

Fundamentos de la Programación Concurrente y Paralela

Tema 4

Ejemplos de Código

- Todos los ejemplos de **código** mostrados en estas transparencias están **disponibles para el alumno**
 - Siguen apareciendo en etiquetas como la que se muestra:

Consulta el código en:

generics/inference

- Para comprender los conceptos explicados, el alumno deberá **abrir** el código, **analizarlo**, **modificarlo**, **ejecutarlo** y **asegurarse** de que lo **entiende**

Contenido

- [Introducción](#)
- [Programación Concurrente y Paralela](#)
- [Proceso e Hilo](#)
- [Paralelización de Algoritmos](#)
- [Creación Explícita de Hilos](#)
- [Condición de Carrera](#)
- [*Context Switching & Thread Pooling*](#)
- [*Foreground & Background Threads*](#)
- [Sincronización de Hilos y Procesos](#)
- [Interbloqueo](#)
- [Estructuras de Datos *Thread-Safe*](#)
- [Tareas \(Tasks\)](#)
- [Paso Asíncrono de Mensajes](#)
- [Paralelización mediante *Task Parallel Library* y *PLINQ*](#)
- [Paradigma Funcional en la Paralelización de Algoritmos](#)

Ley de Moore

- Es una ley empírica formulada en 1965 por Gordon E. Moore que dice que

En número de transistores por unidad de superficie en circuitos integrados se duplica cada 24 meses, sin encarecer su precio

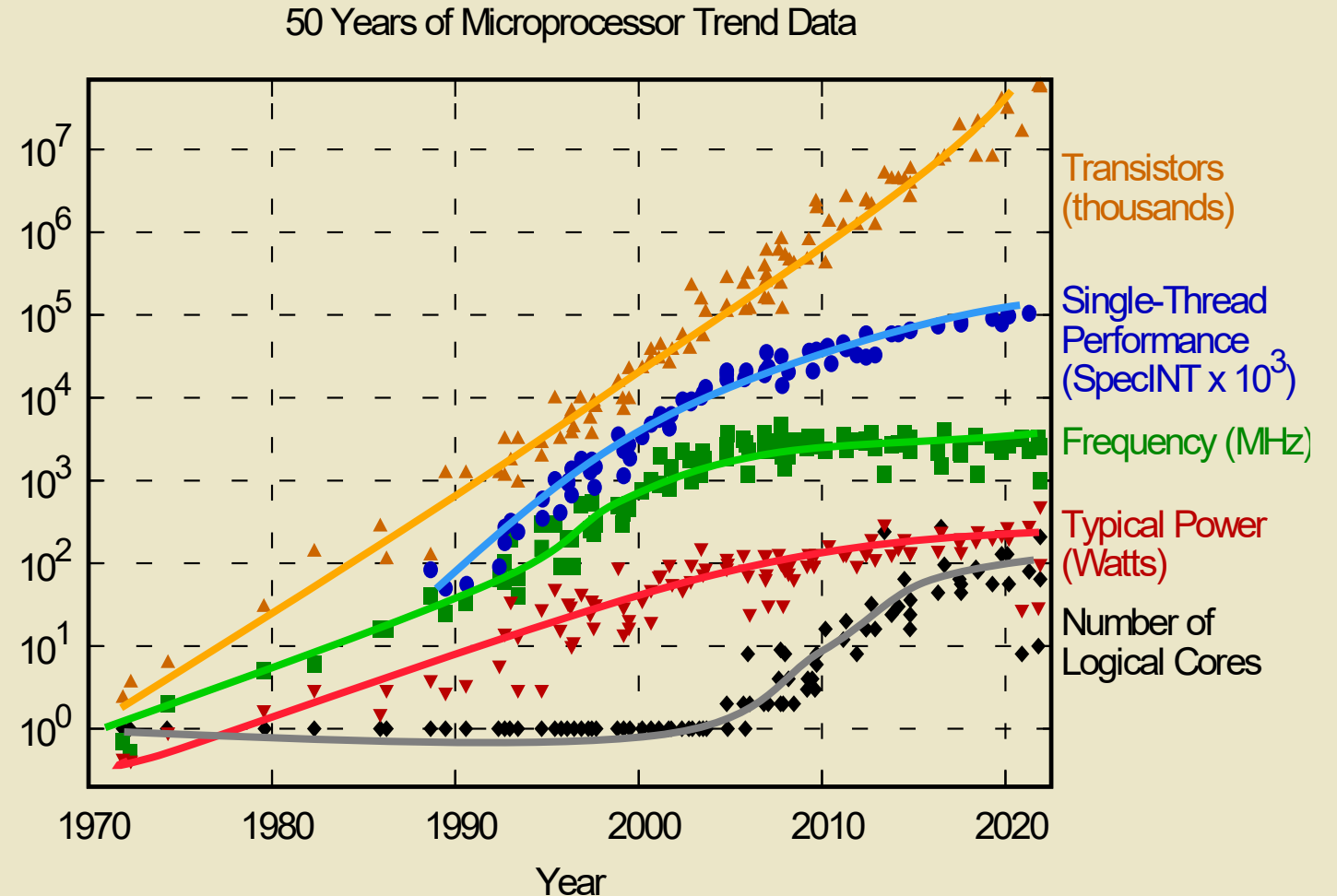
- Esto indica que en dos años, por el mismo precio, tendremos un microprocesador el doble de potente
 - Muchos dispositivos digitales siguen esta ley
- Según Moore, este crecimiento exponencial dejaría de cumplirse entre 2017 y 2022

Ley de Moore

- La ley se ha cumplido respecto al número de transistores
- Pero este crecimiento cambió en lo relativo a la frecuencia del reloj (Mhz), a partir de 2003

Artículo relacionado: <https://www.infoq.com/news/2020/04/Moores-law-55/>

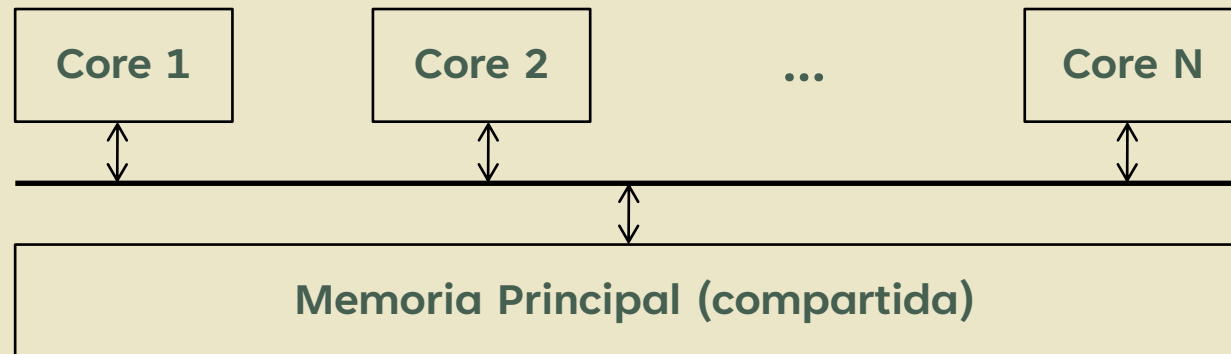
Fuente original (datos) por Karl Rupp:
<https://github.com/karlrupp/microprocessor-trend-data>



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

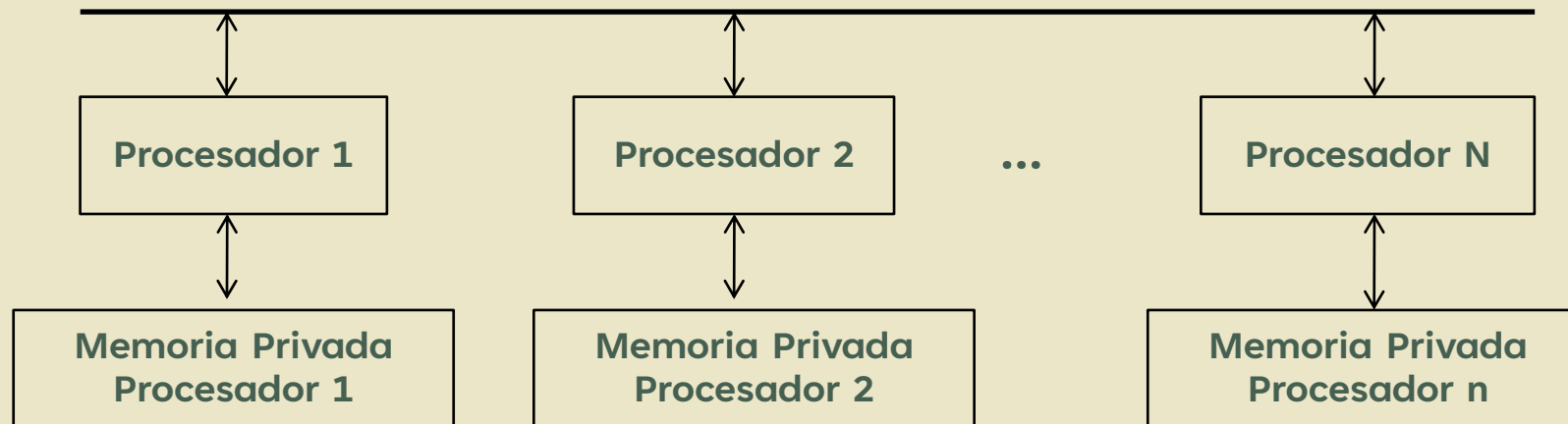
Arquitecturas Multinúcleo

- La tendencia actual es que los ordenadores **amplíen el número de núcleos** (cores) de procesamiento en lugar de la frecuencia de reloj
- Los microprocesadores multinúcleo incluyen **más de un procesador en una misma CPU**
- Ofrecen computación paralela (paralelismo) no sólo a nivel de proceso, sino también a nivel de hilo (**thread**)
- Las arquitecturas multinúcleo poseen una **memoria compartida**



Memoria Distribuida

- En los **sistemas multiprocesador con memoria distribuida**, cada procesador posee una memoria privada
 - Puede tratarse, a su vez, de procesadores multinúcleo
- Si un procesador requiere datos de otro, tiene comunicarse con éste a través de un **canal**
- Ejemplos son un **Cluster, Cloud** o la computación en **Grid**



Programación Concurrente

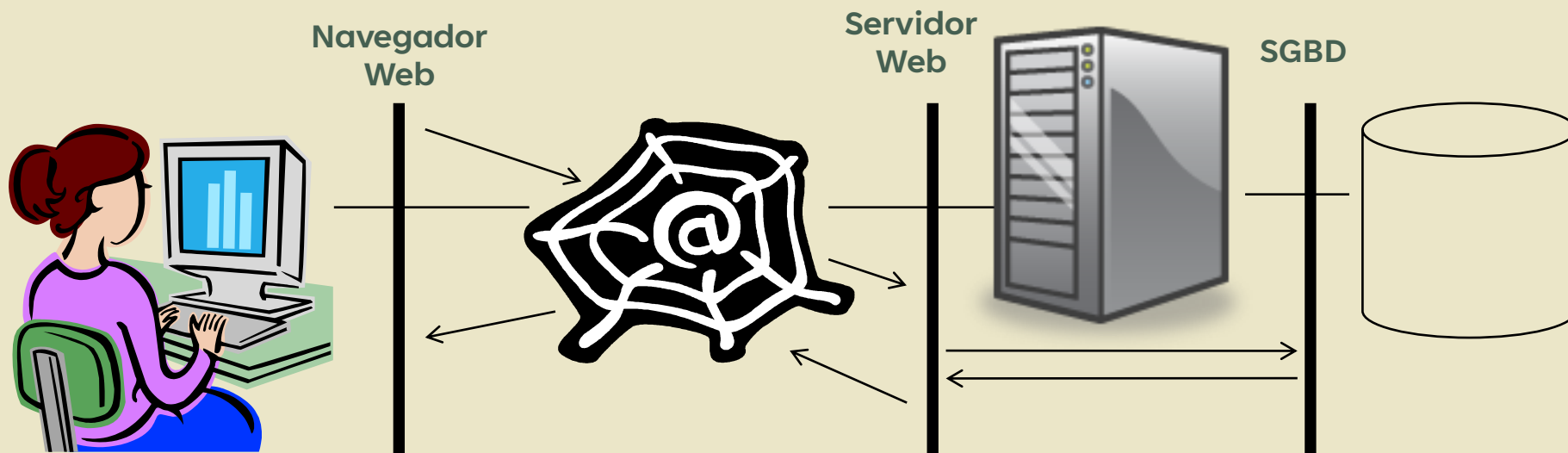
- Debido a esta tendencia, cada vez es más importante realizar programación concurrente y paralela
 - El código tradicional secuencial, donde las instrucciones se ejecutan una tras otra, no hace uso de la capacidad de multiprocesamiento
- **Concurrencia** es la propiedad por la que **varias tareas** se pueden **ejecutar simultáneamente** y potencialmente **interactuar entre sí**
 - Las tareas se pueden ejecutar en varios núcleos, en varios procesadores, o *simulada* en un único procesador
 - Las **tareas** pueden ser hilos o procesos

Programación Paralela

- **Paralelismo** es un caso particular de la concurrencia, en el que las **tareas** se **ejecutan** de **forma paralela** (simultáneamente, no simulada)
 - Con la concurrencia, la simultaneidad puede ser simulada
 - Con el paralelismo, la simultaneidad debe ser real
- El **paralelismo** comúnmente enfatisa la división de un problema en partes (datos, instrucciones, tareas...) más pequeñas
- La programación **concurrente** comúnmente enfatisa la interacción entre tareas

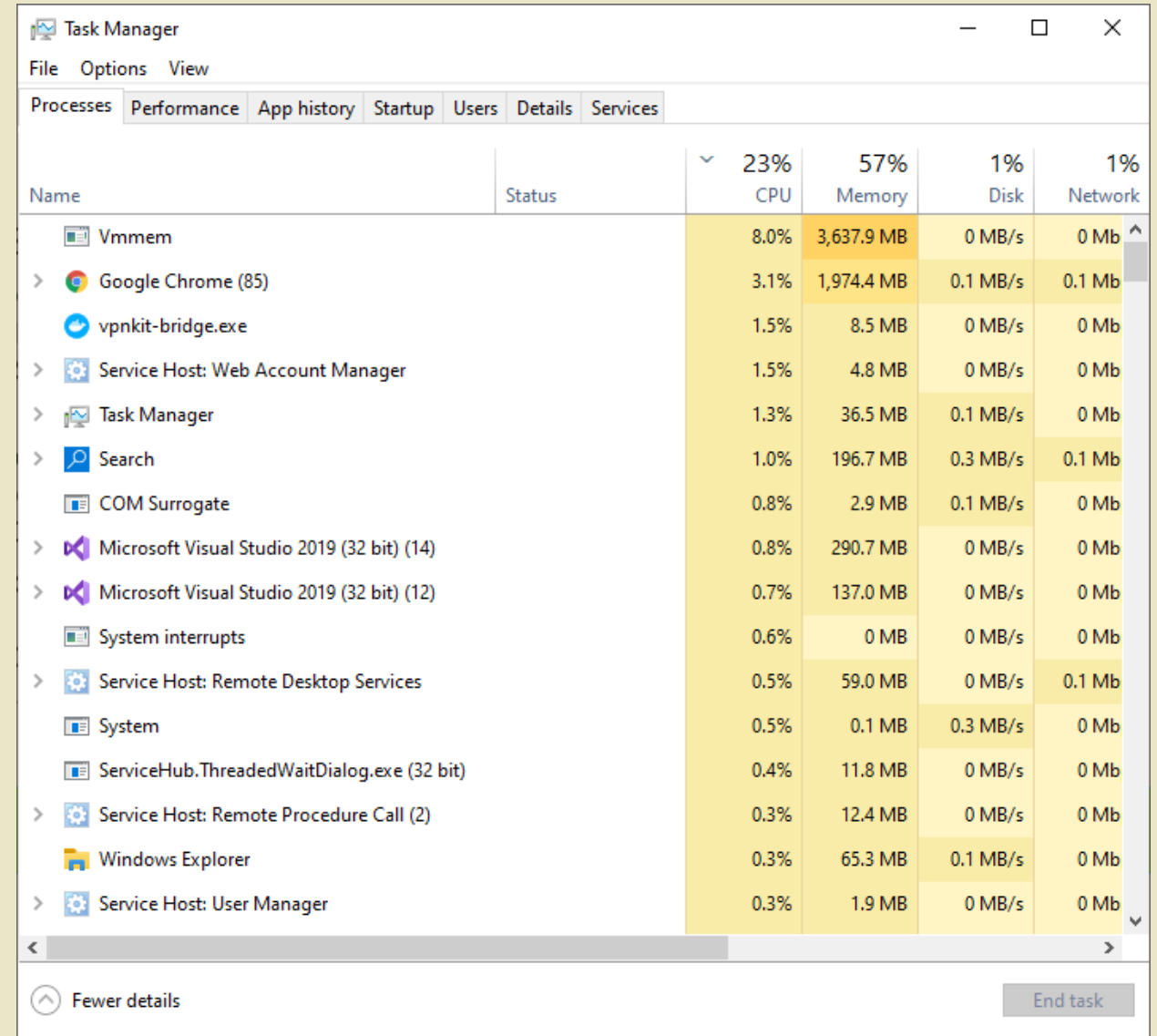
Proceso

- Un **proceso** es un **programa en ejecución**
 - Consta de instrucciones, estado de ejecución y valores de los datos en ejecución
 - En los sistemas de memoria distribuida, las tareas concurrentes en distintos procesadores son procesos
- Ejemplo de sistema con varios procesos



Procesos

- En el SO Windows, podemos ver los procesos en ejecución con Ctrl+Alt+Supr
- Todo proceso tiene un identificador único (PID) – pestaña de servicios
- En .Net los procesos están representados por la clase **Process** (**System.Diagnostics**)

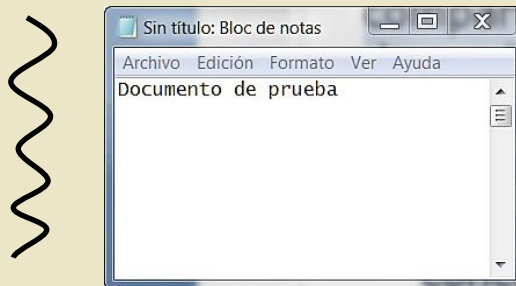


Name	Status	23% CPU	57% Memory	1% Disk	1% Network
Vmmem		8.0%	3,637.9 MB	0 MB/s	0 Mb
> Google Chrome (85)		3.1%	1,974.4 MB	0.1 MB/s	0.1 Mb
vpnkit-bridge.exe		1.5%	8.5 MB	0 MB/s	0 Mb
> Service Host: Web Account Manager		1.5%	4.8 MB	0 MB/s	0 Mb
> Task Manager		1.3%	36.5 MB	0.1 MB/s	0 Mb
> Search		1.0%	196.7 MB	0.3 MB/s	0.1 Mb
COM Surrogate		0.8%	2.9 MB	0.1 MB/s	0 Mb
> Microsoft Visual Studio 2019 (32 bit) (14)		0.8%	290.7 MB	0 MB/s	0 Mb
> Microsoft Visual Studio 2019 (32 bit) (12)		0.7%	137.0 MB	0 MB/s	0 Mb
System interrupts		0.6%	0 MB	0 MB/s	0 Mb
> Service Host: Remote Desktop Services		0.5%	59.0 MB	0 MB/s	0.1 Mb
System		0.5%	0.1 MB	0.3 MB/s	0 Mb
ServiceHub.ThreadedWaitDialog.exe (32 bit)		0.4%	11.8 MB	0 MB/s	0 Mb
> Service Host: Remote Procedure Call (2)		0.3%	12.4 MB	0 MB/s	0 Mb
Windows Explorer		0.3%	65.3 MB	0.1 MB/s	0 Mb
> Service Host: User Manager		0.3%	1.9 MB	0 MB/s	0 Mb

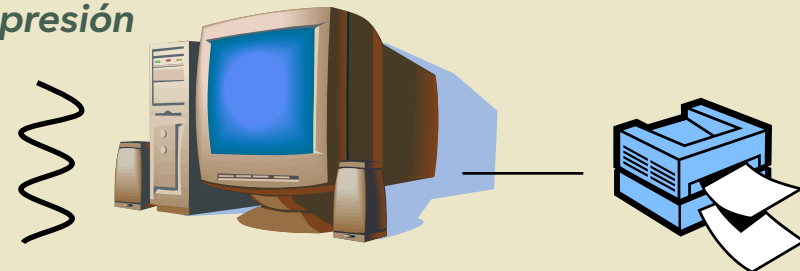
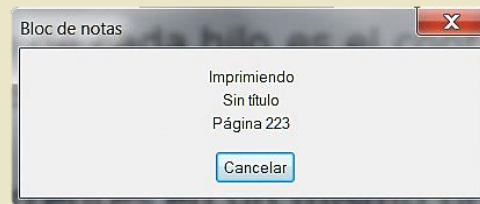
Hilo

- Un **proceso** puede constar de **varios hilos de ejecución** (*threads*)
- Un **hilo** de ejecución es una **tarea de un proceso** que puede ejecutarse **concurrentemente**, **compartiendo la memoria del proceso**, con el resto de sus hilos
 - Propio de cada hilo es el contador de programa, la pila de ejecución y el valor de los registros
 - En los procesadores multinúcleo, los hilos pueden ser tareas paralelas de un mismo proceso
- Ejemplo de aplicación con varios hilos:

GUI



Impresión



Procesos en .Net

- En .Net los procesos se abstraen con instancias de la clase **Process** en `System.Diagnostics`

```
var procesos = Process.GetProcesses();  
foreach (Process proceso in procesos) {  
    Console.WriteLine("PID: {0}\tNombre: {1}\tMemoria  
        virtual: {2:N} MB", proceso.Id, proceso.ProcessName,  
        proceso.VirtualMemorySize64/1024.0/1024);  
}
```

Consulta el código en:

processes.threads/processes

Hilos en .Net

- Los hilos se abstraen con instancias de **Thread** en `System.Threading`

```
Console.WriteLine("Hilo actual. Nombre: {0}, identificador: {1},  
prioridad: {2}, estado: {3}.",  
    Thread.CurrentThread.Name,  
    Thread.CurrentThread.ManagedThreadId,  
    Thread.CurrentThread.Priority,  
    Thread.CurrentThread.ThreadState);
```

Consulta el código en:

[processes.threads/threads](#)

Paralelización de Algoritmos

- Existen dos escenarios típicos de paralelización

- Paralelización de tareas:** Tareas independientes pueden ser ejecutadas concurrentemente

Generar hashes
ficheros



Encriptar
Cadenas



Generar thumbnails
ficheros



- Paralelización de datos:** Ejecutar una misma tarea que computa porciones de los mismos datos

La cadena que tiene que ser encriptada

La cadena



que tiene que



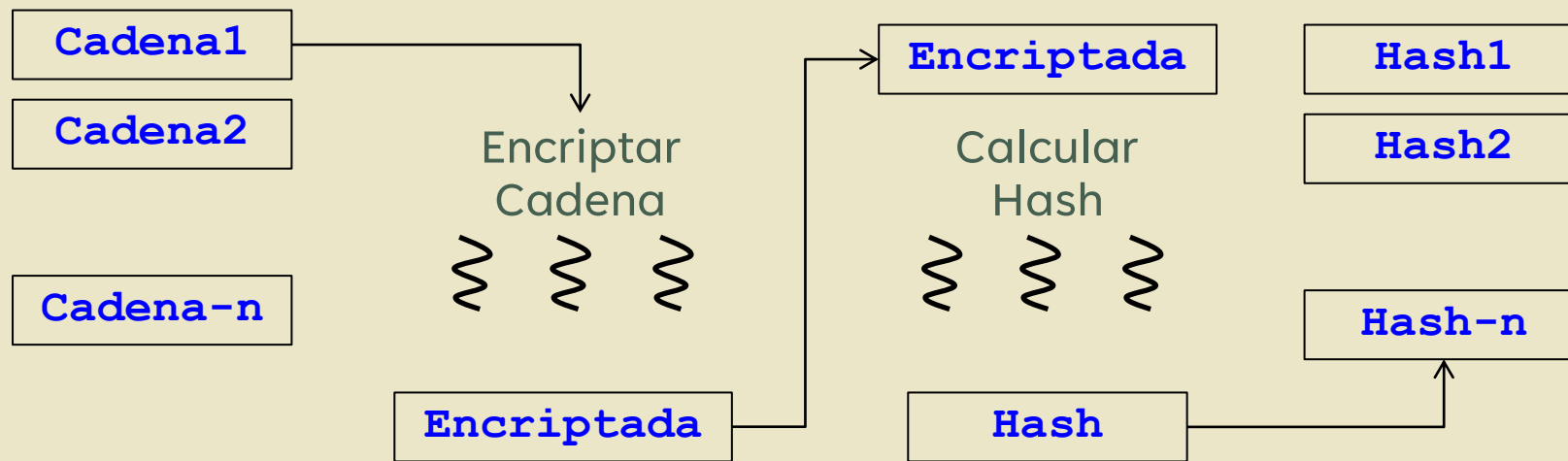
ser encriptada



Encriptar una Cadena

Pipeline

- Existe un modelo híbrido de los dos anteriores denominado *pipeline*
- Si queremos calcular *hashes* de cadenas encriptadas, podemos ejecutar dos tareas paralelas
 1. Encriptar cadena
 2. Calcular hash
- Sincronizamos la salida de una tarea como la entrada de la siguiente



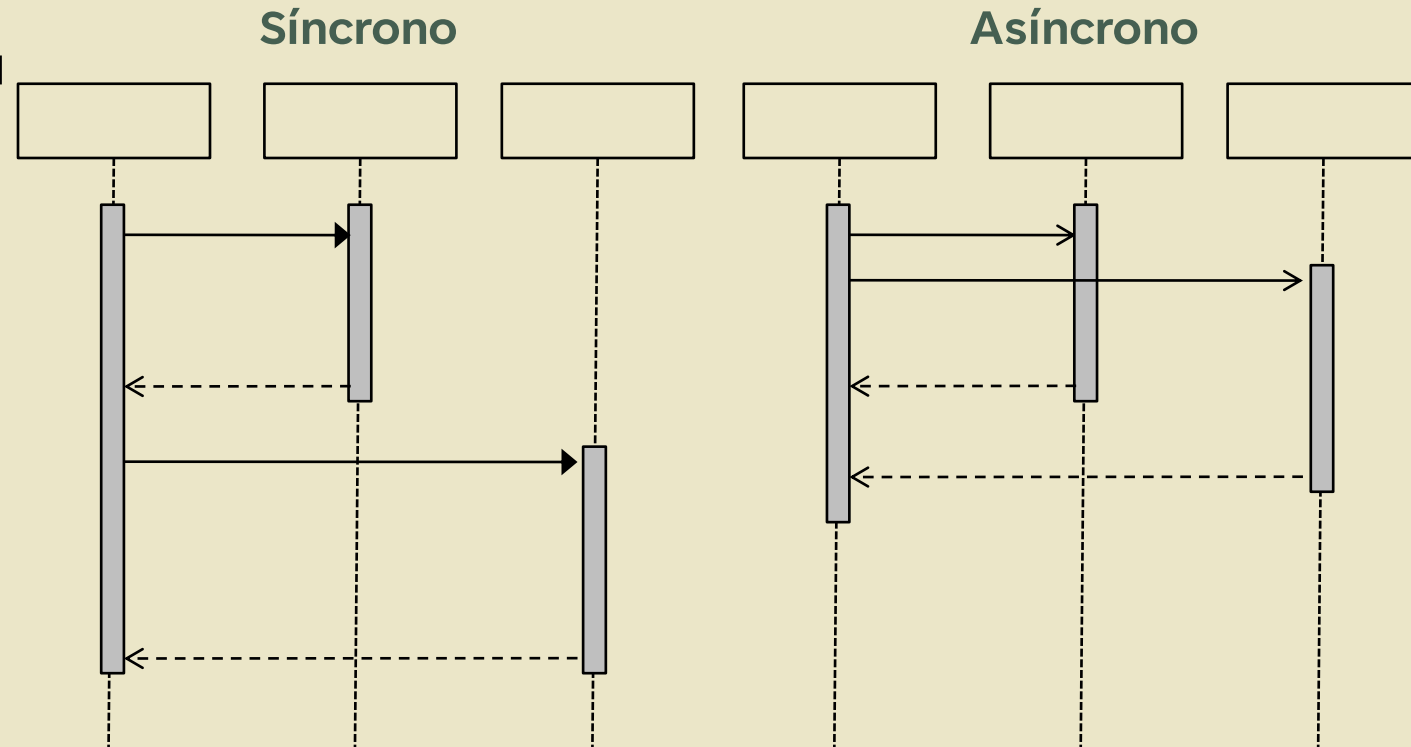


Paso asíncrono de mensajes

Código paralelo

Paso Asíncrono de Mensajes

- Una primera aproximación para crear programas paralelos es el **paso de mensajes asíncrono**
 - Cada mensaje asíncrono crea (potencialmente) un nuevo hilo (*thread*)
- En C# esta funcionalidad se obtiene mediante:
 - **delegados**
 - **Tasks**



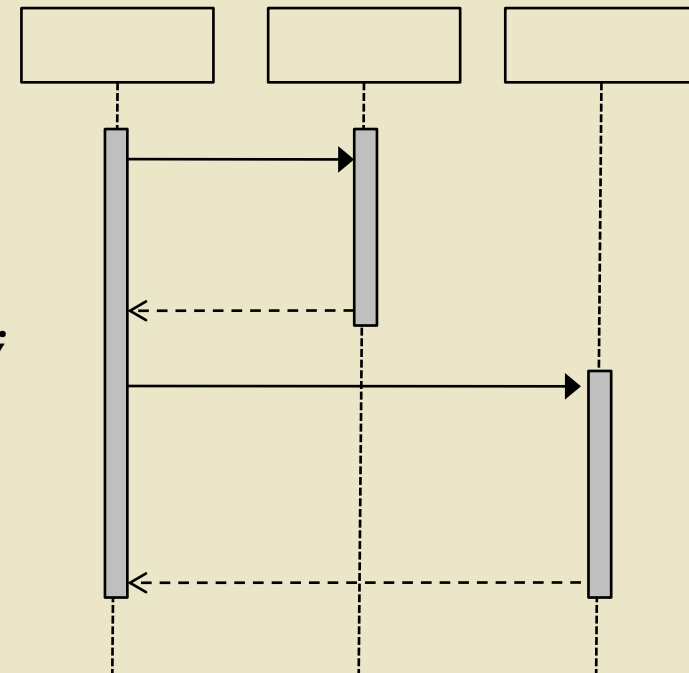
Paso Síncrono de Mensajes

Consulta el código en:
asynchronous.calls/sequential

- El siguiente programa descarga una página Web y cuenta sus *tags* `img`
- La llamada al método `GetNumeroImagenes` es síncrona (secuencial)

¿Es necesario esperar a tener las imagenes de **Uniovi** para descargar las imagenes de la **EII**?

```
PaginaWeb uniovi = new PaginaWeb(  
    "http://www.uniovi.es");  
PaginaWeb escuela = new PaginaWeb(  
    "http://www.ingenieriainformatica.uniovi.es");  
int numeroImgsUniovi =  
    uniovi.GetNumeroImagenes();  
int numeroImgsEscuela =  
    escuela.GetNumeroImagenes();
```

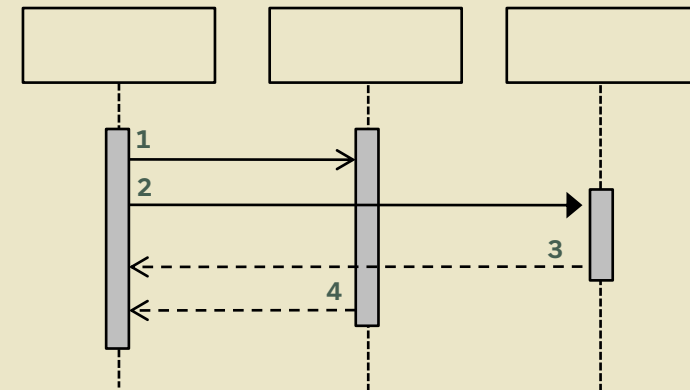


Paso Asíncrono de Mensajes

Se trataría de:

1. Pasar el primer mensaje `GetNumeroImagenes` de un modo **asíncrono** creando un nuevo hilo
2. Pasar el segundo `GetNumeroImagenes` mensaje de un modo **síncrono** para obtener su valor en el hilo principal
3. Obtener el número de imágenes del segundo mensaje (la espera está implícita al ser síncrono)
4. Tomar el número de imágenes del primer mensaje (habría que esperar a que acabase, al ser asíncrono)
5. Mostrar los resultados

¿Cómo podríamos implementar esto?



Paso Asíncrono de Mensajes

- Con el modelo de paso de mensajes asíncrono (*Asynchronous Programming Model* o *APM*) utilizaba llamadas `BeginInvoke`, `EndInvoke`, la interfaz `IAAsyncResult` y **delegados** (funciones de *callback*)
 - Son menos intuitivas
 - Modelo obsoleto. Solo .NET Framework (3.5+)
 - `PlatformNotSupportedException`
- El nuevo modelo de mensajes asíncronos se basa en las palabras clave **`async`**, **`await`** y objetos **`Task`**
 - Permiten escribir código asíncrono de manera fácil e intuitiva
 - Próxima la sintaxis utilizada para escribir código síncrono (invocación a un método)

Consulta el código en:

`old.delegates`

Consulta el código en:

`asynchronous.calls`

async y await

- Nuevas palabras reservadas
 - El lenguaje está diseñado para soportarlas
- Se apoya en el uso de objetos `Task` y `Task<T>` ([ver apartado "Tareas"](#))
- Modifican el orden habitual de ejecución del programa
- **Simplicidad**: el código asíncrono es muy parecido al código síncrono
- **await**
 - Se aplica sobre una expresión sobre la que se puede esperar: `expression.GetAwaiter()`
 - Normalmente expresiones de tipo `Task` y `Task<T>`
- **async**
 - Se aplica a métodos que en su cuerpo usan `await`
 - De lo contrario se comporta como un método síncrono y al compilar produce un Warning.
 - El método devolverá `void`, `Task` o `Task<T>`

async y await: Ejemplo básico

```
public static async Task AsynchronousTask() {  
    //Do something  
    await Task.Run(() => Thread.Sleep(2000));  
    Console.WriteLine("AsynchronousTask body");  
    return;  
}
```

2 – Se ejecuta el cuerpo hasta llegar a una expresión `await`

3 – Se evalúa `await` y en paralelo se devuelve la ejecución al invocador (4)

4b – Si la expresión `await` ha terminado se ejecuta el resto del método

```
static void Main() {  
    ...  
    var t = AsynchronousTask();  
    //This simulates more computations  
    Thread.Sleep(100);  
    Console.WriteLine("AsynchronousTask has ended");  
    t.Wait(); //Synchronization mechanism  
    ...  
}
```

1 – El método se invoca normalmente

4 – Se continúa ejecutando código no dependiente del resultado de la invocación asíncrona

5 – En este punto se necesita el resultado de la llamada asíncrona.

Si ya ha terminado continúa la ejecución
Si no, se duerme hasta que termine

<https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/task-asynchronous-programming-model>

Ejemplo de uso

```
public async Task<int> GetNumberOfImagesAsyncTask()
{
    var client = new WebClient();
    //Call an asynchronous method, waits for this to finish and yields control to the caller
    var html = await Task.Run(() => client.DownloadString(url));
    return Ocurrences(html.ToLower(), "<img");
}

public static void Main() {
    var uniovi = new WebPage("http://www.uniovi.es");
    var school = new WebPage("http://www.ingenieriainformatica.uniovi.es");
    ...
    Task<int> taskGetImagesUniovi = uniovi.GetNumberOfImagesAsyncTask();
    //Synchronous call in the main thread
    int numberOfImgsInSchool = school.GetNumberOfImages();
    Console.WriteLine("Synchronous work finished");
    //Wait for the asynchronous task to finish
    int numberOfImgsInUniovi = taskGetImagesUniovi.Result;
    ...
}
```


Futuros y Promesas

- Aunque **parece sencillo** de utilizar, el código resultante:
 - Modifica la normal ejecución del programa, modifica el manejo de excepciones, etc.
 - Puede ser difícil de comprender
 - En estas transparencias **se pretende introducir su uso**, pero no se profundiza en todos los detalles y problemas de su uso.
- Otros lenguajes utilizan conceptos similares, ya sea a través de las palabras reservadas **async** y **await**, o mediante **futuros** (valor devuelto) y **promesas** (función asíncrona).
- En Java 5 se introdujo la interfaz `Future` que permiten realizar tareas asíncronas devolviendo una instancia de `FutureTask`.



Hilos (Threads)

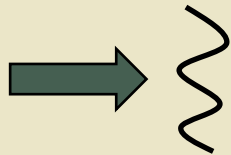
Elemento de bajo nivel para escribir código paralelo

Creación Explícita de Hilos

- Haciendo uso del paradigma Orientado a Objetos, la clase **Thread** (**System.Threading**) encapsula un hilo de ejecución de forma explícita
 - Ofrecen otro mecanismo para la creación de aplicaciones multihilo

Hilo Principal

```
// Lo creamos
Thread hilo = new Thread(deLegado);
// Lo nombramos (opcional)
hilo.Name = "Hilo secundario";
Thread.CurrentThread.Name = "Hilo principal";
// Le asignamos una prioridad (opcional)
hilo.Priority = ThreadPriority.BelowNormal;
// Lo lanzamos
hilo.Start();
...
```



Hilo Secundario

Pregunta: ¿Cuál es la principal diferencia con Java?

Ejemplo

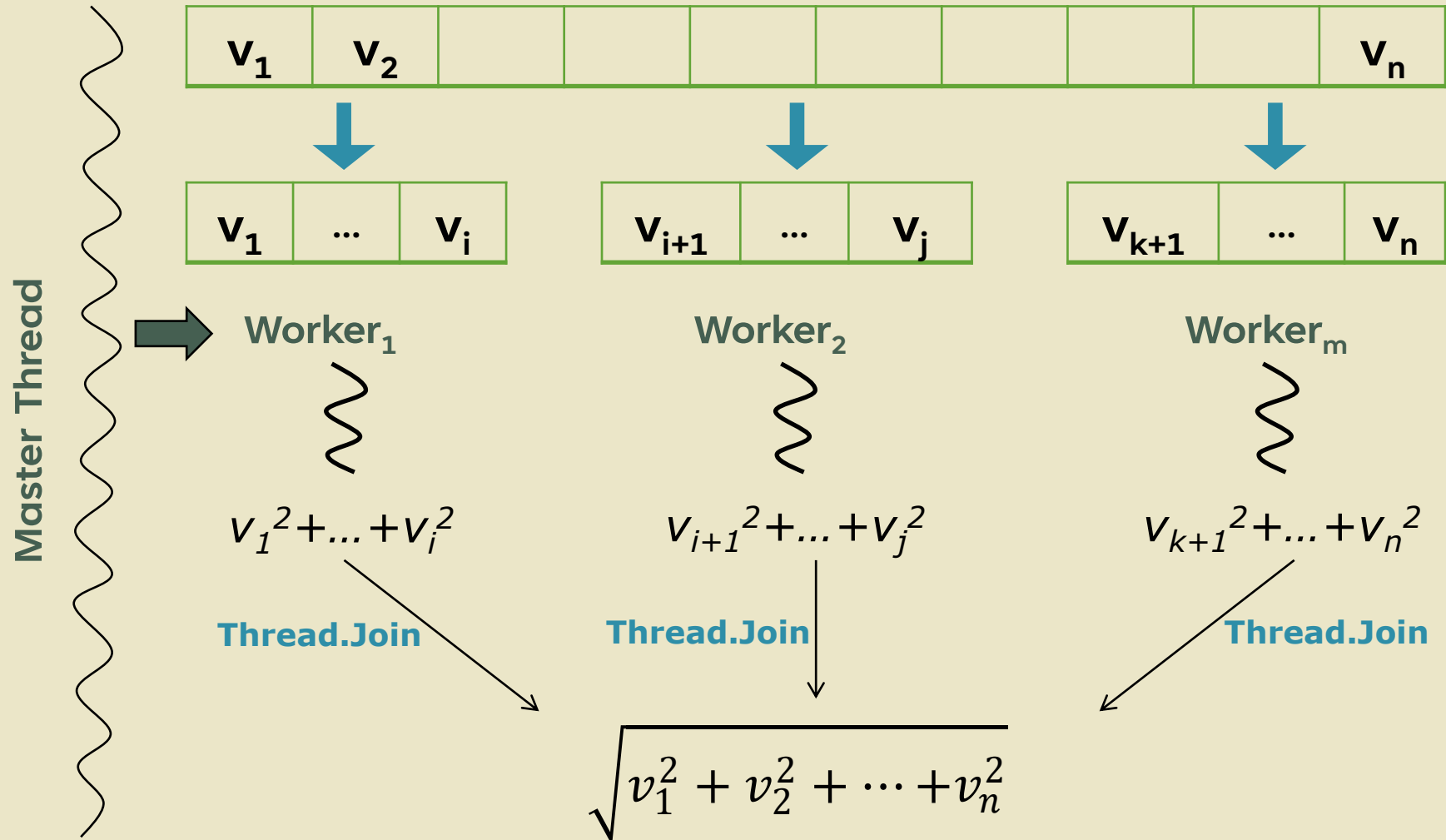
- A continuación, veremos un **ejemplo de programa concurrente** donde se aplica la paralelización por división de datos
 - Los datos se dividen en partes, y distintos *threads* ejecutan el mismo algoritmo para procesar concurrentemente cada una de esas partes
- El problema que resuelve es el cálculo del **módulo de un vector** algebraico de n dimensiones

$$|\vec{v}| = \sqrt{v_1^2 + v_2^2 + \dots + v_n^2}$$

Consulta el código en:

`threads/vector.modulus`

Ejemplo



Ejemplo: Worker

```
internal class Worker {  
    private short[] vector;  
    private int índiceDesde, índiceHasta;  
    private long resultado;  
    internal long Resultado { get { return this.resultado; } }  
    internal Worker(short[] vector, int índiceDesde,  
                    int índiceHasta) {  
        this.vector = vector;  
        this.índiceDesde = índiceDesde;  
        this.índiceHasta = índiceHasta;  
    }  
    internal void Calcular() {  
        this.resultado = 0;  
        for(int i= this.índiceDesde; i<=this.índiceHasta; i++)  
            this.resultado += this.vector[i] * this.vector[i];  
    } } }
```

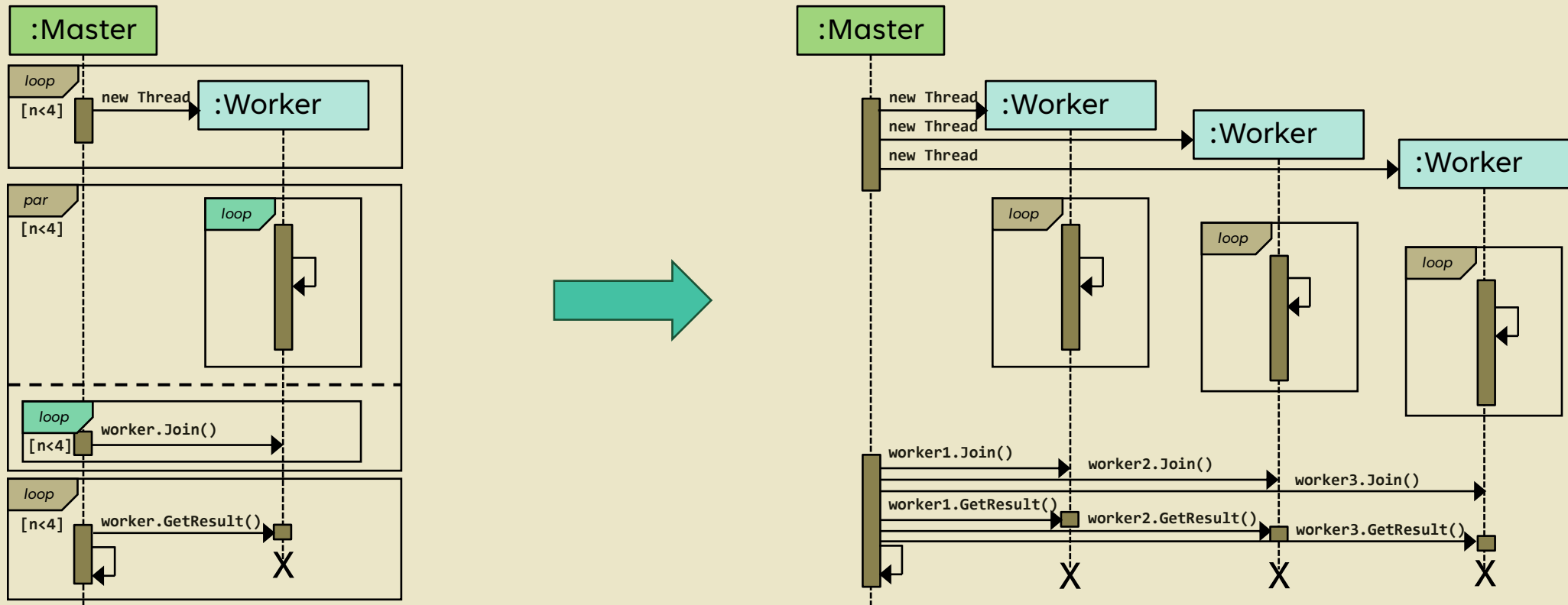
Ejemplo: Master (I)

```
public class Master {  
    private short[] vector;  
    private int numeroHilos;  
    public Master(short[] vector, int numeroHilos) {  
        if (numeroHilos < 1 || numeroHilos > vector.Length)  
            throw new ArgumentException("El número de hilos ha  
                de ser menor o igual que los elementos del  
                vector.");  
        this.vector = vector;  
        this.numeroHilos = numeroHilos;  
    }  
}
```

Ejemplo: Master (II)

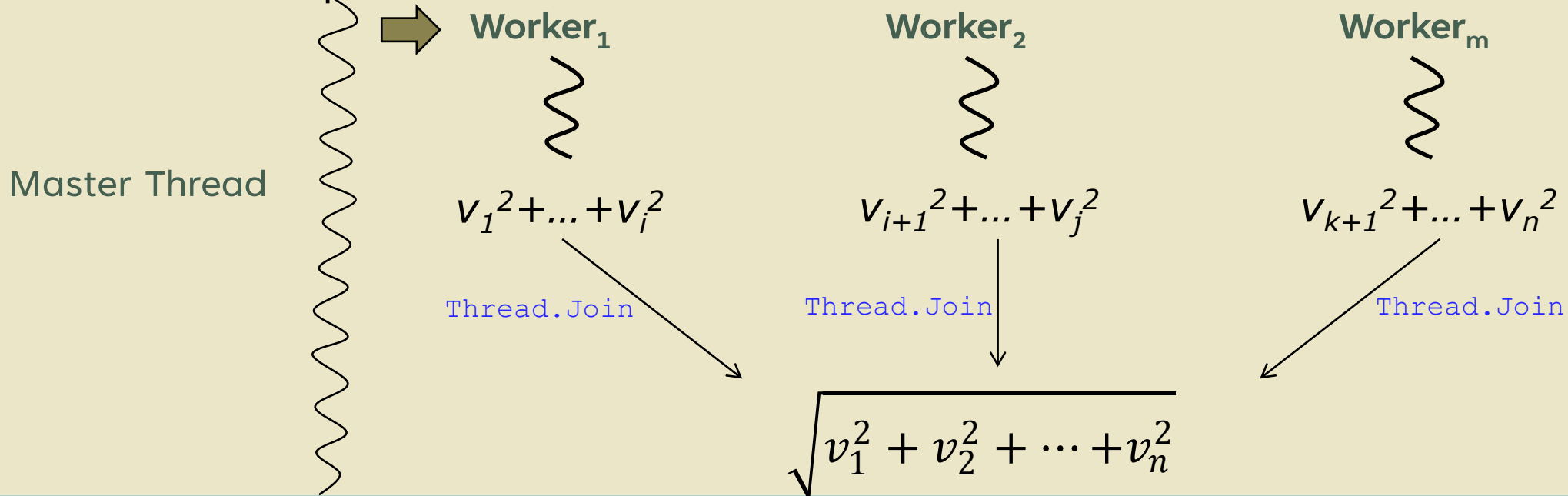
```
public double CalcularModulo() {  
    Worker[] workers = new Worker[this.numeroHilos];  
    int elementosPorHilo = this.vector.Length/numeroHilos;  
    for(int i=0; i < this.numeroHilos; i++)  
        workers[i] = new Worker(this.vector, i*elementosPorHilo,  
                                (i<this.numeroHilos-1) ? // ¿último?  
                                (i+1)*elementosPorHilo-1: this.vector.Length-1 );  
    Thread[] hilos = new Thread[workers.Length];  
    for(int i=0;i<workers.Length;i++) {  
        hilos[i] = new Thread(workers[i].Calcular);  
        hilos[i].Start();  
    }  
    foreach (Thread hilo in hilos) hilo.Join();  
    long resultado = 0;  
    foreach (Worker worker in workers)  
        resultado += worker.Resultado;  
    return Math.Sqrt(resultado); } }
```


Ejemplo: Diagrama de secuencia



Thread.Join

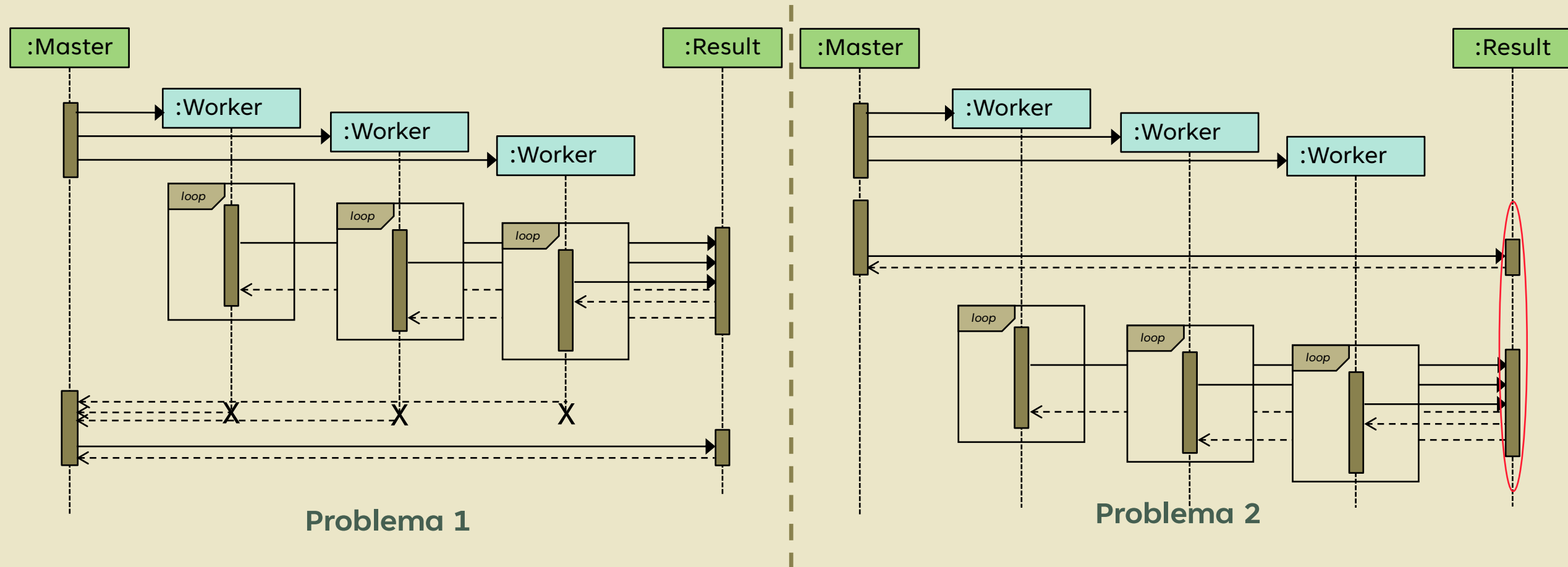
- En nuestro ejemplo, utilizamos el mensaje **Join** de Thread en el hilo principal
- Cuando se llama a **Join**, el hilo que realiza la llamada se bloquea (*duerme*) hasta que finaliza la ejecución del *Thread* que recibió el mensaje
- Es un primer mecanismo de sincronización de hilos



Condición de Carrera

- ¿Qué pasaría si no hubiésemos puesto la llamada a **Thread.Join**?
- El hilo master tomaría los resultados del cálculo (posiblemente) **antes** de que hubiesen acabado los hilos worker de hacer el cómputo
 - El cálculo final varía de una ejecución a otra
- Se dice que múltiples tareas están en **condición de carrera** (*race condition*) cuando su resultado depende del orden en el que éstas se ejecutan
 - Un programa concurrente **no** debe tener condiciones de carrera
- Las condiciones de carrera son un **foco de errores** en programas y sistemas concurrentes

Condición de Carrera: Diagrama Secuencia



Parámetros

- Hemos utilizado la orientación a objetos como paradigma para **encapsular** los datos de un hilo en un objeto
- Si se prefiere utilizar una aproximación más funcional, se pueden pasar parámetros a los hilos

```
static void Mostar10Numero(object desde) {  
    if (!(desde is int))  
        throw new ArgumentException("...");  
    int desdeInt = (int)desde;  
    for (int i = desdeInt; i < 10 + desdeInt; i++) {  
        Console.WriteLine(i); Thread.Sleep(1000);  
    }  
}  
static void Main() {  
    Thread hilo = new Thread(Mostar10Numero);  
    hilo.Start(7); hilo.Join();  
}
```

Consulta el código en:

[*threads/parameters*](#)

Variables Libres (*free*)

- Si se usan funciones **lambda**, hay que tener cuidado con sus variables libres
- Recordemos que cada hilo posee una copia de la pila de ejecución... **¡a partir del ámbito en el que se creó!** (el resto del stack se comparte con el thread que creo el nuevo)
 - Las **variables locales ya declaradas serán compartidas** por todos los hilos

```
int local = 1;
Thread hilo1 = new Thread( () => {
    Console.WriteLine("Hilo 1. Local {0}.", local);
});
local = 2;
Thread hilo2 = new Thread( () => {
    Console.WriteLine("Hilo 2. Local {0}.", local);
});
hilo1.Start(); hilo2.Start();
```

Consulta el código en:

[*threads/bound.variables*](#)

Alternativas

- **Paso de parámetros** (preferible)

```
int local = 1;
Thread hilo1 = new Thread( (parametro) => {
    Console.WriteLine("Parámetro {0}.", parametro);
});
local = 2;
hilo1.Start(local-1);
```

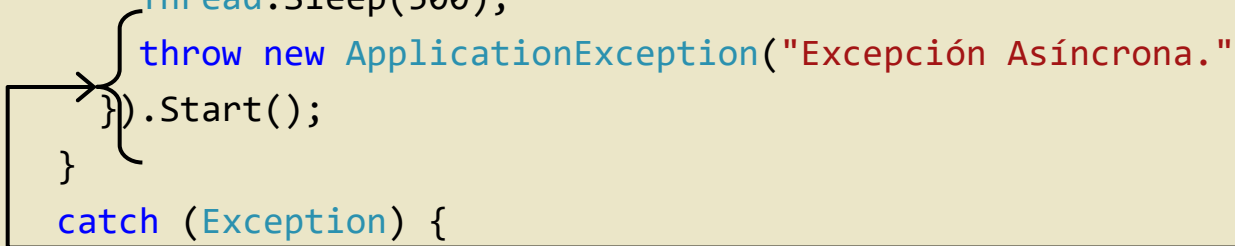
- **Copia de variables**

```
int local = 1;
int copia = local;
Thread hilo1 = new Thread( () => {
    Console.WriteLine("Copia {0}.", copia);
});
local = 2;
hilo1.Start();
```

Excepciones Asíncronas

- ¿Qué sucede cuando un hilo que hemos creado lanza una excepción no capturada?

```
static void Main() {  
    try {  
        new Thread(() => {  
            Thread.Sleep(500);  
            throw new ApplicationException("Excepción Asíncrona.");  
        }).Start();  
    }  
    catch (Exception) {  
        Console.WriteLine("Se captura la excepción.");  
    }  
    Thread.Sleep(1000);  
    Console.WriteLine("Fin del programa.");  
}
```



¡La excepción no es capturada y el programa finaliza bruscamente!

El try/catch se debería haber puesto en el código asíncrono

Este catch es inútil

Consulta el código en:

[threads/asynchronous.exceptions](https://docs.microsoft.com/en-us/dotnet/threads/asynchronous.exceptions)

Mejora de Rendimiento

- Los siguientes son datos de ejecución del cálculo del módulo del vector en un ordenador con **cuatro núcleos** y 8GBytes de RAM
- Se utilizó un vector de 100.000 elementos aleatorios comprendidos en $[-10,10]$
 - Ejecución con 1 hilo *worker*: 30 milisegundos
 - Ejecución con 4 hilos *workers*: 10 milisegundos
- ¿Se reduce el tiempo de ejecución si aumentamos el número de hilos?

Context Switch

- El **contexto** de una tarea (hilo o proceso) es la información que tiene que ser guardada cuando éste es interrumpido para que luego pueda reanudarse su ejecución
- El **cambio de contexto** (*context switch*) es la acción de almacenar/restaurar el contexto de una tarea (hilo o proceso) para que pueda ser reanudada su ejecución
- Esto permite la ejecución concurrente de varias tareas en un mismo procesador

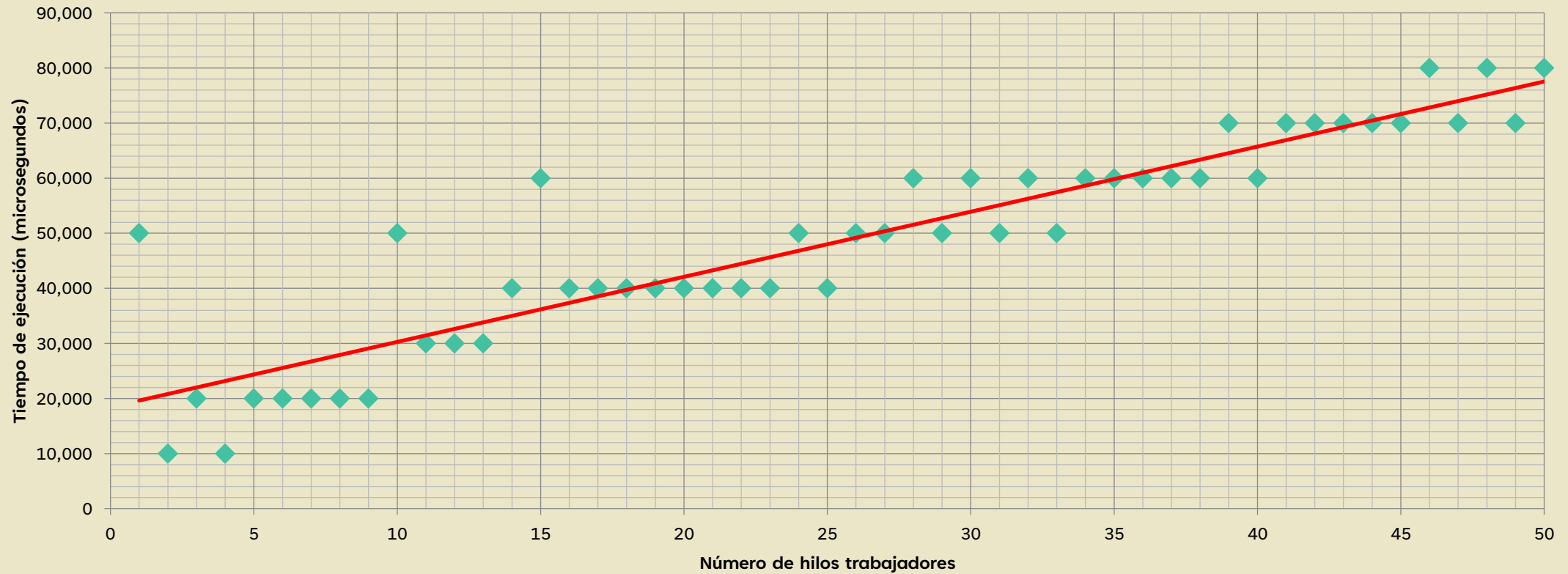
Context Switch

Consulta el código en:

`threads/context.switching`

- El cambio de contexto requiere
 - **Tiempo de computación** para almacenar y restaurar el contexto de varias tareas
 - **Memoria adicional** para almacenar los distintos contextos
- Por tanto, la utilización de **un número elevado de tareas**, en relación con el número de procesadores (cores), **puede conllevar una caída global del rendimiento**
- Hemos ejecutado nuestra aplicación de cálculo del módulo del vector con [1,50] hilos *workers*, y éstos han sido los resultados:

Context Switch

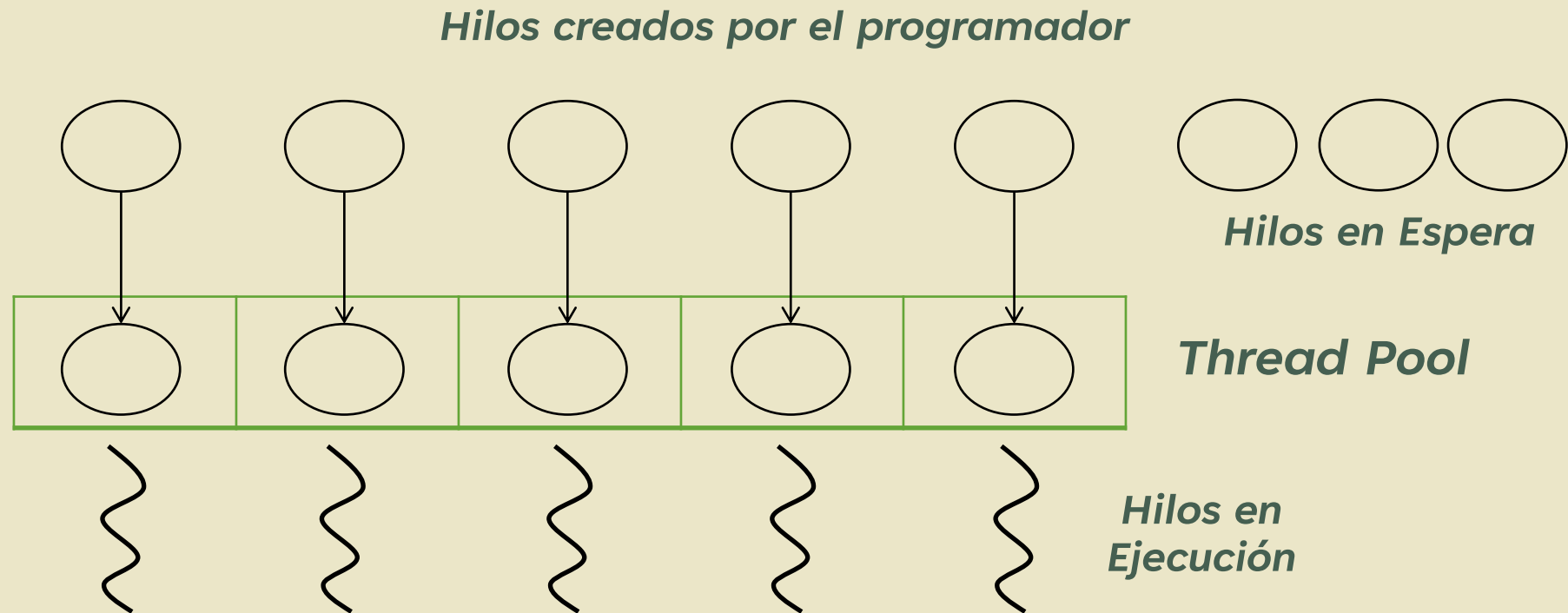


- Con **2 hilos** (y con 4) se obtiene el **mejor rendimiento**
- A partir de 28 hilos, el rendimiento de la aplicación **decae** frente a un algoritmo secuencial
- Cuando se usan más de 9 hilos, el rendimiento **decae linealmente** frente al número de hilos (línea roja)

Thread Pooling

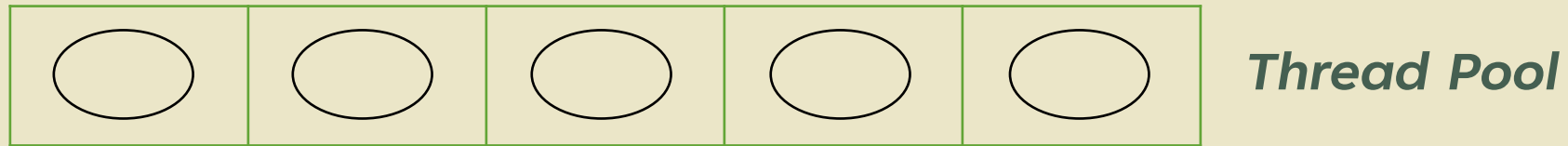
- No sólo el cambio de contexto implica un coste computacional en aplicaciones concurrentes
- La **creación y destrucción de hilos** es un proceso que **también implica un coste** computacional y de memoria
- Por todo ello, se debe:
 1. **Limitar el número máximo de hilos** creados por un proceso, en relación con el número de procesadores y otros recursos la memoria
 2. Minimizar el número de hilos creados (**reutilizarlos**)
- Para conseguir estos objetivos, el CLI ofrece un mecanismo de **Thread Pooling** que optimiza el número máximo de hilos concurrentes por procesador
 - Por ejemplo, en el CLR 2.0 se permiten 25 hilos concurrentes por procesador
- Esta técnica también se utiliza con conexiones a BBDD (*connection pooling*)

Thread Pooling



Thread Pooling

- Si el proceso lleva un tiempo en estado sin crear nuevos hilos...



- El thread pool manager elimina los hilos del pool para ahorrar memoria

Foreground & Background Threads

- Los hilos que hemos creado ahora con la clase **Thread** han sido hilos *foreground*:
 - La aplicación no finalizará hasta que acabe la ejecución de todos los hilos *foreground* creados
- Un hilo *background* (también llamado *daemon*) es aquél que será terminado (*killed*) cuando no queden hilos *foreground* en ejecución
 - Normalmente son proveedores de servicios
 - No confundir con hilos secundarios o *workers*

Consulta el código en:

threads/daemon

Foreground & Background Threads

```
static void Main() {  
    Thread background = new Thread(() => {  
        int segundos = 0;  
        while (true) {  
            Thread.Sleep(1000);  
            Console.WriteLine("\t\t\t\t\t\t\t\t\t\t\t{0} segundos.",  
                              ++segundos);  
        } });  
  
    background.IsBackground = true; // es un daemon  
    background.Start();  
  
    Thread foreground = new Thread(() => {  
        for (int i = 0; i < 100; i++) {  
            Console.WriteLine("Iteración {0}.", i + 1);  
            Thread.Sleep(100);  
        } });  
  
    foreground.Start();  
}
```

Inconvenientes del uso de Hilos

- **Condiciones de carrera:** Debemos esperar explícitamente (`Join`) hasta que todos los hilos han terminado de realizar sus cálculos
- **Parámetros:** sin parámetros o solo un objeto, variables libres compartidas
- **Excepciones asíncronas:** Las excepciones originadas en un hilo no son capturadas por bloques `try-catch` pertenecientes a un hilo diferente.
 - ¡Incluso si ese hilo se ha creado dentro de ese bloque `try-catch`!
- **Rendimiento de los cambios de contexto:** No hay optimización automática del número de hilos creados
 - ¡Si el número de hilos creados es demasiado elevado no se consigue mejorar el rendimiento del programa e incluso se empeora!



Tasks

Subiendo el nivel de abstracción para crear código que se ejecuta en paralelo

Factores a Tener en Cuenta

- Hasta ahora hemos hablado de la creación, utilización y sincronización de hilos, **apenas hemos tenido en cuenta**
 - Si el número de hilos creados mejora el rendimiento global de la aplicación
 - Cuántos hilos crear en función del *thread pool*
 - El número de procesadores o cores existentes
- Además, el hilo es un elemento **de muy bajo nivel**
 - Crear uno implica que el código se va a ejecutar forzosamente en paralelo y consume los recursos para ello
 - **¡Pero impide que el CLR haga una serie de optimizaciones!**

Tareas (Task)

- Varios de los problemas que hemos visto con hilos se resuelven mediante una nueva abstracción la tarea o `Task`
- Tienen un nivel de abstracción mayor y proporcionan más funcionalidades que los hilos, facilitando la programación paralela
 - TPL, PLINQ y otros elementos del lenguaje que facilitan crear código paralelo se basan internamente en ellas
 - Es importante pues conocerlas puesto que la última parte de este tema trata de TPL y PLINQ
 - **Más información:**
 - <https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-based-asynchronous-programming>
 - <https://blogs.encamina.com/piensa-en-software-desarrolla-en-colores/asincronia-en-c-bloqueos-contextos-y-tareas/>
- Una aproximación para gestionar ejecución concurrente de código similar a la de las tareas existe en **Java 8** con los `Executors` y el `ExecutorService`:
 - <https://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html>
 - <https://www.baeldung.com/java-executor-service-tutorial>

Tareas (Task)

- Una Task **representa una operación asíncrona** y su uso tiene dos beneficios principales
 - **Uso más eficiente y escalable de recursos:** las Task se encolan automáticamente en el `ThreadPool`, que optimiza transparentemente el nº real de hilos usados y hace balanceo de carga para maximizar el uso de los recursos del sistema
 - Por tanto, el nº óptimo de hilos a usar en la máquina que esté ejecutando el proceso ¡se determina automáticamente!
 - **Mayor control de ejecución** (comparado con `Thread`): el API de Task soporta espera, cancelación, continuación, manejo robusto de excepciones, consulta detallada de estado, planificación y más características
- En **.NET**, Task y TPL son las elegidas normalmente para escribir código multihilo, asíncrono o paralelo

Composición de tareas

- `Task` y `Task<TResult>` tienen varios métodos para **componer tareas**
- Esto permite implementar patrones típicos y mejorar el uso de las capacidades asíncronas del lenguaje
 - **`Task.WhenAll`**: espera de forma asíncrona a que terminen varios objetos `Task` o `Task<TResult>`
 - **`Task.WhenAny`**: espera de forma asíncrona a que una o varios objetos `Task` o `Task<TResult>` terminen
 - **`Task.Delay`**: crea un objeto `Task` que acaba tras un tiempo determinado
- Para **más información** consultar:
 - <https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/chaining-tasks-by-using-continuation-tasks>

Manejo de excepciones con tareas

- Cuando una tarea lanza una o más excepciones, **todas ellas se encapsulan** en una excepción de tipo `AggregateException`
- Esta excepción **se propaga al hilo vinculado a la tarea**, que normalmente es el que espera a que termine o bien el que accede a su propiedad `Result`
- El código que llame a uno de los siguientes métodos de la tarea obtendrá las excepciones que se produzcan en la ejecución de la misma y podrá tratarlas normalmente en un bloque `try/catch`:
 - `Wait`
 - `WaitAll`
 - `WaitAny`
 - `Result`
- También se pueden tratar las excepciones de una tarea accediendo a la propiedad `Exception` de la misma antes de que sea eliminada por el recolector de basura

Manejo de excepciones con tareas

```
static void CaptureException(){
    try {
        var task = Task.Run(() => { throw new ArgumentNullException(); });
        task.Wait();
    }
    catch (AggregateException e) {
        Console.WriteLine("Task lanza la siguiente excepcion: " + e);
    }
    return;
}
```

```
static void ReThrowException (){
    try {
        var task = Task.Run(() => { throw new ArgumentNullException(); });
        task.Wait();
    }
    catch (AggregateException e) {
        Exception[] list = new Exception[] { e };
        throw new AggregateException("Excepcion relanzada como una
AggregateException", list);
    }
    return;
}
```

Consulta el código en:

[tasks/task.exception](#)



Mecanismos de sincronización

Qué hacer cuando varios hilos utilizan concurrentemente un mismo recurso o estructura de datos

Sincronización de Hilos

- En ocasiones, varios hilos tienen que colaborar entre sí para conseguir un objetivo común
- Puesto que el orden de ejecución es no determinista, es necesario utilizar mecanismos de **sincronización de hilos** para evitar condiciones de carrera
- Un primer mecanismo básico de sincronización de hilos que ya hemos utilizado es **Thread.Join**
 - Permite hacer que un hilo espere a la finalización de otro
- No obstante, la necesidad más típica de sincronización de hilos es por acceso concurrente a **recursos compartidos**
 - Un **recurso compartido** puede ser un dispositivo físico (impresora), lógico (fichero), una estructura de datos, un objeto e, incluso, una variable
- Evitar el uso simultáneo de un recurso común se denomina **exclusión mutua**
- **Esto se aplica a aplicaciones con varios hilos, tanto si se implementan con creación explícita de hilos como si utilizan Tasks**

Ejemplo

- ¿Qué podría suceder si se ejecuta el método **Mostrar** con varios hilos concurrentes?

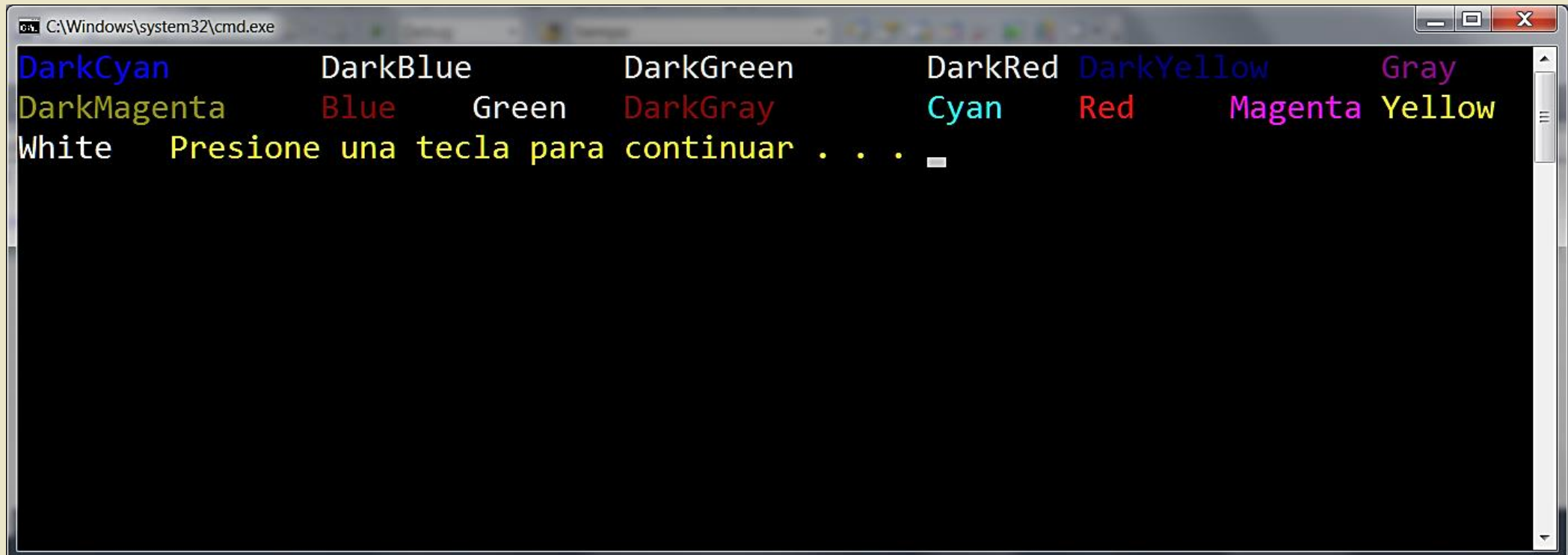
```
public class Color {  
    private ConsoleColor color;  
    public Color(ConsoleColor color) {  
        this.color = color;  
    }  
    virtual public void Mostrar() {  
        ConsoleColor colorAnterior = Console.ForegroundColor;  
        Console.ForegroundColor = this.color;  
        Console.Write("{0}\t", this.color);  
        Console.ForegroundColor = colorAnterior;  
    }  
}
```

Ejemplo

Consulta el código en:

synchronization/synchronized.colors

- Ejemplo de ejecución:



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window displays a color synchronization test. The first line shows six color names: "DarkCyan", "DarkBlue", "DarkGreen", "DarkRed", "DarkYellow", and "Gray", each followed by a space. The second line shows six more color names: "DarkMagenta", "Blue", "Green", "DarkGray", "Cyan", and "Red", each followed by a space. The third line shows the text "White" followed by "Presione una tecla para continuar . . .". The text is displayed in various colors, including dark blue, dark green, dark red, dark yellow, dark magenta, and gray, demonstrating the synchronization of the text color with the background color.

Recurso Compartido

- En el código anterior, el **recurso compartido** es la salida estándar de la **consola**
- El hecho de no proteger su acceso, hace que las instrucciones:

```
ConsoleColor colorAnterior = Console.ForegroundColor;
```

```
Console.ForegroundColor = this.color;
```

```
Console.Write("{0}\t", this.color);
```

```
Console.ForegroundColor = colorAnterior;
```

No se ejecuten de forma **atómica**

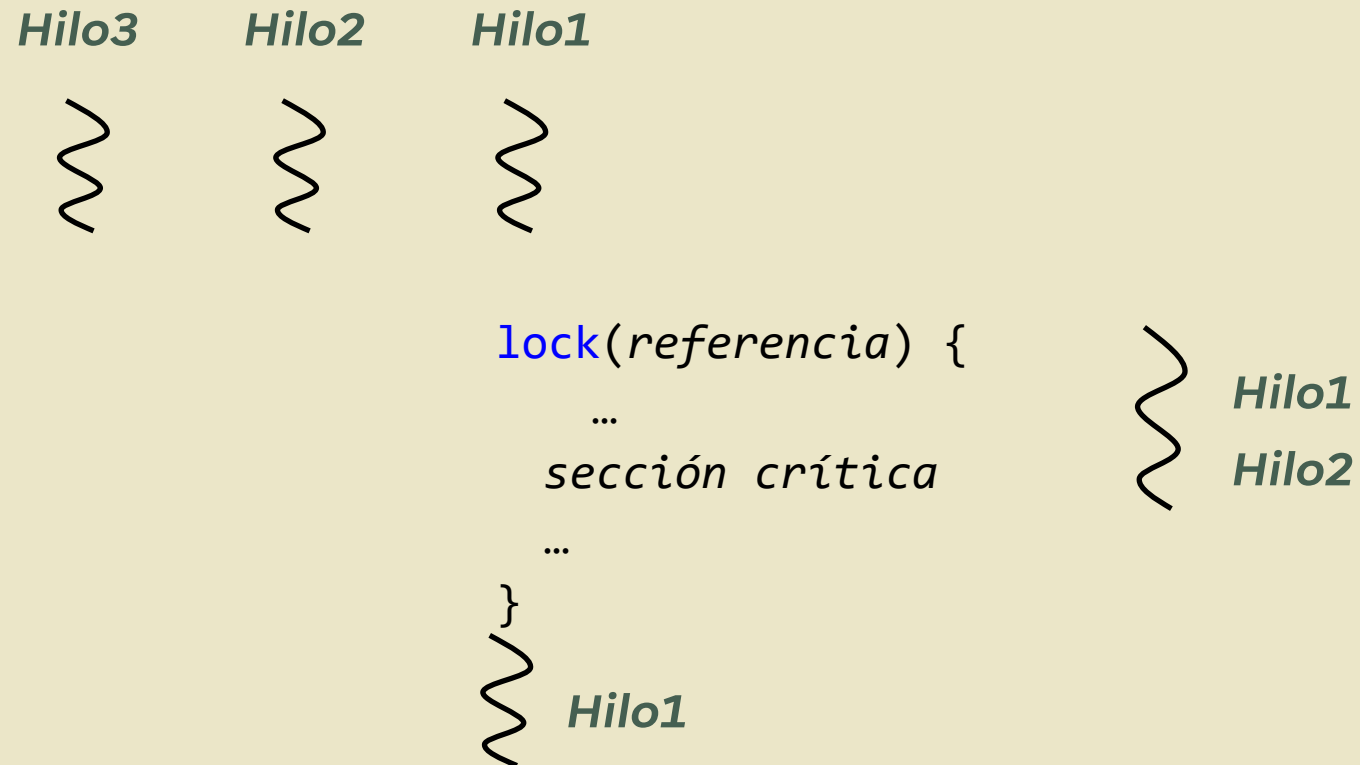
- Una **sección crítica** es un fragmento de **código** que **accede** a un **recurso compartido** que no debe ser accedido concurrentemente por más de un hilo de ejecución
- La **sincronización de hilos** debe usarse para conseguir la **exclusión mutua**

Lock

- La principal técnica utilizada para sincronizar hilos en C# es la palabra reservada **lock**
- Consigue que **únicamente un hilo** pueda ejecutar una sección de código (sección crítica) simultáneamente \Rightarrow **exclusion mútua**
- **lock** requiere especificar un objeto (referencia) como parámetro

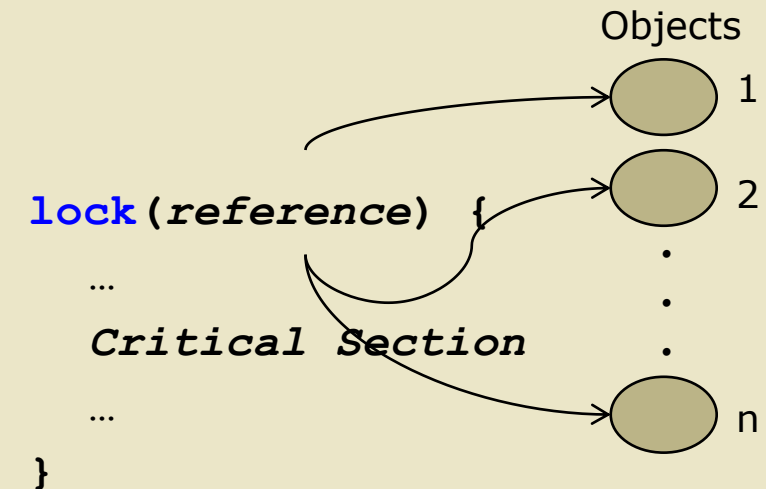
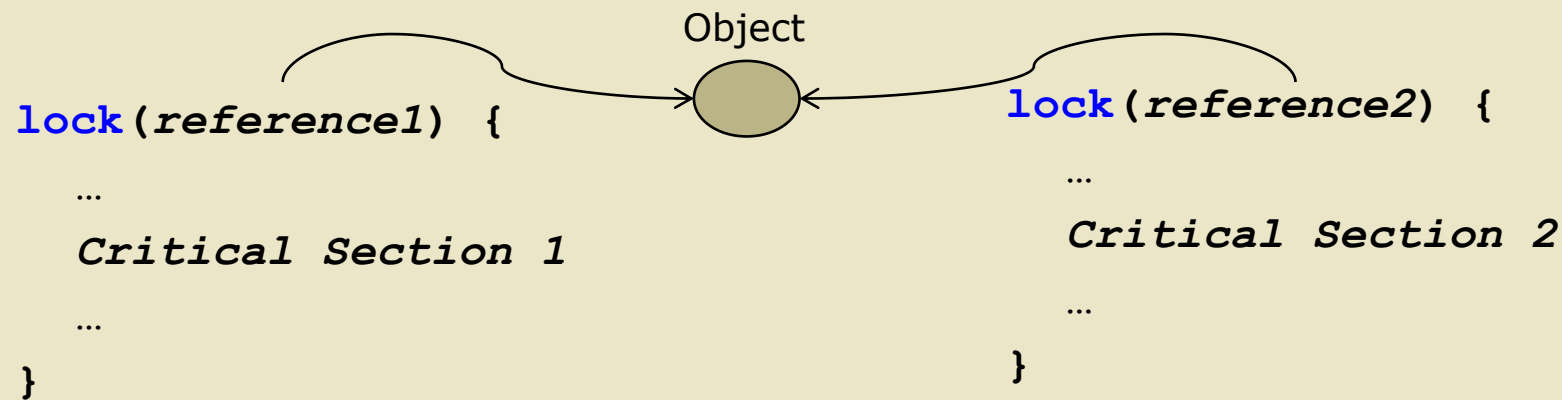
```
lock(referencia) {  
    sección crítica  
}
```
- El objeto modela un **padlock**: 1) Un hilo bloquea el objeto 2) ejecuta la sección crítica y 3) sale del bloqueo
- Si otro hilo ejecuta el *lock* sobre un objeto que ya está bloqueado, entonces se pondrá **en modo de espera** y se **bloqueará** hasta que el objeto sea liberado

Lock



Lock

- **Nota:** El bloqueo se hace considerando el objeto en tiempo de ejecución (no el fragmento de código bloqueado)
- Ejemplo 1: Dos fragmentos de código mutuamente excluyente con el mismo objeto de bloqueo
- Ejemplo 2: El mismo fragmento de código que puede ser ejecutado concurrentemente por N hilos



Ejemplo

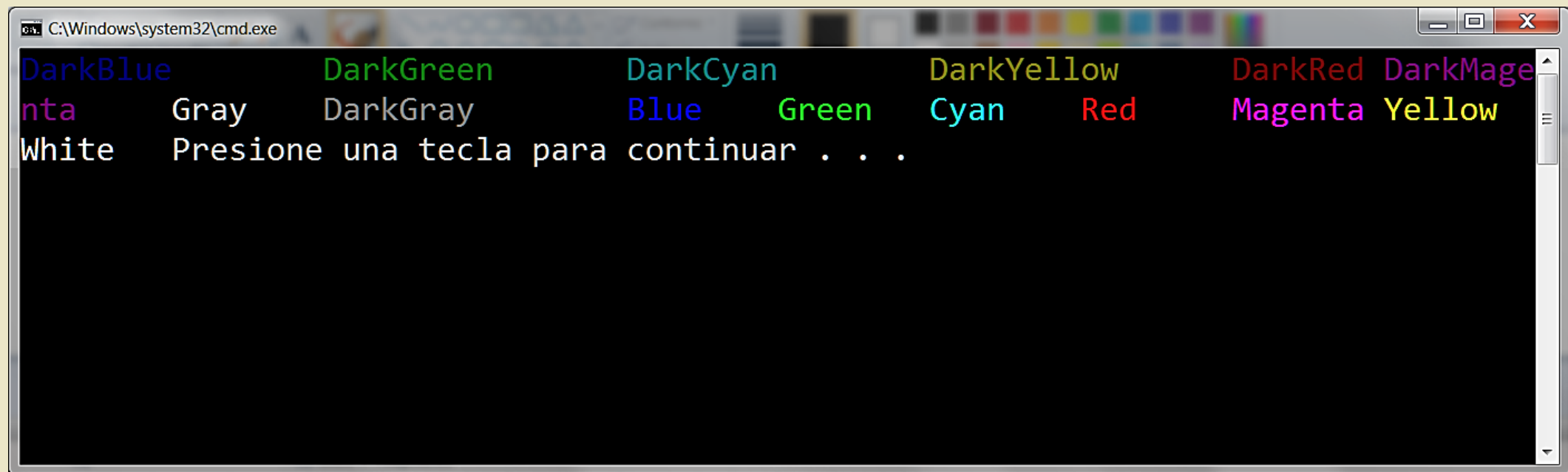
- ¿Qué objeto utilizaremos en nuestro ejemplo para sincronizar el acceso a la salida estándar?

```
public class ColorSincronizado: Color {  
    public ColorSincronizado(ConsoleColor color)  
        : base(color) {  
    }  
    override public void Mostrar() {  
        lock (Console.Out) {  
            base.Mostrar();  
        }  
    }  
}
```

Ejemplo

Consulta el código en:

synchronization/synchronized.colors



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window displays a list of color names in various colors. The first line shows "DarkBlue", "DarkGreen", "DarkCyan", "DarkYellow", "DarkRed", and "DarkMagenta". The second line shows "Magenta", "Gray", "DarkGray", "Blue", "Green", "Cyan", "Red", "Magenta", and "Yellow". The third line shows "White" and the text "Presione una tecla para continuar . . .".

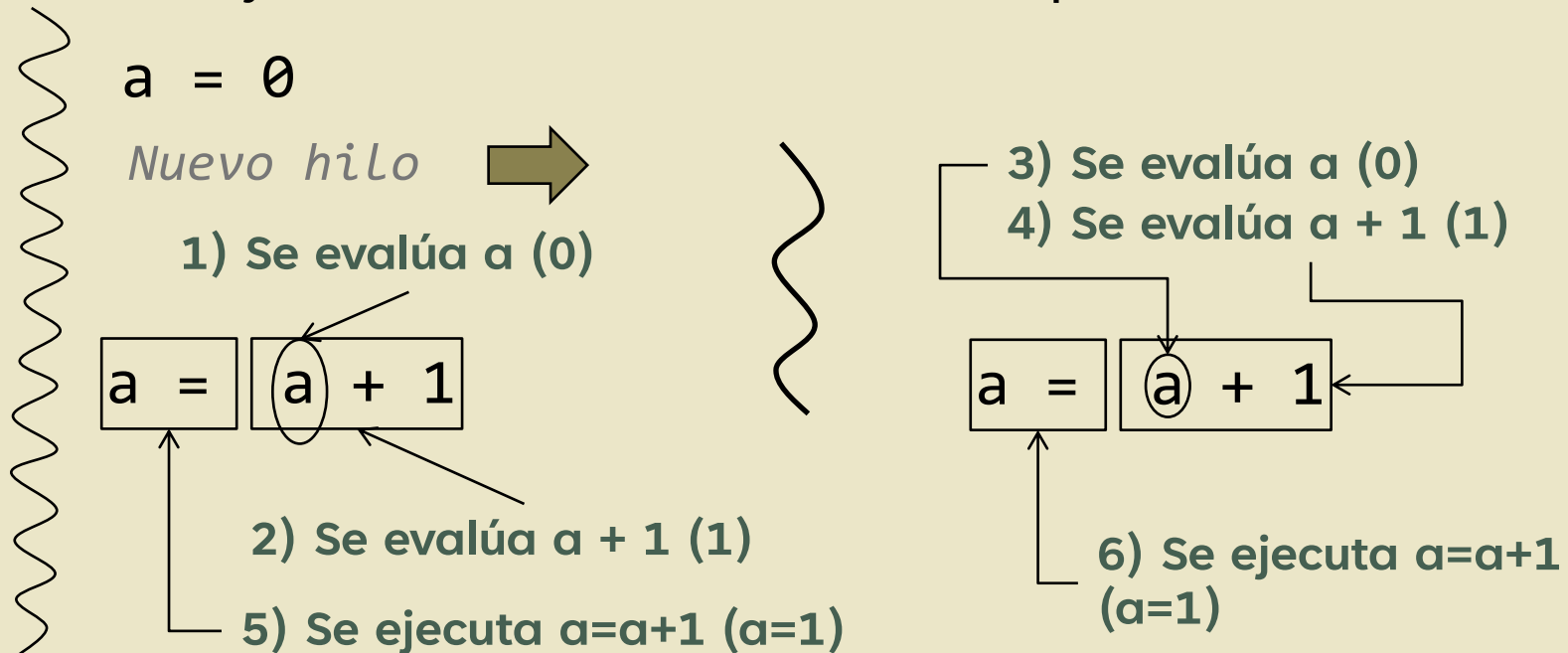
```
C:\Windows\system32\cmd.exe
DarkBlue      DarkGreen     DarkCyan     DarkYellow   DarkRed DarkMagenta
Magenta      Gray      DarkGray     Blue      Green      Cyan      Red      Magenta Yellow
White Presione una tecla para continuar . . .
```

Asignaciones

- Hemos visto cómo la escritura en recursos compartidos requiere sincronización
- ¿**También** sucede esto en las **asignaciones** de **variables** y **atributos**? Sí
- No todas las asignaciones son atómicas en .NET
 - Las asignaciones de 32 bits son atómicas
 - Las asignaciones de 64 bits (**long**, **ulong**, **double**, **decimal**) no son atómicas en un SO de 32 bits
 - Los operadores **+= -= *= /= %= &= |= ^= <<= >>=** no son atómicos
 - Los operadores **++ --** no son atómicos

Asignaciones

- Analicemos la ejecución concurrente de la expresión $a = a + 1$



- Al final de la ejecución, ¡ a vale 1 en lugar de 2 !

Asignaciones

- Por tanto, las **asignaciones multihilo** de una misma variable **deben sincronizarse**
- Una alternativa es utilizar **lock**
 - Esta alternativa bloquea hilos en ejecución (exclusión mútua)
- Otra alternativa es utilizar los métodos de la clase **Interlocked** (**System.Threading**)
 - Esta alternativa hace que el CLR haga una asignación de un modo primitivo
 - Es (mucho) más eficiente que utilizar **lock**
 - Los métodos más utilizados son **Increment**, **Decrement** y **Exchange**

Ejemplo

- ¿Qué valor muestra la siguiente ejecución?

```
static long valor = 100000000;  
static void Main() {  
    const int numeroHilos = 10000;  
    int iteraciones = (int)valor / numeroHilos;  
    Thread[] hilos = new Thread[numeroHilos];  
    for (int i = 0; i < numeroHilos; i++)  
        hilos[i] = new Thread(() => {  
            for (int j = 0; j < iteraciones; j++)  
                valor=valor-1;  
        });  
    foreach (Thread hilo in hilos) hilo.Start();  
    foreach (Thread hilo in hilos) hilo.Join();  
    Console.WriteLine(valor);  
}
```

Consulta el código en:

synchronization/interlocked

Ejemplo

- ¿Y la siguiente?

```
static long valor = 100000000;  
static void Main() {  
    const int numeroHilos = 10000;  
    int iteraciones = (int)valor / numeroHilos;  
    Thread[] hilos = new Thread[numeroHilos];  
    for (int i = 0; i < numeroHilos; i++)  
        hilos[i] = new Thread(() => {  
            for (int j = 0; j < iteraciones; j++)  
                Interlocked.Decrement(ref valor);  
        });  
    foreach (Thread hilo in hilos) hilo.Start();  
    foreach (Thread hilo in hilos) hilo.Join();  
    Console.WriteLine(valor);  
}
```

Consulta el código en:

synchronization/interlocked

Mutex y Semaphore

- Hasta ahora hemos visto sincronización entre hilos
- La sincronización entre procesos se puede conseguir con **mutex** y **semáforos**
- Un **Mutex** es un mecanismo de sincronización de **procesos** (también válido entre hilos)
 - Su funcionamiento es similar a **lock**
 - Su coste de rendimiento es de, aproximadamente, unas 50 veces mayor a **lock**
- Los **semáforos**, adicionalmente, permiten el acceso a n procesos (o hilos) concurrentes
 - Se suele utilizar para limitar la concurrencia (acotar un número máximo de procesos / hilos)

Mutex

Sólo puede existir uno con ese nombre en este ordenador

```
static void Main() {  
    using (var mutex = new Mutex(false, " MutexDemo")) {  
        // Espera cero milis y nos devuelve si el mutex está libre  
        // Sin parámetros, esperaría hasta que estuviese libre  
        if (!mutex.WaitOne(0)) {  
            Console.WriteLine("Otra instancia del programa  
                               está en ejecución.");  
            Console.WriteLine("El programa no se ejecutará.");  
            return; // El programa no se ejecuta  
        }  
        EjecutarPrograma();  
    } }  
  
static void EjecutarPrograma() {  
    Console.WriteLine("Esta sería la ejecución del programa.");  
    Console.WriteLine("Pulse Enter para salir.");  
    Console.ReadLine();  
}
```

Consulta el código en:

synchronization/mutex

Mutex

Threads de distintos
procesos

Hilo P3 Hilo P2 Hilo P1



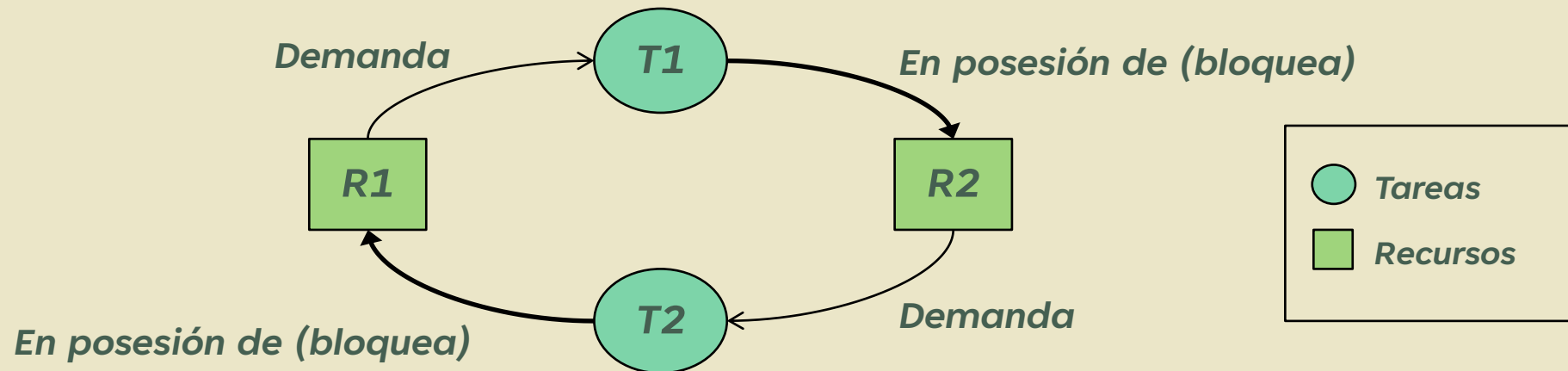
```
using(var mutex = new Mutex(false,  
    "Demo")) {  
    sección crítica  
}
```

 *Hilo P1*
Hilo P2

 *Hilo P1*

Interbloqueo

- Se produce un **interbloqueo** (*deadlock*) entre un conjunto de tareas (procesos o hilos) si todas y cada una de ellas están esperando por un evento que sólo otra puede causar
- Todas las tareas se bloquean de forma permanentemente
- El caso más común es el acceso a recursos compartidos
 - Todas y cada una de las tareas requieren algún recurso obtenido por otra tarea (bloqueando así su uso)



Ejemplo

```
public class Cuenta {  
    private decimal saldo;  
    public bool Retirar(decimal cantidad) {  
        if (this.saldo < cantidad) return false;  
        saldo -= cantidad;  
        return true;  
    }  
    public void Ingresar(decimal cantidad) {  
        saldo += cantidad;  
    }  
    public bool Transferir(Cuenta cuentaDestino, decimal cantidad) {  
        lock (this) {  
            lock (cuentaDestino) {  
                if (this.Retirar(cantidad)) {  
                    cuentaDestino.Ingresar(cantidad);  
                    return true;  
                }  
                else return false;  
            }  
        }  
    }  
}
```

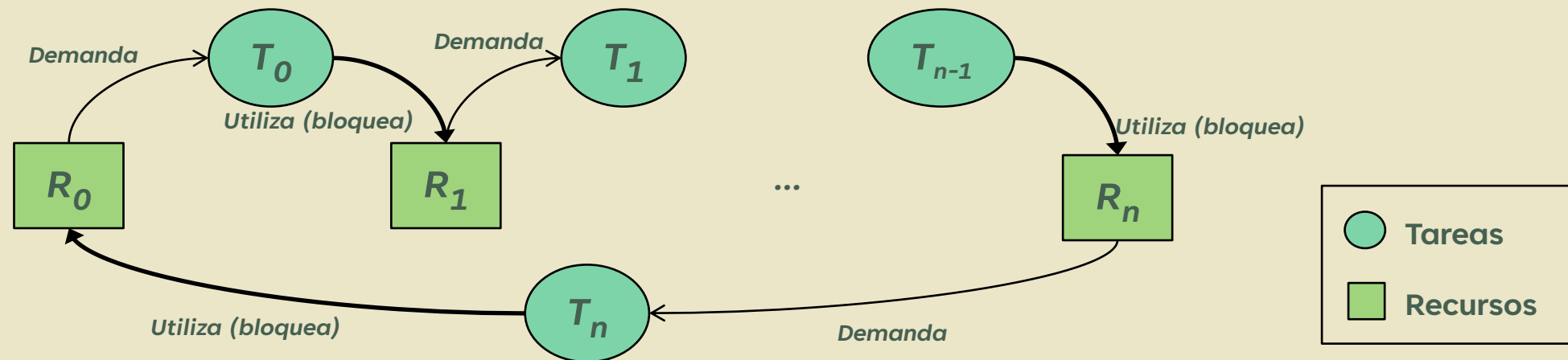
*¿Qué sucede cuando se ejecuta concurrentemente
cuentaA.transferir(cuentaB, imp); y
cuentaB.transferir(cuentaA, imp); ?*

Consulta el código en:

deadlock/

Condición de Espera Circular

- Esta condición se produce cuando dadas N tareas $T_0 \dots T_n$, T_0 posee el recurso R_1 pero espera por el R_0 , que debe ser liberado por la tarea $T_n \dots$ mientras T_n espera por el recurso R_n poseído por la tarea T_{n-1} mientras posee el recurso R_0
- Esta condición causa el interbloqueo de las tareas $T_0 \dots T_n$



Evitando el Interbloqueo

- El interbloqueo se evita **impidiendo que se dé la condición de espera circular**
 - Si ésta no se da, no existe interbloqueo
- Para ello, hay que saber a priori (estáticamente) los recursos que necesita cada proceso
 - La **obtención de recursos** debe ser llevada a cabo de forma que el sistema **no entre en un estado inseguro** (no puedan darse interbloqueos)
 - Algoritmo del banquero de Dijkstra
- No obstante, **no es siempre posible** conocer los recursos que un proceso requiere antes de su ejecución, porque depende de su ejecución (valores dinámicos)
 - No es posible crear un algoritmo general que decida si va a producirse un interbloqueo durante la ejecución de un programa > \Rightarrow es un problema **indecidable** (*undecidable*)

¿Podemos evitar la espera circular?

```
public class Cuenta {  
    private decimal saldo;  
    public bool Retirar(decimal cantidad) {  
        if (this.saldo < cantidad) return false;  
        saldo -= cantidad;  
        return true;  
    }  
    public void Ingresar(decimal cantidad) {  
        saldo += cantidad;  
    }  
    public bool Transferir(Cuenta cuentaDestino, decimal cantidad) {  
        lock (this) {  
            lock (cuentaDestino) {  
                if (this.Retirar(cantidad)) {  
                    cuentaDestino.Ingresar(cantidad);  
                    return true;  
                }  
                else return false;  
            }  
        }  
    }  
}
```

Consulta el código en:

deadlock/

De forma directa

```
public bool Retirar(decimal cantidad) {  
    lock(this) {  
        if (this.saldo < cantidad) }  
        return false;  
        saldo -= cantidad;  
        return true;  
    } }  
public void Ingresar(decimal cantidad) {  
    lock(this) {  
        saldo += cantidad;  
    } }  
public bool Transferir(Cuenta cuentaDestino, decimal cantidad) {  
    if (this.Retirar(cantidad)) {  
        cuentaDestino.Ingresar(cantidad);  
        return true;  
    }  
    else return false;  
}
```

¿Se podría sacar fuera del lock?

Consulta el código en:

deadlock/



Seminario 6

Interbloqueo

Thread Safety

- Un programa, método, función o clase se dice que es *thread-safe* si funciona correctamente (*safely*) cuando es **utilizada por varios hilos simultáneamente**
 - Este concepto también se aplica a estructuras de datos (y a clases, en orientación a objetos)
- El hecho de programar utilizando elementos *thread-safe*, no implica que el código escrito sea *thread-safe*
 - Por ejemplo, suponiendo que **Stack** es *thread-safe*

```
if (!stack.IsEmpty())
```

```
    stack.Pop().Mensaje();
```



*Puede haber sido
vaciada por otro hilo*

Estructuras de Datos *Thread-Safe*

- La mayor parte de tipos (clases) de propósito general ofrecidos por los lenguajes de programación **NO son *thread-safe*** por cuestiones de rendimiento
- Existen dos alternativas:
 1. Utilizar estructuras de datos especiales (`System.Collections.Concurrent`)
 2. O realizar la sincronización con bloqueos de objetos *thread-unsafe*

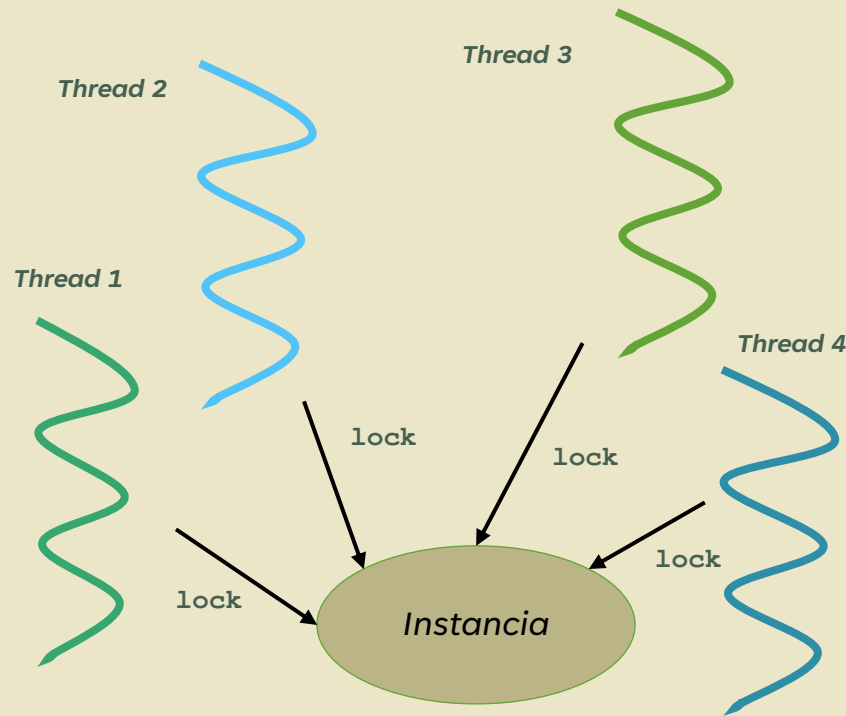
```
static void AñadirYMostrarThreadSafe(List lista) {  
    lock (lista) lista.Add("Item " + lista.Count);  
    string[] items;  
    lock (lista) items = lista.ToArray();  
    lock (Console.Out)  
        foreach (string s in items) Console.WriteLine(s);  
}
```

Implementación EEDD *Thread-Safe*

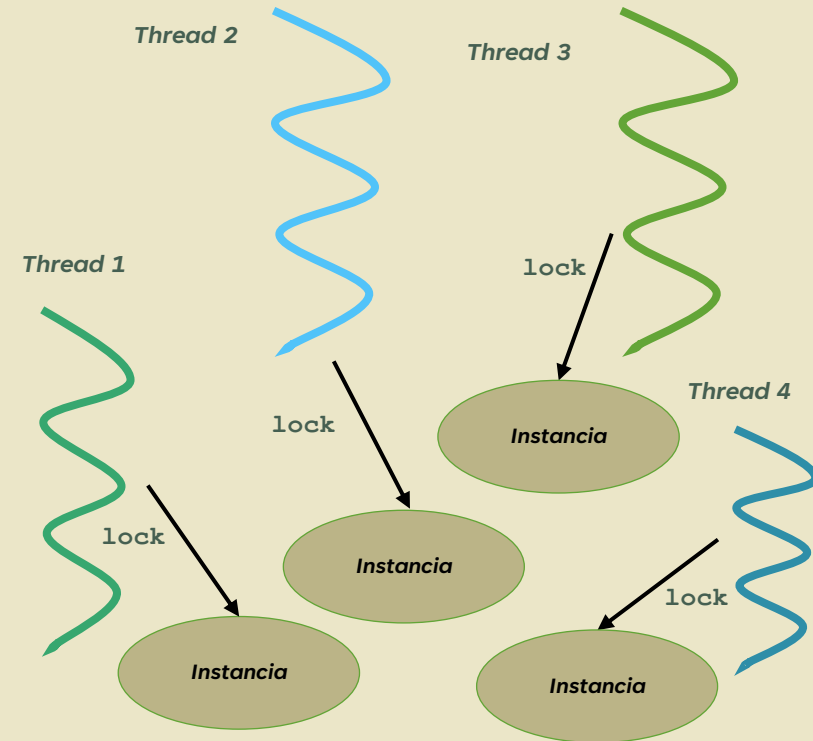
- Una implementación trivial de estructuras de datos *thread-safe* es (patrón de diseño **Decorator**)
 1. Partir de una estructura *thread-unsafe*
 2. Crear una nueva clase con la misma interfaz
 3. Realizar una composición a una instancia de la clase *thread-unsafe*
 4. Realizar un bloqueo (**lock**) en el cuerpo de todos los métodos, utilizando un objeto privado como monitor (la propia colección *thread-unsafe*)



Implementación EEDD *Thread-Safe*



```
class ListaThreadSafe {  
    public object bloqueador = new ...  
    ...  
    public void Add(object obj) {  
        lock(bloqueador) {  
            ... //sección crítica  
        }  
    }  
}
```



```
class ListaThreadSafe {  
    ...  
    public void Add(object obj) {  
        lock(new Object()) {  
            ... //sección crítica  
        }  
    }  
}
```



Preguntas

- En la anterior transparencia comentamos, para hacer una ED *thread safe*:
 - Realizar un bloqueo (**lock**) en el cuerpo de todos los métodos, **utilizando un objeto privado** como monitor (la propia colección *thread-unsafe*)
- ¿Cuál utilizaremos en nuestro ejemplo?
- ¿Por qué es mejor que **this**?



lock (this)

- Se puede usar `lock (this)`, pero puede originar que un hilo espere a que otro finalice sin ninguna razón aparente
- Esto puede afectar el rendimiento del programa o incluso originar interbloqueos (deadlocks)

Mismo objeto

```
class ThreadSafeList {
    public void Add(object [] objs) {
        lock (this) {
            Console.WriteLine("Begin adding objects");
            foreach (object o in objs) {
                Thread.Sleep(1000); //Simulates processing
                Console.WriteLine("Object added");
            }
        }
    }
}
```

```
ThreadSafeList p = new ThreadSafeList();
object[] objs = ...
Thread populate = new Thread(() => p.Add(objs));

Console.WriteLine("Main thread starts");
populate.Start();

Thread.Sleep(1000); //Simulates processing
Console.WriteLine("Begin the main thread execution");
//p is already blocked by the thread populate!
lock (p) {
    // This will not start unless the populate thread finishes
    // This thread is blocked by the populate thread with no
    // apparent reason
    Thread.Sleep(1000); //Simulates processing
    Console.WriteLine("Main thread ends");
}

Console.WriteLine("*** PROGRAM ENDS ***");
```


Implementación EEDD *Thread-Safe*

- Esta es una implementación **sencilla** pero muy **ineficiente**
- En general, las operaciones concurrentes de lectura no requieren **exclusión mutua** de otras operaciones de lectura
 - El bloqueo entre ellas conllevaría una pérdida de rendimiento
- Las **escrituras**, no obstante, **sí deben bloquear** a otras lecturas y escrituras
- En C#, los mecanismos de sincronización que hemos visto (`lock`) no permiten obtener esta funcionalidad
 - No puede distinguir la intención del bloque de código que contiene
- Por ello se ha añadido la clase **ReaderWriterLockSlim** al *.Net Framework 4*

ReaderWriterLockSlim

- `ReaderWriterLockSlim` permite proteger un recurso compartido por varios hilos de ejecución de forma **más eficiente** que `lock`
- Permite a los hilos bloquear el acceso a un recurso de tres modos:
 - **Modo de lectura:** puede haber cualquier número de hilos de ejecución con un bloqueo en este modo sobre una misma instancia de `ReaderWriterLockSlim` sin ser bloqueados
 - **Modo de escritura:** si un hilo de ejecución ha entrado en un bloqueo en este modo sobre una instancia de `ReaderWriterLockSlim`, ningún otro puede entrar en el bloqueo sobre ella **en cualquiera de los tres modos**
 - Tiene por tanto **acceso exclusivo** al recurso
 - Todos los demás hilos **esperarán a que se libere**
 - **Modo de lectura actualizable: bloqueo de lectura especial** que puede actualizarse a modo de escritura sin tener que abandonar el acceso de lectura al recurso
 - Solo puede haber un hilo de ejecución con un bloqueo en este modo sobre una misma instancia de `ReaderWriterLockSlim` en cualquier momento
 - El hilo que ha bloqueado en este modo **puede acceder** a la sección crítica al mismo tiempo que otros hilos que han bloqueado a la instancia en modo lectura

ReaderWriterLockSlim

- Por defecto, el código protegido por `ReaderWriterLockSlim` **no puede tener** llamadas recursivas
 - Se pasa el parámetro `LockRecursionPolicy.NoRecursion` por defecto al crear instancias
- Esta política **disminuye la posibilidad de interbloqueos y mejora el rendimiento**
- Se pueden soportar llamadas recursivas en caso necesario pasando `LockRecursionPolicy.SupportsRecursion` en el constructor
- Independientemente de la posibilidad de hacer llamadas recursivas o no, se cumplen las normas de bloqueos y modos vistas antes
- El siguiente ejemplo muestra como trabajar con una instancia de `ReaderWriterLockSlim` llamada `CacheLock`
- **Más información:** <https://learn.microsoft.com/es-es/dotnet/api/system.threading.readerwriterlockslim?view=net-5.0>

ReaderWriterLockSlim

```

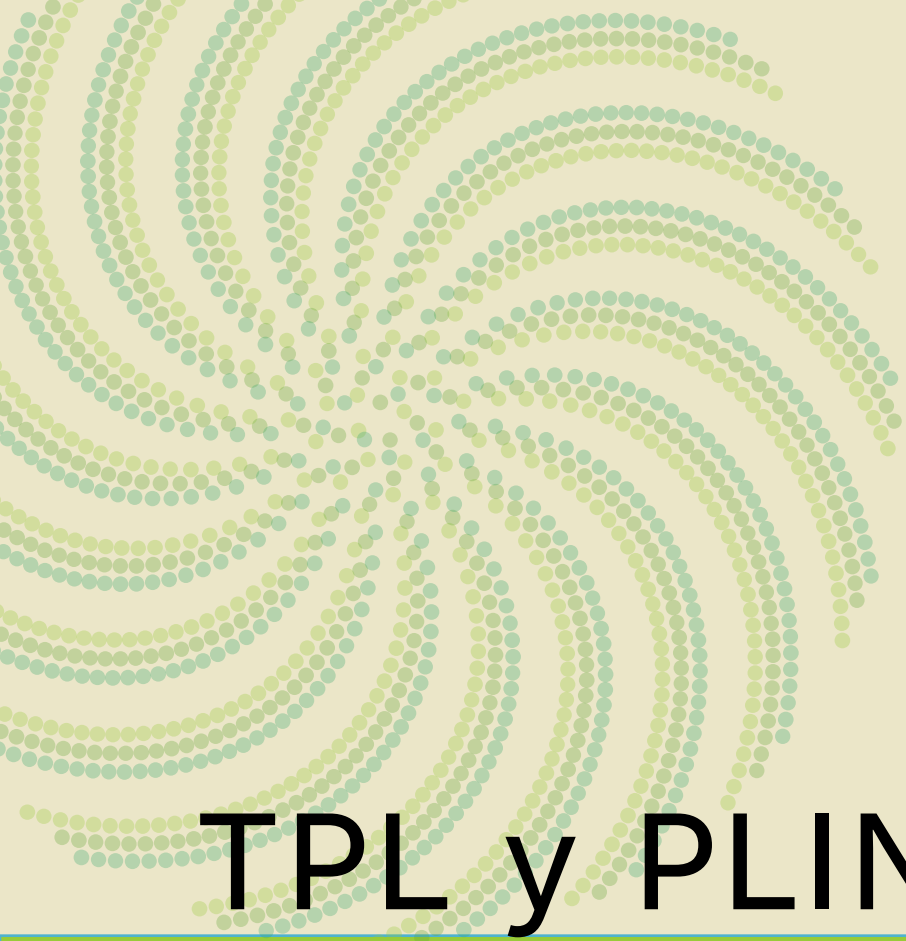
/*Leemos el valor y solo lo añadimos o actualizamos si no existe o tiene un valor distinto: usamos un
bloqueo de lectura actualizable a uno de escritura solo cuando sea necesario*/
CacheLock.EnterUpgradeableReadLock();
try {
    string result = null;
    if (InnerCache.TryGetValue(key, out result)) {
        //El valor existe, por lo que no hay nada más que hacer
        if (result == value) return;
        else {
            //Actualizamos entrando en un bloqueo de escritura
            CacheLock.EnterWriteLock();
            try { InnerCache[key] = value; }
            finally {
                //Siempre libera el lock, evitando interbloqueos
                CacheLock.ExitWriteLock();
            }
        }
    }
}
finally {
    //Siempre libera el lock, evitando interbloqueos
    // si otro hilo intenta crear uno de este tipo
    CacheLock.ExitUpgradeableReadLock();
}

```

```

public string Translate(string key)
{
    /*Varios threads pueden leer.
    Bloquea si otro thread tiene un
    bloqueo de escritura sobre
    CacheLock*/
    CacheLock.EnterReadLock();
    try
    {
        return InnerCache[key];
    }
    finally
    {
        CacheLock.ExitReadLock();
    }
}

```



TPL y PLINQ

Simplificando la creación de código concurrente

Task Parallel Library

- Para simplificar la creación de tareas y crearlas de forma transparente, se creó la **Task Parallel Library** (TPL)
 - Parte del .NET Framework 4.0
 - API en el *namespace* `System.Threading.Tasks`
- Ofrece las siguientes ventajas
 - **Simplifica** la paralelización de aplicaciones
 - **Escala dinámicamente** el número de hilos creados en función del **número de CPUs o cores**
 - Escala y gestiona dinámicamente el número de hilos creados **en función del Thread Pool**
 - Ofrece servicios para la **paralelización** mediante la **división de datos procesados** (*data parallelism*)
 - Ofrece servicios para la **paralelización** mediante **tareas independientes** (*task parallelism*)
- **TPL** ofrece un modelo mucho más **declarativo** de implementar aplicaciones paralelas
- **Más información:** <https://learn.microsoft.com/es-es/dotnet/standard/parallel-programming/task-parallel-library-tpl>

Data Parallelism con TPL

- Para obtener la paralelización mediante división de datos, los dos métodos más utilizados son **ForEach** y **For** de la clase `System.Threading.Tasks.Parallel`
 - Ambos **reciben** la tarea a ejecutar como un delegado (**Action**)
 - **ForEach** crea potencialmente un hilo por cada elemento de un **IEnumerable**
 - **For** crea potencialmente un hilo a partir de un **índice de comienzo y final**, no incluyendo el final
 - Añade una **sincronización** para que en la siguiente instrucción todos los **hilos** hayan **finalizado**

Parallel.ForEach

```
string[] ficheros=Directory.GetFiles(@"..\..\..\pics", "*.jpg");
string nuevoDirectorio = @"..\..\..\pics\rotadas";
Directory.CreateDirectory(nuevoDirectorio);

Parallel.ForEach(ficheros, fichero => {
    string nombreFichero = Path.GetFileName(fichero);
    using (Bitmap bitmap = new Bitmap(fichero)) {
        Console.WriteLine("Procesando el fichero {0} con el hilo {1}.",
            nombreFichero, Thread.CurrentThread.ManagedThreadId);
        bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
        bitmap.Save(Path.Combine(nuevoDirectorio, nombreFichero));
    }
});
// TPL espera a la finalización de todas las tareas
// (sincronización)
```

Este código puede ejecutarse concurrentemente, por lo que debe usarse **lock** en caso necesario! (acceso a estructuras de datos no thread-safe, por ejemplo)

Consulta el código en:

[tpl/data.parallelism](https://github.com/dotnet/tpl/tree/master/data.parallelism)

Comparación con Impl. Secuencial


- La utilización de TPL en un i7-1280P (hasta 4,80GHz), 32GB y Windows 11 22H2 en comparación con una implementación secuencial ha sido

	Secuencial	TPL
Hilos Creados	1	12*
Tiempo (ms)	822	315
Beneficio		160,95%

- ¡Aumentando el número de ficheros, TPL no aumenta el número de hilos creados!
 - Estima dinámicamente que el rendimiento no se verá incrementado
- Los cambios en el código fuente son mínimos
 - La espera a la finalización de hilos es automática

Data Parallelism con TPL y variables locales de partición

- Cada partición (hilo) tiene su propia variable **subtotal**
- Minimiza las “colisiones” de los mecanismos de sincronización

```
short[] vector = CreateRandomVector(100000, -100, 100);  
long result = 0;  
Parallel.ForEach(vector,  
    () => 0, // Method to initialize the local variable  
    (v, loopState, subtotal) => subtotal += v * v,   
    // Method to be executed when each partition has completed.  
    // finalResult is the final value of subtotal for a particular partition.  
    finalResult => Interlocked.Add(ref result, finalResult));  
Console.WriteLine("The result obtained is: {0:N2}.", Math.Sqrt(result));
```

Este código se ejecuta en paralelo por cada partición, y por lo tanto NO necesita **lock**

Task Parallelism with TPL

- Para obtener la paralelización mediante **división de tareas independientes**, TPL ofrece el método **Invoke** de la clase **Parallel**
 - Recibe una **lista variable de** delegados de tipo **Action**
 - Crea potencialmente un hilo **por cada Action** pasado como parámetro
 - Añade una **sincronización** para que en la siguiente instrucción todos los **hilos** hayan **finalizado**

Parallel.Invoke

```
String texto = LeerFicheroTexto(@"..\..\..\clarin.txt");
string[] palabras = PartirEnPalabras(texto);
Parallel.Invoke(
    () => signosDePuntuación = SignosPuntuación(texto),
    () => palabrasMasLargas = PalabrasMasLargas(palabras),
    () => palabrasMasCortas = PalabrasMasCortas(palabras),
    () => palabrasConMasApariciones = PalabrasConMasApariciones(
        palabras, out numeroMayorApariciones),
    () => palabrasConMenosApariciones = PalabrasConMenosApariciones(
        palabras, out numeroMenorApariciones)
);
```

Este código puede ejecutarse concurrentemente, por lo que debe usarse **lock** en caso necesario! (acceso a estructuras de datos no thread-safe, por ejemplo)

Consulta el código en:

[tpl/task.parallelism](https://github.com/dotnet/corefx/blob/master/src/System.Threading.Tasks/Parallel.Invoke.cs)

Comparación con Impl. Secuencial

- La utilización de TPL en un i7-1280P (hasta 4,80GHz), 32GB y Windows 11 22H2 en comparación con una implementación secuencial ha sido

	Secuencial	TPL
Hilos Creados	1	5
Tiempo (ms)	393	296
Beneficio		32,77%

- Aumentando el tamaño del fichero aumenta el beneficio
 - Los datos anteriores son para un fichero de 2,33MB
- En este caso, el beneficio obtenido es pequeño (33%)
- ¡Es importante analizar si la paralelización representa un beneficio o no!
 - La paralelización no siempre conlleva un beneficio de rendimiento**
- Para saber más de como usar TPL se recomienda consultar: <https://learn.microsoft.com/es-es/dotnet/standard/parallel-programming/task-parallel-library-tpl>



Seminario 7

Paralelización con TPL

Paradigma Funcional

- Hemos visto cómo las **aplicaciones con varias tareas** requieren que las **abstracciones** utilizadas sean *thread-safe*
- Esta **condición se rompe** cuando se accede a recursos o estructuras de datos con **estado mutable** (ejemplo, el operador de asignación)
 - La computación depende de un estado que cambia a lo largo del tiempo (recursos, entrada y salida, variables globales, objetos mutables...)
 - Se necesita por tanto sincronización
- En el **paradigma funcional puro**, no hay modificaciones de estado gracias a la transparencia referencial
 - Por tanto, este paradigma facilita la paralelización de algoritmos

Paradigma Funcional en .Net

- Debido a
 - Las ventajas del **paradigma funcional** en la creación de **programas concurrentes**
 - **Alta demanda de aplicaciones concurrentes**
- La plataforma .NET hace uso extensivo de este paradigma para el desarrollo de programas concurrentes (delegados y funciones lambda)
 - Paso asíncrono de mensajes
 - Utilización de *callbacks* para finalización de hilos
 - Creación explícita de hilos
 - *Task Parallel Library* (For, Foreach, Invoke...)
 - Y, especialmente, **Parallel LINQ** (PLINQ)

Parallel LINQ

- *Parallel LINQ* (PLINQ) es una implementación paralela de *LINQ to Objects*
 - *LINQ to Objects* permite hacer consultas contra colecciones en memoria como `IEnumerable` o arrays (no se soportan BBDD o APIs XML)
- Gracias al modelo basado en el paradigma funcional, ofrece
 - **Paralelismo**
 - De un modo declarativo
 - **Transparente** al número de **núcleos**, eligiendo dinámicamente el número de hilos concurrente
 - Una aproximación **conservadora**, analizando la estructura de la consulta en tiempo de ejecución
 - Si la consulta puede obtener mejoras de rendimiento mediante paralelización, PLINQ particiona los datos y ejecuta las tareas de forma concurrente
 - Ejecutándola de modo secuencial si no fuese así

Parallel LINQ

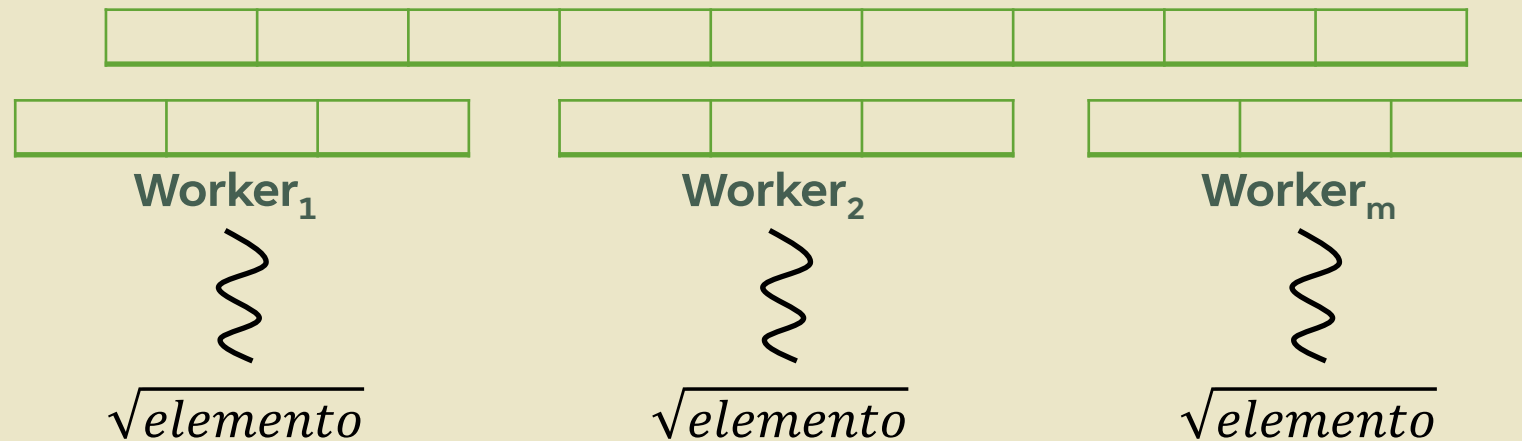
- LINQ trabaja sobre datos **secuencialmente**

```
vector.Select(elemento => Math.Sqrt(elemento))
```

- PLINQ

1. Parte esos datos en segmentos
2. Ejecuta la consulta LINQ en paralelo con un número (dinámico) de hilos *worker*
3. Cada hilo *worker* procesa un segmento distinto

```
vector.AsParallel().Select(elemento => Math.Sqrt(elemento));
```



Este código puede ejecutarse concurrentemente, por lo que ¡debe usarse **lock** en caso necesario! (acceso a estructuras de datos no thread-safe, por ejemplo)

Pregunta

- Volvemos al problema que resuelve es el cálculo del **módulo de un vector** algebraico de n dimensiones

$$|\vec{v}| = \sqrt{v_1^2 + v_2^2 + \dots + v_n^2}$$

- ¿Cuál sería la consulta LINQ que calcula el módulo del vector?
- ¿Cuál sería la consulta PLINQ?

¿Alternativas?

- LINQ (secuencial)

```
Math.Sqrt(  
    vector.Aggregate<short,long>(0, (acc, item) => acc + item * item)  
);
```

- PLINQ (un solo Aggregate)

```
Math.Sqrt(  
    vector.AsParallel().Aggregate<short,long>(0, (acc, item) => acc + item * item)  
);
```

- PLINQ (Select + Aggregate)

```
Math.Sqrt(  
    vector.AsParallel().Select(item => item * item)  
        .Aggregate<short,long>(0, (acc, item) => acc + item * item));
```

Mejora del Rendimiento

- Los siguientes son datos de ejecución del cálculo del módulo del vector en un ordenador con **veinte núcleos** y 8GBytes de RAM
- Se utilizó un vector de 100.000.000 elementos aleatorios comprendidos en [-10,10]
 - Secuencial (LINQ): 605 milisegundos
 - Paralelo v1 (PLINQ): 432 milisegundos
 - Beneficio: 37,70%
 - Paralelo v2 (PLINQ): 137 milisegundos
 - Beneficio: 329,20%
- ¿Por qué crees que el beneficio de rendimiento es de solo un 37,70% en la primera versión? ¿Porqué se incrementa en la segunda versión?
- Para saber más de como usar PLINQ se recomienda consultar:
<https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq>

Consulta el código en:

Plinq/

PLINQ con variables locales de partición

- De manera similar a la sobrecarga vista para **ForEach...**
- En este caso la mejora llega al 833.33% (respecto a secuencial)

```
Math.Sqrt(vector.AsParallel().Aggregate<short, long, long>(
    () => 0, // local init (seed Factory)
    (acc, item) => acc + item * item,
    (acc1, acc2) => acc1 + acc2, // combine local accumulators
    finalResult => finalResult)); // final state
```

Este código se ejecuta en paralelo por cada partición, y por lo tanto tiene una copia local del acumulador

Ley de Amdahl

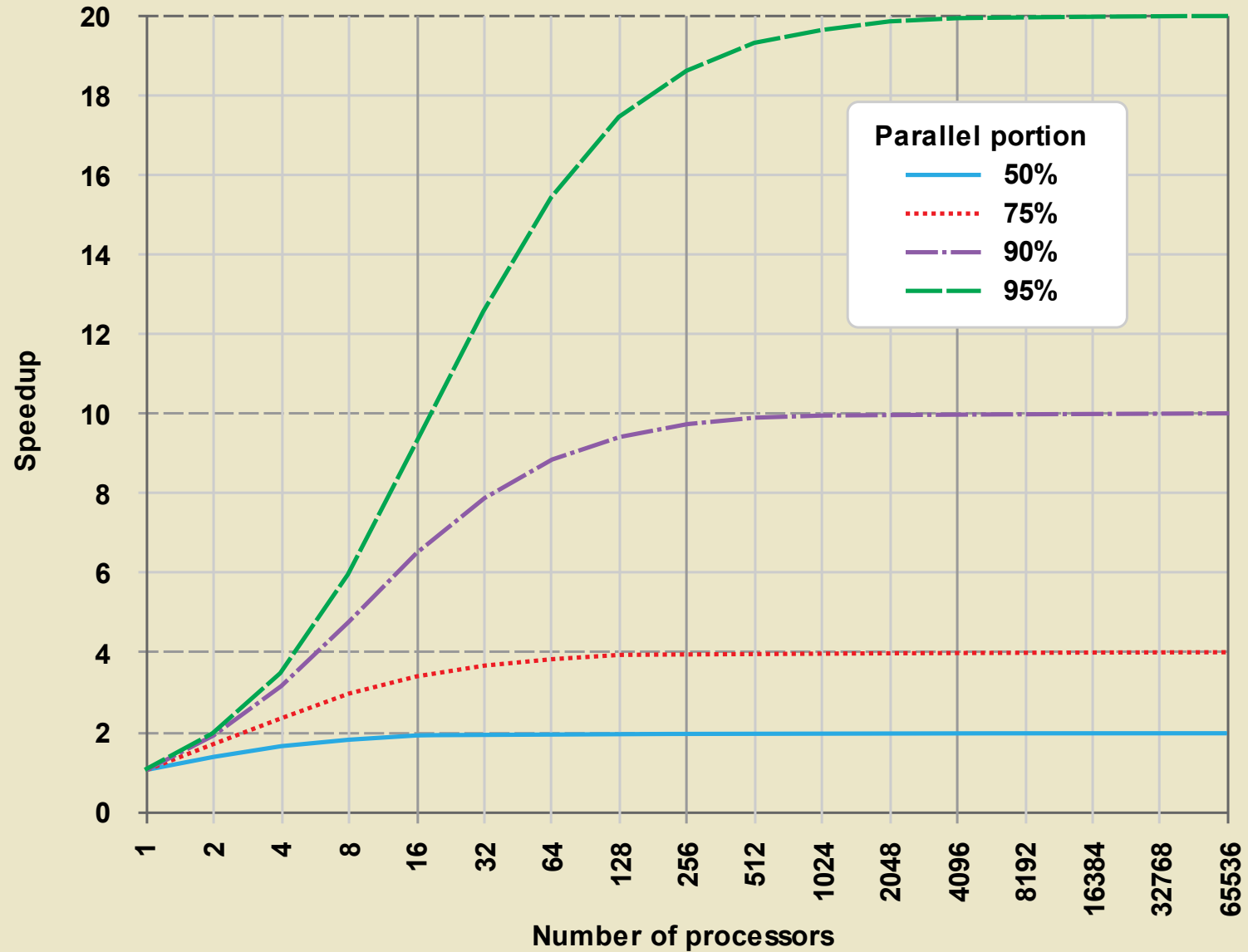
- **La ley de Amdahl** indica la mejora esperada máxima en un sistema cuando solo una parte del mismo es mejorada
- En computación paralela, predice la mejora máxima **teórica** cuando se usan múltiples procesadores
- La mejora de rendimiento de un programa completo está limitada por el tiempo de ejecución de la parte secuencial de dicho programa
 - **50% secuencial, 50% paralelo** \Rightarrow
La mejora máxima teórica de rendimiento es **2** (100%)
 - **25% secuencial, 75% paralelo** \Rightarrow
La mejora máxima teórica de rendimiento es **4** (300%)
 - **10% secuencial, 90% paralelo** \Rightarrow
La mejora máxima teórica de rendimiento es **10** (900%)
- En general, cuanto más tiempo de ejecución tenga la parte secuencial menor mejora de rendimiento tendrá la aplicación completa (y viceversa)

Mejora máxima Teórica = $1/(1-p)$

p = Tanto por 1 del tiempo de ejecución de la parte paralela

Ej.: 75% paralelo $\rightarrow 1/(1-0,75) = 1/0,25 \rightarrow 4$

Amdahl's Law



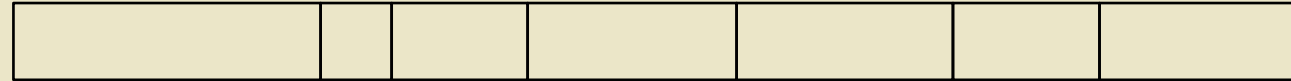
MapReduce

- MapReduce es un modelo de programación de aplicaciones paralelas (**distribuidas**) mediante división de datos (*data paralellism*)
- Hace uso de **dos funciones de orden superior**
 - **Map**: Procesa todos los elementos de una lista y genera otra lista
 - **Reduce (Fold)**: Procesa los elementos de una lista generando un valor
- MapReduce trabaja con diccionarios (clave,valor) en lugar de con listas (interpretación clásica)
- Fue presentado (y patentado) por Google en 2004
 - Lo utiliza para múltiples escenarios: regenerar el índice de páginas Web, contar apariciones de palabras...
- Hay implementaciones para varios lenguajes
 - Un buen ejemplo es Hadoop (Java)

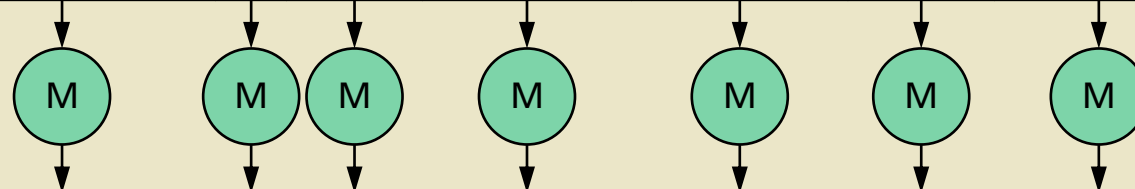
MapReduce, Concurrencia

Ejemplo clásico: Contar las palabras de un documento

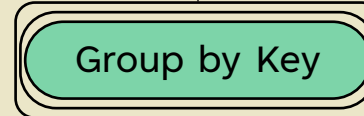
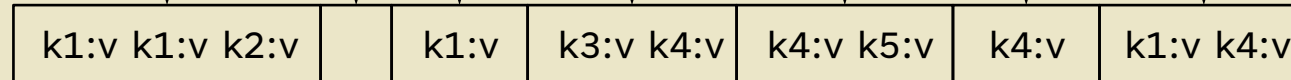
Input



Map que calcula los valores (v) de cada clave (k)

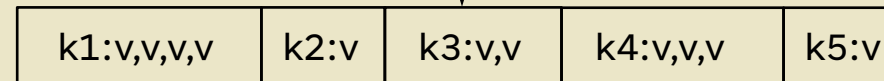


Intermediate

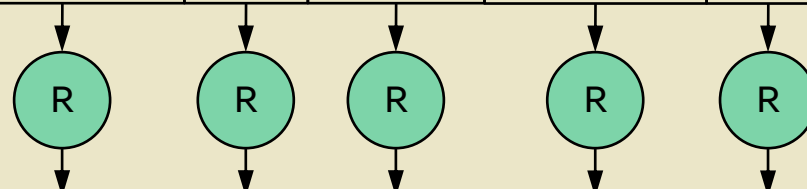


Agrupar resultados por clave

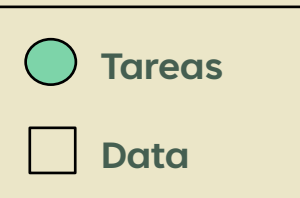
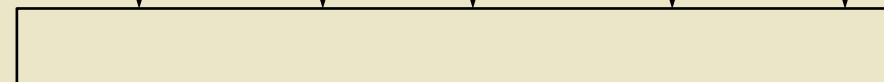
Grouped



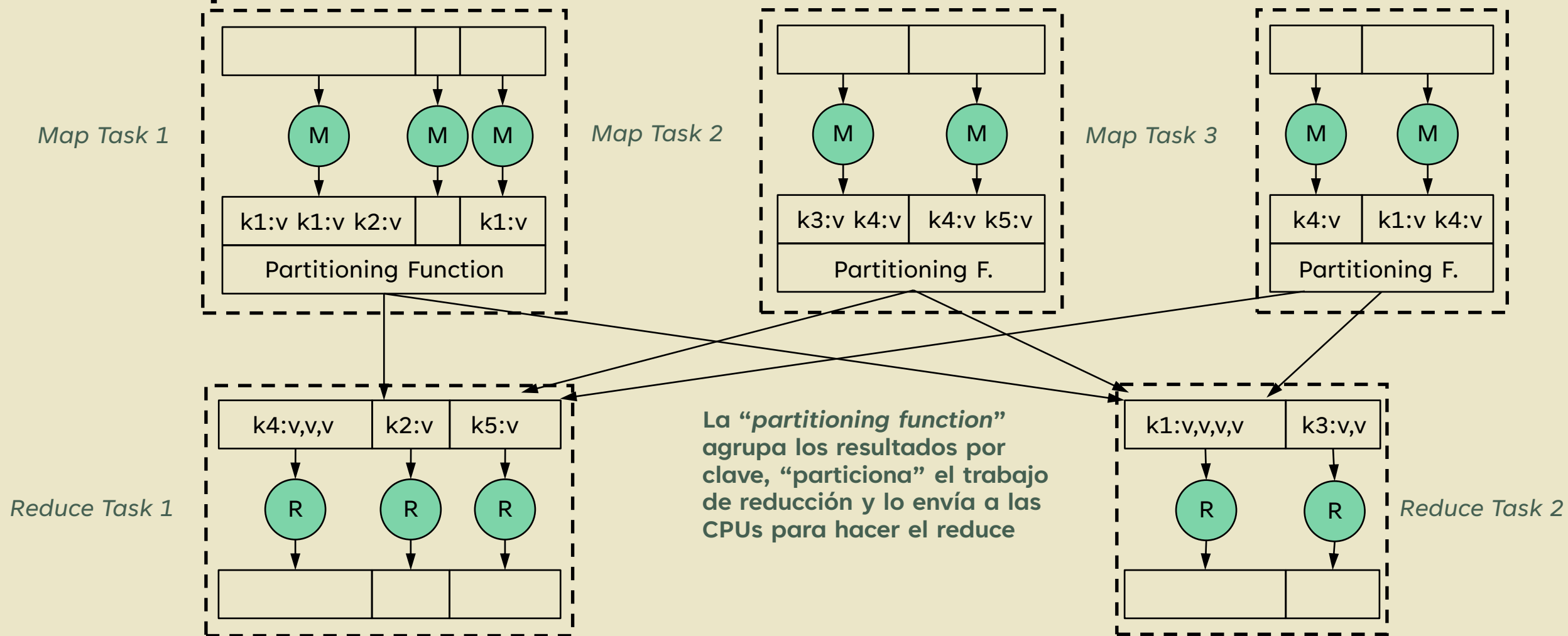
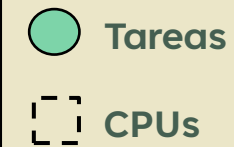
Cada reduce calcula los valores para una clave dada



Output

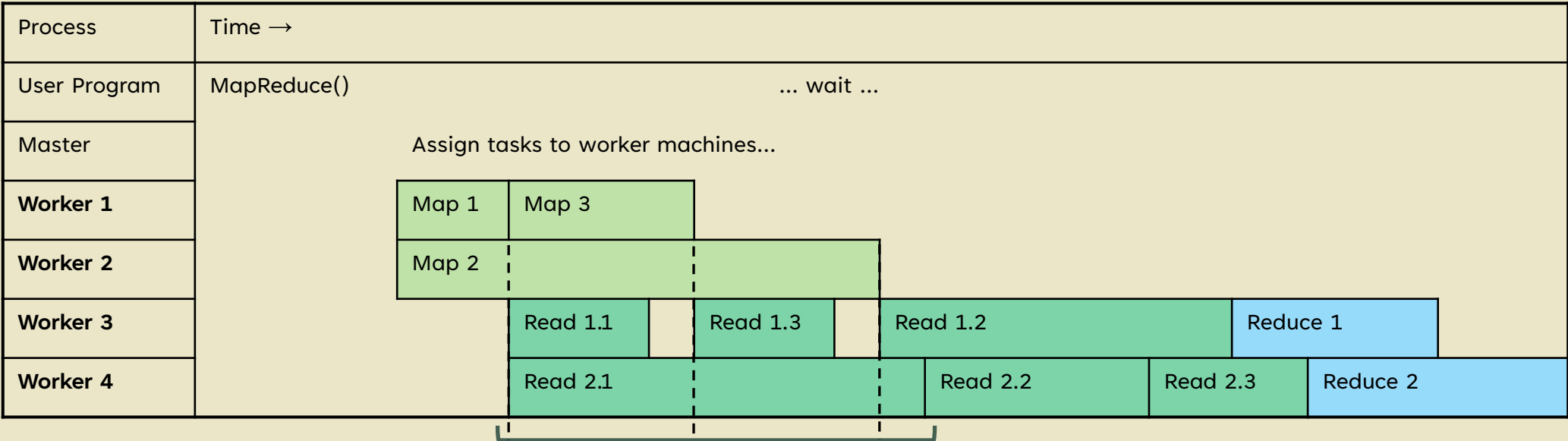


MapReduce, Paralelismo



MapReduce, Sincronización

- Sigue un modelo de paralelismo **pipeline**: Los datos son "particionados" por el "master" y cada "worker" procesa una tarea con los datos que se le envíen
 - Los **reads** implementan la *partitioning function*
 - Los **reduces** ejecutan la reducción (*aggregate*) de las claves que le hayan sido asignadas



`Read worker#-2.map#`

Los “reads” pueden ejecutarse cuando acaben los “maps” asociados