya conocido **lock**.

Mediante **Wait** se pausa un hilo, entra en una cola de espera y **queda liberado el lock** dentro del cual esté, por lo que debe usarse con cuidado. Nosotros lo que haremos es un lock exclusivo para realizar la espera.

Mediante **Pulse** se avisa de que ya puede continuar el hilo en espera usando el mismo objeto de bloqueo.

Teniendo esto en cuenta hay que resaltar que **Wait debe ejecutarse en el propio hilo que se desee pausar** (normalmente dependiendo de alguna condición con if).

Sin embargo **Pulse debe ejecutarse desde otro hilo** para liberar el pausado (si se pone Pulse en el hilo en espera nunca se ejecuta y, por tanto, nunca se libera).

Se puede hacer mediante estos elementos que un hilo espere hasta que otro haya hecho parte de su tarea.

Por ejemplo, supongamos que el hilo writeDown interesa que se ejecute pero solo cuando el principal ha superado 15 en el contaje. En este caso se lanzan los dos hilos, pero *writeDown* queda en espera por causa del Wait. En el momento que el Main lanza el Pulse, este hace que continue writeDown:

```
static readonly object l = new object();
static void Main(string[] args)
{
    Thread thread = new Thread(writeDown);
    thread.Start();

    for (int i = 1; i <= 30; i++)
    {
        lock (l)
        {
            Console.SetCursorPosition(1, 1);
            Console.Write("{0,4}", i);
        }
        Thread.Sleep(50);
        if (i == 15)  // Warn thread writeDown to begin
        {
            lock (l)
            {
                Monitor.Pulse(l);
```

```
                }
            }
        }
        Console.ReadKey();
    }


    static void writeDown()
    {
        lock (l) //  Wait until be warned
        {
            Monitor.Wait(l);
        }



        for (int i = 1; i <= 20; i++)
        {
            Thread.Sleep(50);
            lock (l)
            {
                Console.SetCursorPosition(1, 20);
                Console.Write("{0,4}", i);
            }
        }
    }
```

Este esquema es muy sencillo pero tiene un problema y es **¿qué pasa si se ejecuta antes el Pulse que el Wait?**. Veámoslo con el siguiente ejemplo:

```
        static readonly object l = new object();
        static void Main(string[] args)
        {
            Thread thread = new Thread(action);
            thread.Start();
             Console.WriteLine("Main continues.");
            lock (l)
            {
                Monitor.Pulse(l);
            }
            Console.ReadKey();
        }

        static void action()
        {
            Console.WriteLine("action begins.");
            lock (l)
            {
                Monitor.Wait(l);
             }
            Console.WriteLine("Don't rest anymore...");
        }
```

Si ejecutas este programa pueden ocurrir dos cosas: Que primero se ejecute el Wait con lo que todo va con normalidad y en cuanto se ejecuta el *Pulse* termine el hilo

action. O que se ejecute el *Pulse* antes lo que provocará que el hilo nunca salga de la espera y tenemos un problema (Si fuera necesario mete un pequeño sleep en *action* antes del lock para forzar esta situación).

Como solución se recomienda seguir un esquema estándar como el siguiente en el que nos apoyamos en un flag:

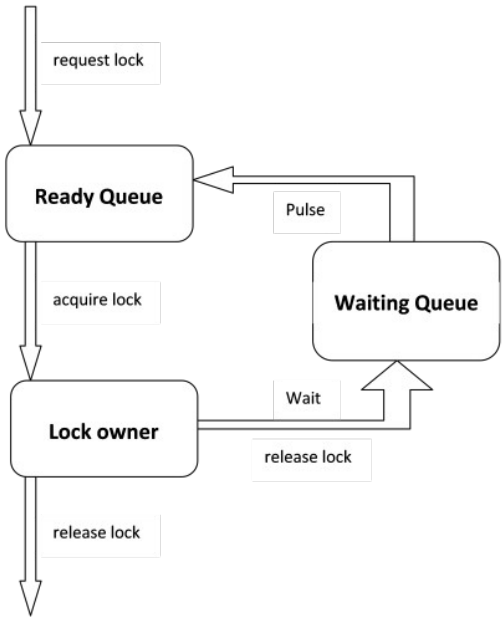
```
        static readonly object l = new object();
        static bool finished;

        static void Main(string[] args)
        {
            finished = false;
            Thread thread = new Thread(action);
            thread.Start();
        // En esta parte iría código que produce resultados que necesitan otros hilos y
que deben esperar a que acabe para recogerlo.
            lock (l)
            {
                finished = true;
                Monitor.Pulse(l);  // or Monitor.PulseAll(l)
            }
            Console.ReadKey();
        }

        static void action()
        {
            lock (l)
            {
                while (!finished) //  'if' works well too in this program
                {
                    Monitor.Wait(l);
                }
                finished = false;
            }

        // En esta parte iría el código que usa los resultados por los que está
esperando. Se asegura que esos resultados son correctos cuando la booleana es false.
            Console.WriteLine("Don't rest anymore...");
         }
```

En este caso, si se ha ejecutado primero el Pulse, en cuanto se ejecute el hilo *action* no hay problema con el *Wait* ya que nunca se ejecuta.



Esto es muy importante ya que **en una gran cantidad de ocasiones necesitarás aplicar el uso de booleanas para sincronizar hilos.**

Se podría usar un if en lugar de un while, y de hecho en muchos de las aplicaciones que hagamos llegaría (como en esta), pero este caso permite solucionar el siguiente problema: Cuando el hilo action recibe el Wait, se va a la cola de espera (Waiting queue) del lock. Cuando se recibe un Pulse, no coge directamente el testigo si no que pasa a la cola de Preparado (Ready queue) pero si otro hilo ya estaba en ella antes y coge antes el lock, puede ser que cambie de nuevo el estado de la variable finished lo que sería un problema.

Con el while, tras hacer el Wait y aunque le llegue un Pulse, **vuelve a comprobar la variable finished** por si otro hilo la ha vuelto a cambiar. Por esto se suele usar siempre este esquema.

Es decir, el hilo que tiene el Wait está esperando que terminen ciertas tareas de otro hilo para asegurar que el contenido resultado al que tiene que acceder es correcto. Quién asegura eso es la variable finished que está a false. Si tras salir del wait otro hilo ha entrado antes y cambia los resultados, cambiará también el

finished y por eso es necesario volver a comprobarlo.

Es importante fijarse que los flags usados (finished en este caso) se deben cambiar dentro del lock pues son zonas de memoria compartidas por varios hilos.

En el ejemplo anterior, el `finished = false;` tampoco hace falta, pero en un esquema genérico sí, para dar permiso a otros hilos que estén intentando acceder al mismo sitio.

Puedes ver más información en:
http://www.codeproject.com/Articles/28785/Thread-synchronization-Wait-and-Pulse-demystified

Existe la función **PulseAll** que es muy cómoda cuando se desean que todos los hilos que están en Wait en un testigo sean sacados de dicha espera en lugar de ir Pulse a Pulse con cada uno.

Apéndice III: Otros métodos de bloqueo

El tema de bloqueo y sincronización es realmente amplio y no lo vamos a abarcar al completo, sin embargo cabe citar algunos métodos y clases que pueden ser de interés para futuras aplicaciones del alumno:

- El tipo *System.Threading.Monitor* además de Pulse y Wait da otras posibilidades.

- *System.Threading.Interlocked* dispone de una serie de funciones estáticas que realizan operaciones aritméticas y lógicas de forma atómica (como si la envolvieramos en un lock) ya que al compilar de C# a IL (Intermediate Language o código máquina del CLR) no tiene porque producir acciones atómicas.

- Se puede sincronizar (hacer un lock) a toda una clase mediante el atributo [Syncronize] pero no lo vamos a ver además de ser un método "poco eficiente" si no se usa bien.

- El *Mutex* es como el *lock* pero para procesos en lugar de para hilos. Es muy útil para que sólo haya una instancia de un programa en ejecución. Se puede ver un ejemplo en http://www.albahari.com/threading/part2.aspx#_Mutex

- Semáforos: Permiten limitar el número de veces que se está ejecutando cierto código en distintos threads. Útil para no sobrecargar ciertos recursos como el disco duro u otro. Ver el ejemplo en: http://www.albahari.com/threading/part2.aspx#_Semaphore

- Sincronización mediante wait y pulse. Explicado en el apéndice II.

Apéndice IV: Trabajo con procesos

La clase Process

Esta clase se encuentra dentro del espacio de nombres *System.Diagnostics* y permite el acceso a procesos de una máquina determinada así como la posibilidad de lanzar o finalizar procesos. Veamos un par de ejemplos:

```
Process p;
p = Process.Start("Notepad");
Console.WriteLine("Name: {0}\nPID: {1}\nSubprocesses: {2}\nInit: {3}",
                        p.ProcessName, p.Id, p.Threads.Count, p.StartTime);
p.WaitForExit();
Console.WriteLine("Aplication finished in date and time {0}", p.ExitTime);
```

El código anterior lanza el Notepad mediante **Start***()* y nos da información sobre el proceso mediante las propiedades del objeto *Process.* El método **WaitForExit***()* es de sincronización y deja el programa parado hasta que el proceso lanzado termine.

Es probable que el número de subprocesos sea 1. Esto es lógico pues es el momento de ejecución de la aplicación, si esperamos un poco antes de sacar la información es posible que dependiendo de la aplicación, aparezca algún subproceso más.

Para probar esto incluye el espacio de nombres System.Threading y añade la siguiente línea **justo después del Process.Start()***:*

```
Thread.Sleep(5000);
```

Al ejecutarlo de nuevo verás que tarda 5 segundo en aparecer la información *(Sleep* hace que el hilo actual se paralice durante los milisegundos indicados) y ya aparecen más subprocesos sobre todo **si intentas abrir un archivo o realizas alguna tarea.**

Es posible pasarle argumentos a la aplicación que queremos abrir mediante un objeto **ProcessStartInfo**. Por ejemplo si queremos abrir el Notepad con un archivo en concreto haríamos algo así:

```
Process p;
string file;
Console.WriteLine("Input file name to open");
file = Console.ReadLine();
ProcessStartInfo startInfo = new ProcessStartInfo("Notepad", file);
p = Process.Start(startInfo);
```

Creamos un objeto *startInfo* con el nombre de la aplicación y los argumentos a pasarle y luego ejecutamos *Start* pero en lugar de pasarle el nombre de la aplicación le pasamos el objeto *startInfo* anterior.

Nota: Para ver cómo pasarle argumentos a una aplicación desde el sistema operativo y como recogerlos luego puedes leer el Apéndice II del Tema 3 (Otros aspectos de C#) del módulo Desarrollo de Interfaces (DI).

Veamos ahora un ejemplo para ver todos los procesos en ejecución:

```
Process[] processes = Process.GetProcesses();
const string FORMAT = "{0,20}{1,7}{2,6}";
Console.WriteLine(FORMAT, "Name", "PID", "Threads");
foreach (Process p in processes)
{
    Console.WriteLine(FORMAT, p.ProcessName, p.Id, p.Threads.Count);
}
```

Se utiliza el método estático *GetProcesses* que devuelve un array de objetos *Process* a partir del cual podemos obtener la información o incluso podríamos parar (forzando el cierre inmediato) uno de ellos mediante el método *Kill()* si fuera de interés.

Algunos métodos de **Process**

Kill(): Fuerza el cierre inmediato del proceso

CloseMainWindow(): En caso de que sea una aplicación lo que queremos cerrar y queremos hacerlo de forma natural (pasando por el evento FormClosing en Formularios) usaremos este método que envía un mensaje Close a la ventana principal

GetProcessById(int) : Método estático que devuelve el proceso representado por su PID. Salta una excepción ArgumentException si no existe el PID.

Modules*:* Colección del tipo ProcessModuleCollection que contiene una lista de objetos tipo ProcessModule cada uno de los cuales representa un módulo que utiliza el programa. Un módulo puede ser un archivo .exe o .dll. Es similar a ejecutar en consola un **tasklist /m.**

Threads*:* Colección del tipo ProcessThreadCollection que devuelve la lista de hilos del proceso en cuestión. Cada hilo dentro de la colección es del tipo *ProcessThread* lo que permite a su vez obtener datos de cada hilo por separado. Un ejemplo de

cómo se recorrería esta colección:

```
ProcessThreadCollection threads = p.Threads;
foreach (ProcessThread t in threads)
{
	Console.WriteLine("Thread ID: {0}\tInit {1}\tPriority {2}\tState {3}",
			t.Id, t.StartTime.ToShortTimeString(), t.PriorityLevel,
			t.ThreadState);
}
```

En algún momento salta una excepción por problema de permisos, puede usarse un try/catch para saltarse el acceso al proceso que requiere dichos permisos. Sin embargo verás que de muchos procesos no sale información. Esto es por dos motivos: Permisos y arquitectura.

Para solucionar la parte de permisos debes ejecutar la aplicación con permisos de administrador, para ello hay que **agregar un nuevo elemento al proyecto** con el botón derecho del ratón y seleccionar: "**Archivo de manifiesto de aplicación**". En .NET6 solo se admite para Windows.

En este archivo XML se permite, entre otras cosas, establecer con que permisos se ejecuta el programa. Tenemos que cambiarlos de `asInvoker` a `requireAdministrator`. Por tanto la línea afectada quedará de la siguiente forma:

```
<requestedExecutionLevel level="requireAdministrator" uiAccess="false" />
```

Aún así será necesario el control de excepciones si intentamos acceder a ciertos procesos del sistema operativo.

Como curiosidad el atributo uiAccess sirve para que la aplicación pueda enviar mensajes a otras aplicaciones, pero para poder hacer esto debe estar firmada digitalmente (requiere pago a MS) y estar en algún directorio de confianza (como Program Files) para que funcione.

Respecto a la arquitectura hay que tener en cuenta que si estamos en un sistema de 64 bits y nuestra aplicación compila en 32 bits no es posible accceder a la información de procesos de 64 bits. Para cambiar esto **Project properties → Build → Platform target → x64** (por defecto está en AnyCPU). Cambiando esto se puede acceder a procesos que son del propio usuario (como Chrome) y son de 64 bits.