PSP03.- Programación multihilo

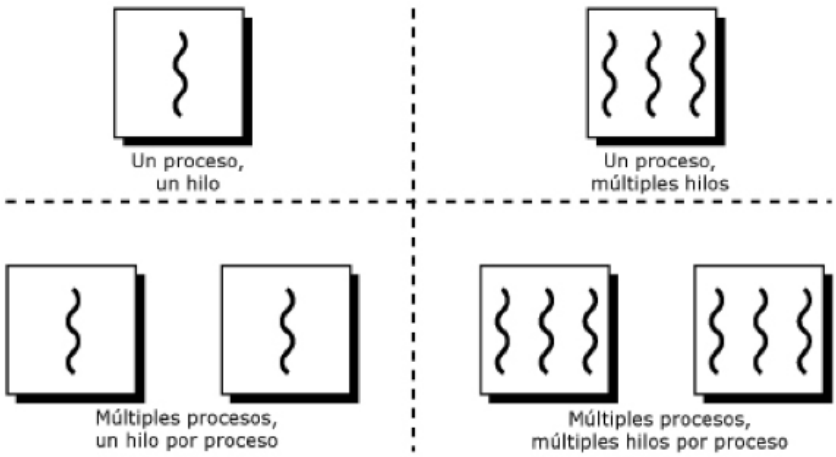
1.- INTRODUCCIÓN A LA PROGRAMACIÓN MULTIHILO.

1.1.- INTRODUCCIÓN.

La idea fundamental es bien sencilla. En la programación tradicional hay un solo flujo de control, motivado fundamentalmente porque la máquina internamente suele tener un solo procesador (una sola "mente" que realiza las instrucciones una tras otra). La **programación multihilo** o multithreading permite la ocurrencia simultánea de varios flujos de control. Cada uno de ellos puede programarse independientemente y realizar un trabajo, distinto, idéntico o complementario, a otros flujos paralelos.

Los hilos son un concepto relativamente nuevo de los sistemas operativos. En este contexto, un proceso recibe el nombre de **proceso pesado**, mientras que un hilo recibe el nombre de **proceso ligero**. El término hilo (en inglés, thread) se refiere sintáctica y semánticamente a hilos de ejecución. El término **multihilo** hace referencia a la capacidad de un sistema operativo para mantener varios hilos de ejecución dentro del mismo proceso.

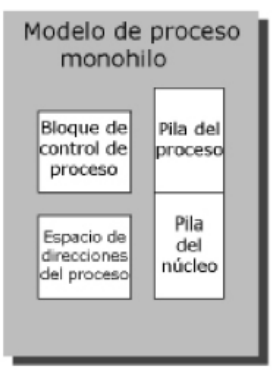
Procesos monohilo y multihilo



Autor *(link: https://www.google.es)* (Licencia) *(link: https://www.google.es)* Procedencia *(link: https://www.google.es)*

En un sistema operativo con **procesos monohilo** (un solo hilo de ejecución por proceso), en el que no existe el concepto de hilo, la representación de un proceso incluye su BCP, un espacio de direcciones del proceso, una pila de proceso y una pila del núcleo.

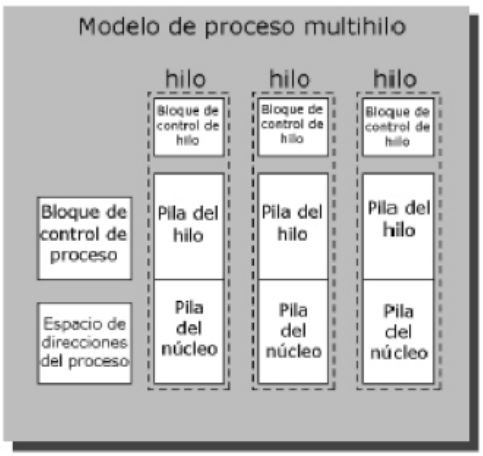
Modelo de proceso monohilo



Autor *(link: https://www.google.es)* (Licencia) *(link: https://www.google.es)* Procedencia *(link: https://www.google.es)*

En un sistema operativo con **procesos multihilo** hay un único BCP y espacio de direcciones para el proceso; sin embargo, existen pilas de hilo, pilas del núcleo y bloques de control separados para cada hilo.

Modelo de proceso multihilo



Autor *(link: https://www.google.es)* (Licencia) *(link: https://www.google.es)* Procedencia *(link: https://www.google.es)*

Un **hilo (proceso ligero)** es una unidad básica de utilización de la CPU, y consiste en un contador de programa, un juego de registros y un espacio de pila. Los hilos dentro de una misma aplicación comparten:

- La sección de código.
- La sección de datos.
- Los recursos del sistema operativo (archivos abiertos y señales).

Un **proceso** tradicional o **proceso pesado** es igual a una tarea con un solo hilo. Los hilos permiten la ejecución concurrente de varias secuencias de instrucciones asociadas a diferentes funciones dentro de un mismo proceso, compartiendo un mismo espacio de direcciones y las mismas estructuras de datos del núcleo.

1.2.- ESTADOS DE UN HILO.

Los principales estados de un hilo son: **ejecución, preparado y bloqueado.** Además, existen 4 operaciones básicas relacionadas con el cambio de estado de los hilos:

- **Creación:** en general, cuando se crea un nuevo proceso se crea también un hilo para ese proceso. Posteriormente, ese hilo puede crear nuevos hilos dándoles un puntero de instrucción y algunos argumentos. Ese hilo se colocará en la cola de preparados.
- **Bloqueo:** cuando un hilo debe esperar por un suceso, se le bloquea guardando sus registros. Así el procesador pasará a ejecutar otro hilo preparado. Pasa a la pila de bloqueados.
- **Desbloqueo:** cuando se produce el suceso por el que un hilo se bloqueó, pasa a la cola de preparados.
- **Terminación:** cuando un hilo finaliza, se liberan su contexto y pilas.

Un punto importante es la posibilidad de que el bloqueo de un hilo lleve al bloqueo de todo el proceso al que pertenece; es decir, que el bloqueo de un hilo lleve al bloqueo de todos los hilos que lo componen, aun cuando el proceso esté en estado preparado.

1.3.- RECURSOS COMPARTIDOS Y NO COMPARTIDOS DE LOS HILOS.

Los hilos permiten la ejecución concurrente de varias secuencias de instrucciones asociadas a diferentes funciones dentro de un mismo proceso, compartiendo un mismo espacio de direcciones y las mismas estructuras de datos del núcleo.

Recursos compartidos entre los hilos:

- Código (instrucciones).
- Variables globales.
- Ficheros y dispositivos abiertos.

Recursos no compartidos entre los hilos:

- Contador del programa (cada hilo puede ejecutar una sección distinta de código).
- Registros de CPU.
- Pila para las variables locales de los procedimientos a los que invoca después de crear un hilo.
- Estado: en ejecución, preparado o bloqueado.

2.- PROCESOS E HILOS.

Semejanzas. Los hilos operan en muchos sentidos como los procesos:

- Pueden estar en diferentes estados: preparado, bloqueado y en ejecución.
- Comparten la CPU.
- Sólo hay un hilo activo (en ejecución) en un instante dado.
- Un hilo dentro de un proceso se ejecuta secuencialmente.
- Cada hilo tiene su propia pila y contador de programa.
- Pueden crear sus propios hilos hijos.

Diferencias. Los hilos, a diferencia de los procesos, no son independientes entre sí:

- Como todos los hilos pueden acceder a todas las direcciones de la tarea, un hilo puede leer la pila de cualquier otro hilo o escribir sobre ella. Aunque pueda parecer lo contrario la protección no es necesaria ya que el diseño de una tarea con múltiples hilos tiene que ser de un único usuario.

Ventajas de los hilos sobre los procesos:

- Se tarda menos tiempo en crear un nuevo hilo en un proceso existente que en crear un nuevo proceso.
- Se tarda menos tiempo en terminar un hilo que un proceso.
- Se tarda menos tiempo en conmutar entre hilos de un mismo proceso que entre procesos.
- Los hilos hacen más rápida la comunicación entre procesos, ya que al compartir memoria y recursos, se pueden comunicar entre sí sin invocar al núcleo del sistema operativo.

Ejemplos de uso de los hilos. Hay miles de ejemplos en los que puede ser útil pensar en varios flujos de ejecución (hilos): la posibilidad de editar mientras seguimos cargando o salvando un gran fichero, la posibilidad de visualizar una página mientras se están buscando las siguientes, la visualización de varios procesos que ocurren a la vez de forma independiente, etc. Se pueden clasificar en:

- **Trabajo interactivo y en segundo plano:** en un programa de hoja de cálculo, un hilo podría estar leyendo la entrada del usuario y otro podría estar ejecutando las órdenes y actualizando la información, o mientras escribimos en un procesador de texto el sistema nos va corrigiendo.
- **Procesamiento asíncrono:** se podría implementar, con el fin de protegerse de cortes de energía, un hilo que se encargará de salvaguardar el buffer de un procesador de textos una vez por minuto.
- **Estructuración modular de los programas:** los programas que realizan una variedad de actividades se pueden diseñar e implementar mediante hilos.

3.- 1. CREAR HILOS EN C#.

Para crear hilos en C# se dispone de la Clase Thread. A diferencia de otros lenguajes de programación no es posible heredar de la clase Thread. Por lo tanto, se debe instanciar un Thread y hacer uso del objeto para utilizar un hilo. La clase Thread tiene el constructor sobrecargado y acepta un único parámetro de entraga:

- public delegate **void ThreadStart()**
- public delegate **void ParameterizedThreadStart(object obj)**

Una vez el Thread ha sido instanciado hay que llamar al método Start.

3.1.- LA CLASE THREAD.

Algunas de las características de la **clase Thread** son las siguientes:

Atributos más relevantes:

- Priority
- IsAlive
- ThreadState
- IsBackground
- IsThreadPoolThread

Para saber más

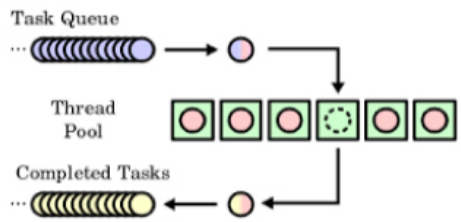
Métodos más relevantes:

Referencia *(link: https://docs.microsoft.com/es-es/dotnet/api/system.threading.thread?view=netframework-4.8#m%C3%A9todos)*

3.2.- LA CLASE TASK.

C# ofrece la clase Task a los programadores para la realización de tareas de forma asíncrona en el Pool de Threads. La clase Task ofrece una interfaz de programación de un nivel más alto que Thread. Al ser tareas asíncronas en caso de necesitar sincronizar algunas tareas entre hay que utilizar el operador await o diferentes métodos de la clase que se explican a más adelante. Otra diferencia con los hilos es que las tareas pueden devolver un resultado directamente sin necesidad de un delegado.

la clase Task



Autor *(link: https://www.google.es)* (Licencia) *(link: https://www.google.es)* Procedencia *(link: https://www.google.es)*

Constructores

Task(Action)	Inicializa una nueva instancia de Task con la acción especificada.
Task(Action, CancellationToken)	Inicializa una nueva instancia de Task con la acción y CancellationTokenespecificados.

Hay mas constructores para su consulta existe **la documentación oficial .**

Propiedades

AsyncState	Obtiene el objeto de estado que se proporcionó al crearse el objeto Task, o null si no se proporcionó ningún objeto de estado.
CompletedTask	Obtiene una tarea que ya ha finalizado correctamente.
CreationOptions	Obtiene el objeto TaskCreationOptions usado para crear esta tarea.

CurrentId	Devuelve el identificador del objeto Task que se está ejecutando actualmente.
Exception	Obtiene la excepción AggregateException que causó la finalización prematura del objeto Task. Si Task se completó correctamente o no ha iniciado ninguna excepción, el valor devuelto será null.
Factory	Proporciona acceso a patrones de diseño para crear y configurar instancias de Task y Task<TResult>.
Id	Obtiene un identificador para esta instancia de Task.
IsCanceled	Obtiene un valor que indica si esta instancia de Task ha dejado de ejecutarse debido a una cancelación.
IsCompleted	Obtiene un valor que indica si la tarea se ha completado.
IsFaulted	Obtiene un valor que indica si el objeto Task se ha completado debido a una excepción no controlada.
Status	Obtiene el objeto TaskStatus de esta tarea.

Métodos

ConfigureAwait(Boolean)	Configura un awaiter utilizado para esperar a este objeto Task.
ContinueWith(Action<Task,Object>, Object)	Crea una continuación que recibe información de estado proporcionada por el autor de la llamada y se ejecuta cuando el elemento Task de destino se completa.
ContinueWith(Action<Task,Object>, Object, CancellationToken)	Crea una continuación que recibe información de estado proporcionada por el autor de la llamada y un token de cancelación y que se ejecuta de forma asincrónica cuando el elemento Task de destino se completa.
ContinueWith(Action<Task,Object>, Object, CancellationToken, TaskContinuationOptions, TaskScheduler)	Crea una continuación que recibe información de estado proporcionada por el autor de la llamada y un token de cancelación y que se ejecuta cuando el elemento Task de destino se completa. La continuación se ejecuta según un conjunto de condiciones especificadas y usa un programador especificado.
ContinueWith(Action<Task,Object>, Object, TaskContinuationOptions)	Crea una continuación que recibe información de estado proporcionada por el autor de la llamada y se ejecuta cuando el elemento Task de destino se completa. La continuación se ejecuta según un conjunto de condiciones especificadas.
ContinueWith(Action<Task,Object>, Object, TaskScheduler)	Crea una continuación que recibe información de estado proporcionada por el autor de la llamada y se ejecuta de forma asincrónica cuando el elemento Task de destino se completa. La continuación usa un programador especificado.
ContinueWith(Action<Task>)	Crea una continuación que se ejecuta de manera asincrónica cuando se completa el objeto Task de destino.
ContinueWith(Action<Task>, CancellationToken)	Crea una continuación que recibe un token de cancelación y se ejecuta de forma asincrónica cuando el elemento Task de destino se completa.
ContinueWith(Action<Task>, CancellationToken, TaskContinuationOptions, TaskScheduler)	Crea una continuación que se ejecuta cuando se completa la tarea de destino según el elemento TaskContinuationOptions especificado. La continuación recibe un token de cancelación y usa un programador especificado.
ContinueWith(Action<Task>, TaskContinuationOptions)	Crea una continuación que se ejecuta cuando se completa la tarea de destino según el elemento TaskContinuationOptions especificado.
ContinueWith(Action<Task>, TaskScheduler)	Crea una continuación que se ejecuta de manera asincrónica cuando se completa el objeto Task de destino. La continuación usa un programador especificado.

ContinueWith<TResult> (Func<Task,Object,TResult>, Object)	Crea una continuación que recibe información de estado proporcionada por el autor de la llamada y se ejecuta de forma asincrónica cuando el elemento Task de destino se completa y devuelve un valor.
ContinueWith<TResult> (Func<Task,Object,TResult>, Object, CancellationToken)	Crea una continuación que se ejecuta de forma asincrónica cuando el elemento Task de destino se completa y devuelve un valor. La continuación recibe información de estado proporcionada por el autor de la llamada y un token de cancelación.
ContinueWith<TResult> (Func<Task,Object,TResult>, Object, CancellationToken, TaskContinuationOptions, TaskScheduler)	Crea una continuación que se ejecuta según las opciones de continuación de la tarea especificadas cuando el elemento Task de destino se completa y devuelve un valor. La continuación recibe información de estado proporcionada por el autor de la llamada y un token de cancelación y usa el programador especificado.
ContinueWith<TResult> (Func<Task,Object,TResult>, Object, TaskContinuationOptions)	Crea una continuación que se ejecuta según las opciones de continuación de la tarea especificadas cuando el elemento Task de destino se completa.La continuación recibe información de estado proporcionada por el autor de la llamada.
ContinueWith<TResult> (Func<Task,Object,TResult>, Object, TaskScheduler)	Crea una continuación que se ejecuta de manera asincrónica cuando se completa el objeto Task de destino. La continuación recibe información de estado proporcionada por el autor de la llamada y usa a un programador especificado.
ContinueWith<TResult> (Func<Task,TResult>)	Crea una continuación que se ejecuta de forma asincrónica cuando el elemento Task<TResult> de destino se completa y devuelve un valor.
ContinueWith<TResult> (Func<Task,TResult>, CancellationToken)	Crea una continuación que se ejecuta de forma asincrónica cuando el elemento Task de destino se completa y devuelve un valor. La continuación recibe un token de cancelación.
ContinueWith<TResult> (Func<Task,TResult>, CancellationToken, TaskContinuationOptions, TaskScheduler)	Crea una continuación que se ejecuta según las opciones de continuación especificadas y devuelve un valor. Se pasa un token de cancelación a la continuación y usa un programador especificado.
ContinueWith<TResult> (Func<Task,TResult>, TaskContinuationOptions)	Crea una continuación que se ejecuta según las opciones de continuación especificadas y devuelve un valor.
ContinueWith<TResult> (Func<Task,TResult>, TaskScheduler)	Crea una continuación que se ejecuta de forma asincrónica cuando el elemento Task de destino se completa y devuelve un valor. La continuación usa un programador especificado.
Delay(Int32)	Crea una tarea que se completa después de un número especificado de milisegundos.
Delay(Int32, CancellationToken)	Crea una tarea cancelable que se completa después de un número especificado de milisegundos.
Delay(TimeSpan)	Crea una tarea que se completa después de un intervalo de tiempo especificado.
Delay(TimeSpan, CancellationToken)	Crea una tarea cancelable que se completa después de un intervalo de tiempo específico.
Dispose()	Libera todos los recursos usados por la instancia actual de la clase Task.
Dispose(Boolean)	Desecha el objeto Task y libera todos sus recursos no administrados.
Equals(Object)	Determina si el objeto especificado es igual al objeto actual. (Inherited from Object)

FromCanceled(CancellationToken)	Crea una Task que se finaliza debido a la cancelación con un token de cancelación especificado.
FromCanceled<TResult> (CancellationToken)	Crea una Task<TResult> que se finaliza debido a la cancelación con un token de cancelación especificado.
FromException(Exception)	Crea una Task que finalizó con una excepción especificada.
FromException<TResult> (Exception)	Crea una Task<TResult> que finalizó con una excepción especificada.
FromResult<TResult>(TResult)	Crea un objeto Task<TResult> que se ha completado correctamente con el resultado especificado.</td>
GetAwaiter()	Obtiene un awaiter utilizado para esperar este objeto Task.
GetHashCode()	Sirve como la función hash predeterminada. (Inherited from Object)
GetType()	Obtiene el Type de la instancia actual. (Inherited from Object)
MemberwiseClone()	Crea una copia superficial del objeto Object actual. (Inherited from Object)
Run(Action)	Pone en cola el trabajo especificado para ejecutarlo en el grupo de subprocesos y devuelve un objeto Task que representa ese trabajo.
Run(Action, CancellationToken)	Pone en cola el trabajo especificado para ejecutarlo en el grupo de subprocesos y devuelve un objeto Task que representa ese trabajo. Un token de cancelación permite cancelar el trabajo.
Run(Func<Task>)	Pone en cola el trabajo especificado para ejecutarlo en el grupo de subprocesos y devuelve un proxy para la tarea devuelta por function.
Run(Func<Task>, CancellationToken)	Pone en cola el trabajo especificado para ejecutarlo en el grupo de subprocesos y devuelve un proxy para la tarea devuelta por function.
Run<TResult> (Func<Task<TResult>>)	Pone en cola el trabajo especificado para ejecutarlo en el grupo de subprocesos y devuelve un proxy para Task(TResult) que devuelve function.
Run<TResult> (Func<Task<TResult>>, CancellationToken)	Pone en cola el trabajo especificado para ejecutarlo en el grupo de subprocesos y devuelve un proxy para Task(TResult) que devuelve function.
Run<TResult>(Func<TResult>)	Pone en cola el trabajo especificado para ejecutarlo en el grupo de subprocesos y devuelve un objeto Task<TResult> que representa ese trabajo.
Run<TResult>(Func<TResult>, CancellationToken)	Pone en cola el trabajo especificado para ejecutarlo en el grupo de subprocesos y devuelve un objeto Task(TResult) que representa ese trabajo. Un token de cancelación permite cancelar el trabajo.
RunSynchronously()	Ejecuta sincrónicamente el objeto Task en el objeto TaskScheduler actual.
RunSynchronously(TaskScheduler)	Ejecuta sincrónicamente el objeto Task en el objeto TaskSchedulerproporcionado.
Start()	Inicia el objeto Task, programando su ejecución en el objeto TaskScheduleractual.
Start(TaskScheduler)	Inicia el objeto Task, programando su ejecución en el objeto TaskSchedulerespecificado.
ToString()	Devuelve una cadena que representa el objeto actual. (Inherited from Object)
Wait()	Espera a que se complete la ejecución del objeto Task.
Wait(CancellationToken)	Espera a que se complete la ejecución del objeto Task. La espera

	finalizará si un token de cancelación se cancela antes de que finalice la tarea.
Wait(Int32)	Espera a que el objeto Task complete la ejecución dentro de un número especificado de milisegundos.
Wait(Int32, CancellationToken)	Espera a que se complete la ejecución del objeto Task. La espera finalizará si transcurre un intervalo de tiempo de espera o un token de cancelación se cancela antes de que finalice la tarea.
Wait(TimeSpan)	Espera a que Task complete la ejecución dentro de un intervalo de tiempo especificado.
WaitAll(Task[])	Espera que se complete la ejecución de todos los objetos Taskproporcionados.
WaitAll(Task[], CancellationToken)	Espera que se complete la ejecución de todos los objetos Taskproporcionados, a menos que se cancele la espera.
WaitAll(Task[], Int32)	Espera a que todos los objetos proporcionados de Task completen la ejecución dentro de un número especificado de milisegundos.
WaitAll(Task[], Int32, CancellationToken)	Espera a que todos los objetos Task proporcionados completen la ejecución dentro de un número especificado de milisegundos o hasta que se cancele la espera.
WaitAll(Task[], TimeSpan)	Espera a que todos los objetos Task cancelables que se hayan proporcionado completen la ejecución en un intervalo de tiempo especificado.
WaitAny(Task[])	Espera a que se complete la ejecución de cualquiera de los objetos Taskproporcionados.
WaitAny(Task[], CancellationToken)	Espera que se complete la ejecución de cualquiera de los objetos Taskproporcionados, a menos que se cancele la espera.
WaitAny(Task[], Int32)	Espera a que cualquiera de los objetos Task que se hayan proporcionado complete su ejecución dentro de un número especificado de milisegundos.
WaitAny(Task[], Int32, CancellationToken)	Espera a que cualquiera de los objetos Task proporcionados complete la ejecución dentro de un número especificado de milisegundos o hasta que se cancele un token de cancelación.
WaitAny(Task[], TimeSpan)	Espera a que se complete la ejecución de cualquier objeto Taskproporcionado en un intervalo de tiempo especificado.
WhenAll(IEnumerable<Task>)	Crea una tarea que se completará cuando todos los objetos Task de una colección enumerable se hayan completado.
WhenAll(Task[])	Crea una tarea que se completará cuando todos los objetos Task de una matriz se hayan completado.
WhenAll<TResult> (IEnumerable<Task<TResult>>)	Crea una tarea que se completará cuando todos los objetos Taskde una colección enumerable se hayan completado.
WhenAll<TResult>(Task<TResult>[])	Crea una tarea que se completará cuando todos los objetos Task<TResult>de una matriz se hayan completado.
WhenAny(IEnumerable<Task>)	Crea una tarea que se completará cuando se haya completado cualquiera de las tareas proporcionadas.
WhenAny(Task[])	Crea una tarea que se completará cuando se haya completado cualquiera de las tareas proporcionadas.
WhenAny<TResult> (IEnumerable<Task<TResult>>)	Crea una tarea que se completará cuando se haya completado cualquiera de las tareas proporcionadas.
WhenAny<TResult>(Task<TResult>[])	Crea una tarea que se completará cuando se haya completado cualquiera de las tareas proporcionadas.

Yield()	Crea una tarea que admite "await" que, de forma asincrónica, devuelve al contexto actual cuando es "awaited".

Ejemplo

```
  static void Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");
    Egg eggs = FryEggs(2);
    Console.WriteLine("eggs are ready");
    Bacon bacon = FryBacon(3);
    Console.WriteLine("bacon is ready");
    Toast toast = ToastBread(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");
    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}

static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");
    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = MakeToastWithButterAndJamAsync(2);

    var eggs = await eggsTask;
    Console.WriteLine("eggs are ready");
    var bacon = await baconTask;
    Console.WriteLine("bacon is ready");
    var toast = await toastTask;
    Console.WriteLine("toast is ready");
    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");

    Console.WriteLine("Breakfast is ready!");

    async Task<Toast> MakeToastWithButterAndJamAsync(int number)
    {
        var toast = await ToastBreadAsync(number);
        ApplyButter(toast);
        ApplyJam(toast);
        return toast;
    }
}
await Task.WhenAll(eggTask, baconTask, toastTask);
Console.WriteLine("eggs are ready");
Console.WriteLine("bacon is ready");
Console.WriteLine("toast is ready");
Console.WriteLine("Breakfast is ready!");


static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");
    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = MakeToastWithButterAndJamAsync(2);

    var allTasks = new List<Task>{eggsTask, baconTask, toastTask};
    while (allTasks.Any())
    {
        Task finished = await Task.WhenAny(allTasks);
        if (finished == eggsTask)
        {
            Console.WriteLine("eggs are ready");
        }
```

```
        else if (finished == baconTask)
        {
            Console.WriteLine("bacon is ready");
        }
        else if (finished == toastTask)
        {
            Console.WriteLine("toast is ready");
        }
        allTasks.Remove(finished);
    }
    Console.WriteLine("Breakfast is ready!");

    async Task<Toast> MakeToastWithButterAndJamAsync(int number)
    {
        var toast = await ToastBreadAsync(number);
        ApplyButter(toast);
        ApplyJam(toast);
        return toast;
    }
}
```

4.- 1. PARAR EN C# LA EJECUCIÓN DE LOS HILOS.

4.1.- EL MÉTODO SLEEP DE LA CLASE THREAD.

Función: detener la ejecución del hilo actual por la cantidad de milisegundos indicada en el parámetro, siempre sujeto a la precisión de los temporizadores y planificadores del sistema.

Sintaxis:

```
  public static void Sleep (int millisecondsTimeout);
```

Ejemplo 4

```
  using System;
using System.Threading;

class Example
{
    static void Main()
    {
        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine("Sleep for 2 seconds.");
            Thread.Sleep(2000);
        }

        Console.WriteLine("Main thread exits.");
    }
}

/* This example produces the following output:

Sleep for 2 seconds.
Sleep for 2 seconds.
Sleep for 2 seconds.
Sleep for 2 seconds.
Sleep for 2 seconds.
Main thread exits.
 */
```

4.2.- EL MÉTODO YIELD DE LA CLASE THREAD.

Función: Hace que el subproceso que realiza la llamada ceda la ejecución a otro subproceso que está listo para ejecutarse en el procesador actual. El sistema operativo selecciona el subproceso al que se va a ceder la ejecución.

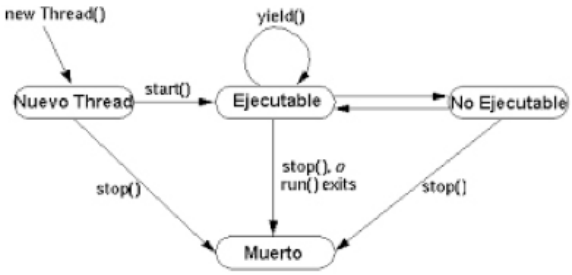
Sintaxis:

```
public static bool Yield ();
```

5.- CICLO DE VIDA DE UN HILO.

En la siguiente figura se pueden ver los 4 posibles estados de los hilos en Java así como las transiciones entre ellos a través de métodos de la clase **Thread:**

Ciclo de vida de un hilo



Autor *(link: https://www.google.es)* (Licencia) *(link: https://www.google.es)* Procedencia *(link: https://www.google.es)*

Los **estados de los hilos** son los siguientes:

- Nuevo **thread:** es el estado cuando se crea un objeto hilo con el operador new. En este estado el hilo aún no se ejecuta; es decir, el programa no ha comenzado la ejecución del código del método **run** del hilo.
- **Ejecutable:** cuando se invoca al método **start**, el hilo pasa a este estado. El sistema operativo tiene que asignar tiempo de CPU al hilo para que se ejecute; por tanto, el hilo puede estar o no en ejecución.
- **No ejecutable:** cuando el hilo no puede ejecutarse, ya que hay algo que lo impide.
- **Muerto:** un hilo puede morir por varias razones. En primer lugar, por muerte natural (cuando el método **run** finaliza con normalidad) o por muerte no natural (debido a alguna excepción no capturada en el método **run**).

Las posibles **transiciones** entre los estados de los hilos son las siguientes:

- Transición de **Nuevo thread** a **Ejecutable**: cuando el hilo invoca al método **start**.
- Transición de **Ejecutable** a **No ejecutable**. Se produce por alguna de las siguientes situaciones:
  - El hilo invoca el método **sleep**.
  - El hilo invoca el método **wait**.
  - El hilo se bloquea en una o**peración de I/O** (entrada/salida).
- Transición de **No ejecutable** a **Ejecutable**. Se produce por alguna de las siguientes situaciones:
  - Si el hilo había invocado el método **sleep** y el número de milisegundos de pausa ya ha transcurrido.
  - Si el hilo había invocado el método **wait** de un objeto y otro hilo le pide que continúe llamando a los métodos **notify** o **notifyAll** del mismo objeto.
  - Si el hilo estaba bloqueado en **operación de I/O** y la operación de entrada/salida se ha completado.
- Transición de **Ejecutable** a **Muerto**: cuando termina el método **run**.

El método **isAlive** ayuda a conocer el estado de un hilo. Los valores que devuelve son:

- **True:** si el hilo ha sido lanzado (método start invocado) y no detenido.
- **False:** si el hilo está en estado Nuevo thread (no ha sido lanzado) o en estado Muerto (método run terminado).

6.- EJECUCIÓN CONCURRENTE Y PARALELA DE HILOS.

Paralelismo: se produce en sistemas con múltiples CPUs o CPUs con varios núcleos; cada CPU puede ejecutar un hilo distinto.

Hilo A	Hilo B
Hilo A	Hilo B
Hilo A	Hilo B
CPU 1	**CPU 2**

Concurrencia: cuando el paralelismo no es posible, una CPU es responsable de ejecutar múltiples hilos.

- Hilo A
- Hilo B
- Hilo A
- Hilo B
- **CPU**

7.- AFINIDAD DEL PROCESADOR.

En aquellos equipos que disponen de CPUs con varios procesadores o de procesadores con varios núcleos, la afinidad del procesador permite asignar un determinado procesador o núcleo a un programa dado.

En Java es posible conocer el número de procesadores que tiene el equipo de la siguiente forma:

Ejemplo 6

```
public class Procesadores {
   public static void main(String args[]) {
      System.out.println("Número de procesadores "+ Runtime.getRuntime().availableProcessors());
   }
}
```

7.1.- CAMBIO DE AFINIDAD EN SISTEMAS OPERATIVOS WINDOWS.

Windows Modo Gráfico

- Pulsar simultáneamente las teclas `Control` + `Alt` + `Supr` para poder acceder al Administrador de tareas.
- Dentro del Administrador de tareas, seleccionar la aplicación a la que queremos asignarle una afinidad, pinchar en el botón derecho del ratón e ir a su proceso. Si no se trata de una aplicación podemos ir directamente a la pestaña de procesos.
- En el caso de la máquina virtual de Java, el proceso a seleccionar sería uno llamado **javaw.exe**, el cual se encarga de ejecutar los programas Java. Seleccionar el proceso **javaw.exe** y, con el botón derecho, asignar su afinidad. Seleccionar el procesador que se le quiere asignar al proceso **javaw.exe**.
- En el caso del IDE Eclipse, el proceso a seleccionar sería uno llamado **eclipse.exe**, el cual se encarga de compilar y ejecutar los programas Java. Seleccionar el proceso **eclipse.exe** y, con el botón derecho, asignar su afinidad. Seleccionar el procesador que se le quiere asignar al proceso **eclipse.exe**.

Windows modo comando

Por ejemplo, para asignar una determinada CPU a la aplicación Bloc de notas, se puede usar el siguiente comando:

```
c:\windows\system32\cmd.exe /C start /affinity 1 notepad.exe
```

Se puede ver en el administrador de tareas que el proceso asociado a la aplicación Bloc de Notas sólo tiene asignada la **CPU 0**

Para iniciar un proceso en la **CPU 0**, habría que usar el siguiente modificador de comando:

```
/affinity 1
```

Para iniciar un proceso en la **CPU 1**, habría que utilizar el siguiente modificador de comando:

```
/affinity 2
```

Se puede utilizar un modificador de comando hasta el número de núcleos de CPU del equipo. La afinidad es esencialmente **CPU core # + 1**, por lo que **/affinity 5** usaría el **CPU 4.**

7.2.- CAMBIO DE AFINIDAD EN SISTEMAS OPERATIVOS LINUX.

La afinidad de CPU no es más que una propiedad del planificador del sistema operativo que enlace un proceso con una determinada CPU. El planificador de Linux respetará la afinidad de CPU establecida y el proceso no se ejecutará en ninguna otra CPU.

El planificador de Linux funciona en base a la llamada afinidad natural de CPU:

- El planificador intenta mantener a los procesos en la misma CPU hasta que sea posible por cuestiones de rendimiento.
- Por lo tanto, forzar una afinidad de CPU específica es sólo útil con ciertas aplicaciones.

Para establecer la afinidad para una tarea o proceso concretos, es preciso usar el comando taskset, el cual no viene instalado por defecto. Para instalarlo, es necesario instalar el paquete schedutils.

7.3.- CAMBIO DE AFINIDAD EN SISTEMAS OPERATIVOS MACOS.

El sistema operativo OS X soporta una API de afinidad de hilos desde la versión 10.5. Un conjunto de afinidad es una colección de hilos que comparten recursos de memoria y que quieren compartir una cache L2.

Los conjuntos de afinidad se identifican con una etiqueta. A los hilos se les asigna a un determinado conjunto de afinidad mediante dicha etiqueta. Un hilo puede pertenecer como mucho a un conjunto de afinidad; es decir, tiene una etiqueta de afinidad.

Efecto de establecer distintas etiquetas de afinidad. Por ejemplo: una aplicación que quiera ejecutar 2 hilos en cachés L2 diferentes, deberá configurar los hilos con diferentes etiquetas de afinidad.

Ejemplo de utilización. Una aplicación que quiera ubicar un hilo en cada procesador disponible, haría lo siguiente:

- Obtener el número de procesadores del sistema, usando sysctl(3).
- Crear ese número de hilos.
- Asignar a cada hilo una etiqueta de afinidad diferente.
- Iniciar todos los hilos.

8.- PRIORIDADES DE LOS HILOS.

ThreadPriority define el conjunto de todos los valores posibles para una prioridad de subproceso. Prioridades de subprocesos especifican la prioridad relativa de un subproceso frente a otro. Cada subproceso tiene una prioridad asignada. Los subprocesos creados en el tiempo de ejecución se les asigna inicialmente la Normal prioridad, mientras que los subprocesos creados fuera el tiempo de ejecución conservan su prioridad anterior cuando entran en el tiempo de ejecución. Puede obtener y establecer la prioridad de un subproceso mediante el acceso a su Priority propiedad. Los subprocesos están programados para ejecutarse según su prioridad. El algoritmo de programación que se usa para determinar el orden de ejecución de subprocesos varía en función de cada sistema operativo. El sistema operativo también puede ajustar la prioridad del subproceso dinámicamente según se mueve el foco de la interfaz de usuario entre el primer y el segundo plano. La prioridad de un subproceso no afecta al estado del subproceso; el estado del subproceso debe estar Running antes de que el sistema operativo puede programarlo.

AboveNormal	3	Thread puede programarse después de los subprocesos con prioridad Highest y antes que los subprocesos con prioridad Normal.
BelowNormal	1	Thread puede programarse después de los subprocesos con prioridad Normal y antes que los subprocesos con prioridad Lowest.

Highest	4	Thread puede programarse antes que los subprocesos que tengan cualquier otra prioridad.
Lowest	0	Thread puede programarse después de los subprocesos que tengan cualquier otra prioridad.
Normal	2	Thread puede programarse después de los subprocesos con prioridad AboveNormal y antes que los subprocesos con prioridad BelowNormal. Los subprocesos tienen prioridad Normal de forma predeterminada.

Ejemplo

```
  using System;
using System.Threading;
using Timers = System.Timers;

class Test
{
    static void Main()
    {
        PriorityTest priorityTest = new PriorityTest();

        Thread thread1 = new Thread(priorityTest.ThreadMethod);
        thread1.Name = "ThreadOne";
        Thread thread2 = new Thread(priorityTest.ThreadMethod);
        thread2.Name = "ThreadTwo";
        thread2.Priority = ThreadPriority.BelowNormal;
        Thread thread3 = new Thread(priorityTest.ThreadMethod);
        thread3.Name = "ThreadThree";
        thread3.Priority = ThreadPriority.AboveNormal;

        thread1.Start();
        thread2.Start();
        thread3.Start();
        // Allow counting for 10 seconds.
        Thread.Sleep(10000);
        priorityTest.LoopSwitch = false;
    }
}

class PriorityTest
{
    static bool loopSwitch;
    [ThreadStatic] static long threadCount = 0;

    public PriorityTest()
    {
        loopSwitch = true;
    }

    public bool LoopSwitch
    {
        set{ loopSwitch = value; }
    }

    public void ThreadMethod()
    {
        while(loopSwitch)
        {
            threadCount++;
        }
        Console.WriteLine("{0,-11} with {1,11} priority " +
            "has a count = {2,13}", Thread.CurrentThread.Name,
            Thread.CurrentThread.Priority.ToString(),
            threadCount.ToString("N0"));
    }
}
// The example displays output like the following:
//    ThreadOne   with      Normal priority has a count =   755,897,581
//    ThreadThree with AboveNormal priority has a count =   778,099,094
//    ThreadTwo   with BelowNormal priority has a count =     7,840,984
```

9.- HILOS Y LA PORTABILIDAD DE JAVA: DEBILIDADES.

9.1.- TIME SLICING.

Los responsables de la ejecución de los hilos son los sistemas operativos. Pero no todos los sistemas operativos manejan los hilos de la misma forma. Por ejemplo: los sistemas operativos NT y Solaris tienen diferentes niveles de prioridades *(incluso diferentes respecto de las definidas por Java).*

Time slicing (subdivisión de tiempo): cuando el sistema operativo asigna una porción de tiempo a la ejecución de cada hilo. En este caso, la ejecución de un hilo es interrumpida no sólo por otro hilo de mayor prioridad que pasa a estado "ejecutable", sino también cuando su tiempo asignado de ejecución se acaba. No todos los sistemas operativos implementan el concepto de **time slicing.**

9.2.- HILOS EGOÍSTAS.

Este concepto aparece en los sistemas operativos que no implementan el concepto de **time slicing**, cuando un determinado hilo entra en estado Ejecutable y continua su ejecución hasta morir. Durante ese período de tiempo ningún otro hilo podrá ser ejecutado.

Por lo tanto:

- No se debe asumir que la ejecución de una aplicación Java se vaya a producir siempre en sistemas que soporten el concepto de **time slicing**.
- En consecuencia, para evitar el egoísmo de los hilos, es necesario incluir adecuadamente invocaciones a los métodos **yield, sleep y wait**, si los hilos no se bloquean en operaciones de entrada/salida.

9.3.- ACCESO A DATOS COMPARTIDOS.

Es común que dos o más hilos tengan acceso a objetos comunes.

Ejemplo 7 – Aplicación con 2 hilos (Primero y Segundo) que actualizan un objeto compartido (mensajes) de la clase Historial

```
        // Hilos.java
public class Hilos extends Thread {
        String mensaje;
        Historial historial;

public Hilos (String msg, Historial h){
                mensaje = msg;
                historial = h;
        }

public void run() {
                for (int i=1;i<=20;i++){
                                historial.agregar( mensaje );
                                yield();
                }
        }
}

//Historial.java

public class Historial {
        String[] mensajes = new String[1000];
        int pos = 0;

public void agregar(String msg) {
                mensajes[pos] = msg;
                pos++;
        }
}

//Ejemplo7.java

public class Ejemplo7 {
```

```
        public static void main(String arg[]) {

                Historial historial= new Historial();
                Hilos p = new Hilos("Primero", historial);
                Hilos s = new Hilos("Segundo", historial);
                p.start();
                s.start();

        }
}
```

Se espera que ocurra lo siguiente:

pos=0	Hilo Primero	Hilo Segundo
mensaje[pos]=msg;	mensaje[0]=”Primero	
pos=pos+1	pos=1	
mensaje[pos]=msg;		mensaje[1]=”Segundo”
pos=pos+1		pos=2
mensaje[pos]=msg;	mensaje[2]=”Primero”;	
pos=pos+1	pos=3	

Pero podría ocurrir lo siguiente:

pos=0	Hilo Primero	Hilo Segundo
mensaje[pos]=msg;	mensaje[0]=”Primero”	
mensaje[pos]=msg;		mensaje[0]=”Segundo”
pos=pos+1	pos=1	
pos=pos+1		pos=2
mensaje[pos]=msg;	mensaje[2]=”Primero”;	
pos=pos+1	pos=3	

Es decir, en ocasiones se puede machacar la posición si el segundo hilo hace la asignación antes de que el primero incremente la posición.

10.- BLOQUEO DE OBJETOS COMPARTIDOS.

10.1.- LOCKS.

La sincronización para el acceso a objetos compartidos se basa en el concepto de **“monitor”**, desarrollado por Charles Antony Richard Hoare. Un monitor es una porción de código protegida por un **“mutex”** (“mutual exclusion semaphore”).

Sólo un hilo puede tener el mutex de un objeto en un momento dado. Si un segundo hilo trata de obtener un mutex ya adquirido por otro hilo, se bloquea hasta que el primero libere el mutex. En el momento de liberarse un mutex, todos los hilos en espera de él se “despertarán” (en base a algún criterio) y el mutex se asignará a uno de ellos.

Analogía de los conceptos de monitor y mutex

Para entender el funcionamiento del monitor y del mutex, se va a pensar en un edificio en el cual algunas oficinas tienen llave y otras tienen libre acceso. El monitor es el conjunto de oficinas cuyo acceso requiere la llave. Los hilos son las personas que quieren acceder a las

oficinas.

Para entrar a una oficina con llave, una persona tiene que obtener el manojo con las llaves de la oficina. El manojo con las llaves es el **mutex** del edificio: solo la persona que tiene el manojo puede entrar a las oficinas con llave.

Ejemplo

```
using System;
using System.Threading.Tasks;
public class Account
{
    private readonly object balanceLock = new object();
    private decimal balance;
    public Account(decimal initialBalance)
    {
        balance = initialBalance;
    }
    public decimal Debit(decimal amount)
    {
        lock (balanceLock)
        {
            if (balance >= amount)
            {
                Console.WriteLine($"Balance before debit :{balance, 5}");
                Console.WriteLine($"Amount to remove     :{amount, 5}");
                balance = balance - amount;
                Console.WriteLine($"Balance after debit  :{balance, 5}");
                return amount;
            }
            else
            {
                return 0;
            }
        }
    }
    public void Credit(decimal amount)
    {
        lock (balanceLock)
        {
            Console.WriteLine($"Balance before credit:{balance, 5}");
            Console.WriteLine($"Amount to add        :{amount, 5}");
            balance = balance + amount;
            Console.WriteLine($"Balance after credit :{balance, 5}");
        }
    }
}

class AccountTest
{
    static void Main()
    {
        var account = new Account(1000);
        var tasks = new Task[100];
        for (int i = 0; i < tasks.Length; i++)
        {
            tasks[i] = Task.Run(() => RandomlyUpdate(account));
        }
        Task.WaitAll(tasks);
    }
    static void RandomlyUpdate(Account account)
    {
        var rnd = new Random();
        for (int i = 0; i < 10; i++)
        {
            var amount = rnd.Next(1, 100);
            bool doCredit = rnd.NextDouble() < 0.5;
            if (doCredit)
            {
                account.Credit(amount);
            }
            else
            {
                account.Debit(amount);
            }
```

```
        }
    }
}
```

En el momento de que un hilo o tarea entra a un bloque lock, el hilo se encontrará con una de las siguientes situaciones:

- **Mutex libre:** el hilo tomará el mutex, ejecutará el método y lo liberará al terminar la ejecución del método.
- **Mutex tomado por otro hilo:** el hilo se bloqueará en espera de que el primero lo libere.

Por lo tanto, sólo un hilo a la vez podrá ejecutar un método **synchronized** sobre un objeto. Si por lo que fuera, se interrumpe la ejecución del hilo que ha tomado el mutex, el paso se dará a otro hilo que no requiera el mutex sobre tal objeto. Una vez liberado el mutex por el hilo que lo tiene tomado, los hilos bloqueados en espera de él se vuelven "ejecutables", aunque sólo a uno de ellos se le asignará el mutex.

La invocación al método **sleep** de la clase Thread por parte de un hilo, no libera el mutex de los objetos que eventualmente pudiera tener bloqueados.

Los **inconvenientes** de declararbloques de código lock son los siguientes:

- Mayor lentitud en la ejecución de dichos métodos, ya que el hilo que toma el mutex bloquea la posible ejecución de dicho método para otros hilos.
- **Peligro de deadlock (abrazo mortal):** bloqueo mutuo de dos hilos que esperan adquirir mutex intercambiados.

10.2.- DEADLOCK.

El DeadLock es el bloqueo mutuo de dos hilos que esperan adquirir mutex intercambiados. Este tipo de bloqueos se deben a que dos hilos o tareas necesitan el mutex de dos objetos diferentes y cada uno de ellos adquiere unos de los dos y al intentar adquirir el segundo se quedan bloqueados a la espera. Una posible solución a dicho problema es que el orden de los objetos sobre los que se quiere adquirir el mutex sea el mismo o incluso adquirir al inicio del bloque todos los mutex necesarios. La última opción puede generar problemas de rendimiento severos.

Ejemplo

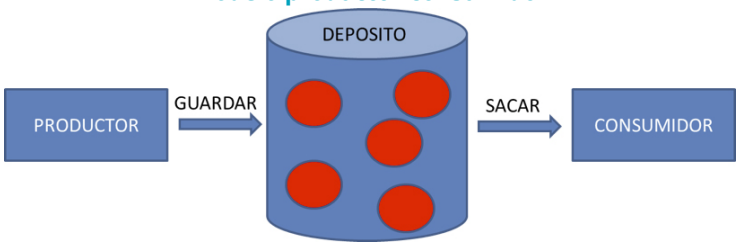
```
using System;
using System.Threading;
namespace deadlockincsharp
{
public class Akshay
    {
        static readonly object firstLock = new object();
        static readonly object secondLock = new object();
        static void ThreadJob()
        {
            Console.WriteLine("\t\t\t\tLocking firstLock");
            lock (firstLock)
            {
                Console.WriteLine("\t\t\t\tLocked firstLock");
                // Wait until we're fairly sure the first thread
                // has grabbed secondLock
                Thread.Sleep(1000);
                Console.WriteLine("\t\t\t\tLocking secondLock");
                lock (secondLock)
                {
                    Console.WriteLine("\t\t\t\tLocked secondLock");
                }
                Console.WriteLine("\t\t\t\tReleased secondLock");
            }
            Console.WriteLine("\t\t\t\tReleased firstLock");
        }
        static void Main()
        {
            new Thread(new ThreadStart(ThreadJob)).Start();
            // Wait until we're fairly sure the other thread
            // has grabbed firstLock
            Thread.Sleep(500);
            Console.WriteLine("Locking secondLock");
            lock (secondLock)
            {
```

```
                Console.WriteLine("Locked secondLock");
                Console.WriteLine("Locking firstLock");
                lock (firstLock)
                {
                    Console.WriteLine("Locked firstLock");
                }
                Console.WriteLine("Released firstLock");
            }
            Console.WriteLine("Released secondLock");
            Console.Read();
        }
    }
}
```

11.- SINCRONIZACIÓN DE HILOS.

11.1.- EL MODELO PRODUCTOR-CONSUMIDOR.

Modelo productor-consumidor



Autor *(link: https://www.google.es)* (Licencia) *(link: https://www.google.es)* Procedencia *(link: https://www.google.es)*

Este modelo se basa en lo siguiente:

- Un hilo (el productor) genera un elemento que es agregado a un depósito.
- El elemento agregado es consumido por otro hilo (consumidor).
- El depósito tiene capacidad limitada y cuando está lleno, el productor debe esperar hasta que haya espacio.
- El consumidor debe esperar hasta que haya elementos en el depósito para poder retirarlos.

Supongamos que la capacidad del depósito es de un elemento. El depósito no solo debe soportar acceso concurrente, sino que el productor y el consumidor tienen que actuar de forma sincronizada.

Una solución simple sería que el productor verificase si hay espacio en el depósito antes de agregar un elemento al mismo y que el consumidor verificase si hay elementos en el depósito antes de sacarlos.

El inconveniente que tiene esta solución es que existe un alto consumo de recursos de CPU en procesos improductivos. Una solución podría ser detener los hilos hasta que se den las condiciones adecuadas para que actúen. Dicha detención se explicará en el siguiente apartado de este documento con unos métodos de la clase Monitor: wait, pulse y pulseAll.

11.2.- MÉTODOS WAIT, NOTIFY Y NOTIFYALL DE LA CLASE OBJECT.

- La clase Monitor provee el método wait() para detener la ejecución de un hilo hasta que le sea notificada la posibilidad de continuar.
- El método wait() debe ser invocado sobre un objeto compartido por los hilos a sincronizar; por ejemplo, en el modelo productor-consumidor, el objeto deposito.
- Para poder invocar el método wait() es necesario que el hilo tenga el mutex del objeto compartido.
- La invocación del método wait() detiene el hilo, lo pone en una lista de espera asociada al objeto y libera su mutex.
- El hilo saldrá de la lista de espera cuando otro hilo invoque el método pulse() sobre el objeto compartido. Al salir de la lista de espera, se bloqueará en espera del mutex del objeto para continuar su ejecución.
- Una vez recuperado el mutex del objeto, el hilo que salió de la lista de espera continuará la ejecución en la instrucción siguiente a la llamada al método wait().

- Si hay más de un hilo en la lista de espera, pulse() reactivará sólo uno de ellos. El criterio de selección del hilo a reactivar depende de diferentes factores.
- El hilo que invoca al método pulse() no tiene ninguna referencia sobre el hilo que está en espera (la notificación actúa sobre un objeto compartido y no sobre un hilo).
- El método pulseAll() permite reactivar todos los hilos bloqueados en la lista de espera de un objeto; es decir, todos los hilos bloqueados se vuelven ejecutables, aunque sólo uno de ellos podrá tomar el mutex a la vez.

11.3.- SINCRONIZACIÓN Y BLOQUEO ITERATIVO (SPIN LOCK).

Tal y como se verá en los ejercicios propuestos cuando hay más de un consumidor el problema no es tan sencillo de resolver. Para ello se dispone de la clase BlockingCollection<T>, la cual ofrece las siguientes características:

- Thread-safe
- Implementación del patrón Consumer-Producer
- Bloqueo de inserciones y borrados cuando la colección está vacía o llena.

Ejemplo

```
  // A bounded collection. It can hold no more
// than 100 items at once.
BlockingCollection<Data> dataItems = new BlockingCollection<Data>(100);


// A simple blocking consumer with no cancellation.
Task.Run(() =>
{
    while (!dataItems.IsCompleted)
    {

        Data data = null;
        // Blocks if dataItems.Count == 0.
        // IOE means that Take() was called on a completed collection.
        // Some other thread can call CompleteAdding after we pass the
        // IsCompleted check but before we call Take.
        // In this example, we can simply catch the exception since the
        // loop will break on the next iteration.
        try
        {
            data = dataItems.Take();
        }
        catch (InvalidOperationException) { }

        if (data != null)
        {
            Process(data);
        }
    }
    Console.WriteLine("\r\nNo more items to take.");
});

// A simple blocking producer with no cancellation.
Task.Run(() =>
{
    while (moreItemsToAdd)
    {
        Data data = GetData();
        // Blocks if numbers.Count == dataItems.BoundedCapacity
        dataItems.Add(data);
    }
    // Let consumer know we are done.
    dataItems.CompleteAdding();});
```