

# Fundamentos de Programación Funcional y Concurrente

Principios de concurrencia y paralelismo

Juan Francisco Díaz Frias

Profesor Titular (1993-hoy)  
juanfco.diaz@correounivalle.edu.co  
Edif. B13 - 4009



**Universidad del Valle**

Octubre 2023

# Plan

1 Computación paralela y concurrente

2 Concurrencia en la *JVM*

# Plan

- 1 Computación paralela y concurrente
- 2 Concurrencia en la *JVM*

# Contents

1 Computación paralela y concurrente

2 Concurrencia en la *JVM*

# Computación paralela

- La **computación paralela** es un tipo de computación en la cual muchos cálculos se realizan al mismo tiempo.
- **Principio básico**: la computación se puede dividir en computaciones más pequeñas, cada una de las cuales se puede desarrollar simultáneamente.
- **Suposición**: tenemos a nuestra disposición hardware paralelo, que es capaz de ejecutar estas computaciones en paralelo.

# Algo de historia

- El **hardware paralelo** ha estado presente desde los principios de la computación (Ver [Fotos evolución computadores](#) ).
- Sin embargo, la computación paralela, entendida como al menos dos procesadores independientes, utilizados para realizar un cálculo, sólo aparece unos años después de la aparición de los primeros computadores comerciales.
- IBM fue pionero en el desarrollo de computadores paralelos comerciales. Al principio, la computación en paralelo, estuvo confinada a comunidades muy específicas que trabajaban en computación de alto desempeño.
- Recientemente, inicios del siglo XXI:
  - La velocidad de los procesadores escaló a niveles difíciles de superar
  - Los vendedores de procesadores decidieron proveer múltiples núcleos de CPU en el mismo chip, cada uno pudiendo ejecutar separadamente flujos de instrucciones.
  - **Tema común:** la computación en paralelo puede proveer potencia de cálculo donde la computación secuencial no puede hacerlo.

# ¿Por qué computación paralela?

Handwritten diagram illustrating a race condition. It shows two parallel paths: one adding 3 to 1, and another multiplying 2 by 2. Both paths converge at a point where the value of  $x$  is 3. From there, two different calculations are shown:  $(3+3)*2 = 12$  and  $(3*2)+3 = 9$ , demonstrating how the final result depends on the execution order.

- La programación paralela es mucho más difícil que la programación secuencial:
  - Separar computaciones secuenciales en computaciones en paralelo es retador, y algunas veces, imposible.
  - Asegurar que los programas paralelos son correctos, es mucho más difícil, debido a nuevos tipos de errores
- La **aceleración** (*speedup*) de los cálculos es la única razón por la que aceptamos que programar sea más complejo.

# Programación concurrente vs Programación paralela

- **Paralelismo** y **concurrencia** son conceptos estrechamente relacionados.
- La **programación en paralelo** usa hardware en paralelo para ejecutar computaciones más rápidamente. La eficiencia es su principal preocupación.
- La **programación concurrente** puede o no realizar múltiples ejecuciones al mismo tiempo. Su principal preocupación es la modularidad, la capacidad de respuesta y la mantenibilidad.



# La granularidad del paralelismo

El paralelismo se manifiesta en diferentes niveles de granularidad:

- Paralelismo a nivel de bits: procesamiento de múltiples datos en paralelo
- Paralelismo a nivel de instrucciones: ejecución de diferentes instrucciones del mismo flujo de instrucciones, en paralelo
- **Paralelismo a nivel de tareas**: ejecución de flujos de instrucciones separadas en paralelo

En este curso nos enfocaremos en paralelismo a nivel de tareas

# Tipos de computadores paralelos

El hardware para computación en paralelo viene en diferentes formas:

- Procesadores multi núcleos
- Multiprocesadores simétricos
- Unidades de procesamiento gráfico de propósito general
- Arreglos de compuertas programables
- Clusters de computadores

En este curso nos enfocaremos en programación para procesadores multinúcleos y multiprocesadores simétricos

# Foco del curso

- Fundamentos de programación paralela/concurrente y análisis de programas paralelos/concurrentes
- Paralelismo de tareas: algoritmos paralelos básicos
- Paralelismo de datos: colecciones paralelas en Scala
- Estructuras de datos para la computación paralela

# Contents

1 Computación paralela y concurrente

2 Concurrencia en la *JVM*

# *JVM* y paralelismo

- Existen muchas formas de paralelismo
- Supondremos un modelo de programación paralela para sistemas multinúcleo o multiprocesadores con memoria compartida.
- Y el sistema operativo y la *JVM* como los entornos en tiempo de ejecución subyacentes

# Procesos

- Sistema operativo: es el software encargado de administrar los recursos de hardware y software, y planificar la ejecución de los programas.
- Un **proceso** es una instancia de un programa que se está ejecutando en el sistema operativo.
- El mismo programa puede ser iniciado como un proceso más de una vez, o aún simultáneamente en el mismo sistema operativo.
- El sistema operativo multiplexa muchos procesos diferentes y un número limitado de CPUs, consiguiendo fragmentos de tiempos de ejecución para cada proceso. Este mecanismo se denomina **multitareas** (*multitasking*).
- Dos procesos diferentes no pueden acceder directamente a la memoria del otro. Ellos están aislados.

# Hilos

- Cada proceso puede contener múltiples unidades concurrentes independientes llamadas **hilos**
- Los hilos se pueden iniciar desde el mismo programa y comparten el mismo espacio de memoria.
- Cada hilo tiene su propia pila de ejecución
- Los hilos de *JVM* no pueden modificar la pila de memoria de otros hilos. Ellos sólo pueden modificar la memoria *heap*.

# Creando y haciendo seguimiento a los hilos

- Mientras no se creen más hilos, todo corre en un hilo:

```
0 scala> def log(msg: String): Unit = println(s"Thread.currentThread.getName :msg")
1 def log(msg: String): Unit
2 scala> log(": este es mi nombre")
3 main: : este es mi nombre
```

- La clase *Thread* permite crear hilos:

```
0 def thread(cuerpo: =>Unit): Thread = {
1   val t = new Thread {
2     override def run() = cuerpo
3   }
4   t.start()
5   t
6 }
```

El método *thread*, recibe un bloque de código *cuerpo*, y crea un hilo nuevo que ejecutará ese bloque cuando el hilo se inicie. Luego inicia el hilo (*t.start()*), y devuelve una referencia *t* al hilo, para que otros procesos puedan invocar métodos sobre ese hilo.

- Cuando el hilo actual requiere detenerse hasta tanto el hilo *t* no termine, el hilo actual invoca *t.join*.



# Semántica de intercalación (1)

- El cuerpo de cada hilo se ejecuta secuencialmente
- Cuando hay dos o más hilos en ejecución, es el sistema operativo quien decide a **cuál hilo** darle la mano primero y por **cuánto tiempo**.
- Hay diferentes estrategias de **planificación de la ejecución de los hilos**. Lo importante es que sea **justa**, lo cual significa que a todo hilo ejecutable se le da la mano en algún momento.
- Se cual sea la estrategia utilizada, la semántica de la ejecución de los hilos es una **semántica de intercalación**:

```
0  def log(msg: String): Unit =  
1      println(s" Thread.currentThread.getName :msg" )  
2  
3  def thread(cuerpo: =>Unit): Thread = {  
4      val t = new Thread {  
5          override def run() = cuerpo  
6      }  
7      t.start()  
8      t  
9  }  
10 val t = thread { log(" Hilo-nuevo-corriendo" ) }  
11 log(" ... " )  
12 log(" ... " )  
13 t.join()  
14 log(" Hilo-nuevo-terminado" )
```

# Semántica de intercalación (2)

- Si ejecutamos el código anterior varias veces, la semántica de intercalación puede dar lugar a diferentes ejecuciones
- La primera vez:

```
0 scala> main: ...  
1 Thread-0: Hilo nuevo corriendo.  
2 main: ...  
3 main: Hilo nuevo terminado.  
4 val t: Thread = Thread[Thread-0,5,]
```

- La segunda vez:

```
0 scala> main: ...  
1 main: ...  
2 Thread-1: Hilo nuevo corriendo.  
3 main: Hilo nuevo terminado.  
4 val t: Thread = Thread[Thread-1,5,]
```

- Este comportamiento **no determinístico** es el que hace difícil la programación concurrente o paralela.

# Atomicidad

- En ocasiones, se desea asegurar que una secuencia de instrucciones dentro de un hilo se ejecutan todas o no se ejecuta ninguna, pero no puede haber lugar a intercalación con otros hilos: **atomicidad**.
- Considere el siguiente programa para generar identificadores únicos frescos para una aplicación:

```
0  var uidCount = 0L
1  // Version no atomica de getUnique
2  def getUniqueId() = {
3    val freshUid = uidCount + 1
4    Thread.sleep(1000)
5    uidCount = freshUid
6    freshUid
7  }
```

- Nótese que no se generan identificadores únicos.... Debemos asegurar atomicidad de *getUnique* (bloque sincronizado):

```
0  // Version atomica de getUnique
1  def getUniqueId() = synchronized {
2    val freshUid = uidCount + 1
3    Thread.sleep(1000)
4    uidCount = freshUid
5    freshUid
6  }
```

# Modelo de memoria

- El modelo de memoria es el conjunto de reglas que describen cómo interactúan los hilos cuando acceden a memoria compartida.
- El modelo de memoria de la JVM:
  - Dos hilos que escriben en ubicaciones separadas en la memoria, no necesitan sincronización
  - Un hilo *X* que invoca *join* sobre un hilo *Y* tiene garantizado observar todo lo escrito por el hilo *Y* una vez este termine.

# Abstracciones para estudiar concurrencia

Las abstracciones que usaremos en el curso para estudiar la programación concurrente se implementarán en términos de :

- Hilos
- Primitivas de sincronización como *synchronized*.

No sobra, conocer, qué es lo que hay por debajo.