PROGRAMACIÓN CONCURRENTE Y AVANZADA

CUADERNO DE PRÁCTICAS

P03. Threads

Concurrencia en la JVM y Modelo de Memoria de Java

Desde su creación, Scala se ejecuta sobre la JVM, y esto ha dirigido el diseño de muchas de sus bibliotecas de concurrencia. El modelo de memoria de Scala, su capacidad para multithreading, y la sincronización inter-thread son heredados de la JVM.

Procesos y threads

En sistemas operativos modernos, expulsores (pre-emptive), multi tarea, el programador tiene poco o ningún control sobre el procesador en que se ejecutará el programa. De hecho, el mismo programa puede ejecutarse en muchos procesadores diferentes durante la ejecución y a veces incluso simultáneamente en varios procesadores. A menudo, el **Sistema Operativo (SO)** se ocupa de asignar partes ejecutables del programa a procesadores específicos, mecanismo denominado **multitasking**, y esto es transparente para los usuarios.

Históricamente, multitasking se introdujo en los sistemas operativos para mejorar la experiencia del usuario, permitiendo a varios usuarios o programas usar los recursos del mismo ordenador simultáneamente. En cooperative multitasking, los programas eran capaces de decidir cuándo parar de usar el procesador y ceder el control a otros programas. Sin embargo, esto requería mucha disciplina por parte de los programadores y los programas podían dar fácilmente la impresión de no responder. Por ejemplo, un gestor de descargas que empieza a descargar un fichero debe tener cuidado al ceder control a otros programas. Bloquear la ejecución hasta que se complete una descarga arruinaría completamente la experiencia del usuario. La mayoría de los sistemas operativos actuales se basan en pre-emptive multitasking, donde a cada programa se le asignan secciones del tiempo de ejecución en un procesador específico. Estas secciones de tiempo se denominan rodajas de tiempo **(time slices)**. El multitasking se da de forma transparente para el programador de aplicaciones y para el usuario.

El mismo programa de ordenador puede arrancarse más de una vez, incluso simultáneamente dentro del mismo SO. Un **proceso** es una instancia de programa que se está ejecutando. Cuando un ordenador arranca, el SO reserva una parte de la memoria y otros recursos computacionales, y los asocia con un programa específico. El SO asocia el procesador con el proceso, y el proceso se ejecuta durante una rodaja de tiempo. En algún momento, el SO da el control del procesador a otros procesos. Es importante que la memoria y otros recursos de un proceso se aislen de los de otros procesos.

La mayoría de los programas consisten en un solo proceso, aunque algunos programas ejecutan varios procesos. En este caso, tareas diferentes del programa se expresan como procesos separados. Como distintos procesos no pueden acceder directamente a las mismas áreas de memoria, puede ser trabajoso expresar multitasking usando varios procesos.

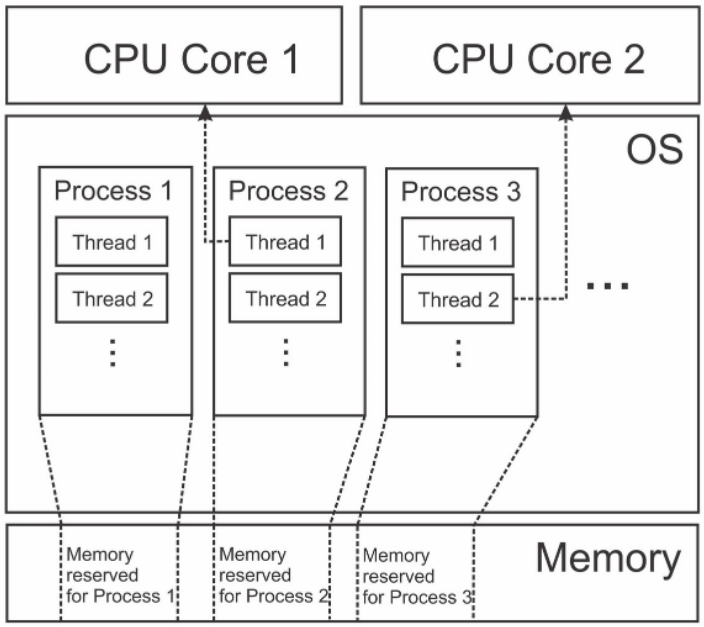
Multitasking era ya importante antes de los microprocesadores. Los programas grandes, como web browsers se dividían en muchas unidades lógicas. El gestor de descargas de un browser descarga ficheros independientemente de la representación (rendering) de la página web. Mientras el usuario navega en una página web de una red social, la descarga del fichero continúa en segundo plano, pero ambas computaciones ocurren como parte del mismo proceso. Estas computaciones independientes dentro del mismo proceso se llaman **threads**. En un sistema operativo típico, hay muchos más threads que procesadores.

Cada thread describe el estado actual de la pila **(stack)** y el contador de programa **(program counter)** durante la ejecución del programa. La pila del programa contiene una secuencia de llamadas a métodos que se están ejecutando actualmente, junto con las variables locales y los

parámetros de cada método. El contador de programa describe la posición de la instrucción actual en el método actual. Un procesador puede avanzar la computación en cualquier thread manipulando el estado de su pila o el estado de los objetos del programa, y ejecutando la instrucción señalada por el contador de programa actual. Cuando decimos que un thread realiza una acción tal como escribir en una posición de memoria, lo que estamos diciendo es que el procesador que ejecuta ese thread realiza dicha acción. En pre-emptive multitasking, la ejecución de un thread se planifica por el sistema operativo. Un programador debe aceptar que el tiempo de procesador asignado a su thread es similar al de otros threads del Sistema.

OS threads son un mecanismo de programación proporcionado por el SO, a menudo expuestos mediante una interfaz específica del sistema operativo. A diferencia de los procesos separados, los OS threads dentro del mismo proceso comparten un área de memoria, y se comunican escribiendo y leyendo partes de esa memoria. Otra forma de definir un proceso es como un conjunto de OS threads junto con la memoria y los recursos compartidos por esos threads.

Basándose en la discusión precedente sobre las relaciones entre procesos y threads, el siguiente diagrama proporciona una visión simplificada de un SO típico:



Processes and threads

El diagrama anterior muestra un SO en que se ejecutan varios procesos simultáneamente. Solo los 3 primeros procesos se muestran en la figura. A cada proceso se le asigna una zona fija de la memoria del ordenador. En la práctica, el sistema de memoria del SO es mucho más complejo, pero esta aproximación puede servir como modelo mental simple.

Cada uno de los procesos contiene varios OS threads, de los cuales se muestran dos por cada proceso. En este momento, el **Thread 1** del **Process 2** se está ejecutando en el **CPU Core 1**, y el **Thread 2** del **Process 3** se está ejecutando en el **CPU Core 2**. El SO asigna periódicamente los CPU cores a diferentes OS threads para permitir que avance la computación de todos los procesos.

Una vez mostrada la relación entre OS threads y procesos, ponemos nuestra atención en cómo se relacionan estos conceptos en la **Java Virtual Machine** (**JVM**), plataforma de ejecución sobre la que se ejecutan los programas Scala.

El arranque de una nueva instancia de JVM crea siempre un único proceso. Dentro del proceso de la JVM, pueden ejecutarse varios threads simultáneamente. La JVM representa sus threads

con la clase `java.lang.Thread`. A diferencia de las plataformas de ejecución de lenguajes como Python, la JVM no implementa sus propios threads, sino que cada Java thread se mapea con un OS thread. Esto significa que los Java threads se comportan de una forma muy similar a los OS threads, y que la JVM depende del SO y sus restricciones.

Creación y arranque de threads

Cuando se arranca un nuevo proceso JVM, crea varios threads por defecto. El thread más importante entre ellos es el **main thread**, que ejecuta el método `main` del programa Scala. Esto se muestra en el siguiente programa, que coge el nombre del thread actual y lo imprime en la salida estándar:

```
object ThreadsMain extends App {
  val t: Thread = Thread.currentThread
  val name = t.getName
  println(s"I am the thread $name")
}
```

En la JVM, los objetos thread se representan con la clase `Thread`. El programa anterior usa el método estático `currentThread` para obtener una referencia al objeto thread actual y lo almacena en una variable local `t`. Después llama al método `getName` para obtener el nombre del thread. Si ejecutas el programa, deberías ver la siguiente salida:

```
I am the thread main
```

Cada thread pasa por diferentes **thread states** durante su existencia. Cuando se crea un objeto `Thread`, inicialmente está en el estado **new**. Cuando el thread recién creado empieza a ejecutar, pasa al estado **runnable**. Cuando termina de ejecutar, el objeto thread pasa al **terminated state**, y ya no puede ejecutarse.

Arrancar un thread consta de dos pasos. Primero tenemos que crear un objeto `Thread` para reservar memoria para la pila y el estado del thread. Para empezar la computación, tenemos que llamar al método `start` de dicho objeto. Mostramos cómo hacer esto en la siguiente aplicación llamada `ThreadsCreation`:

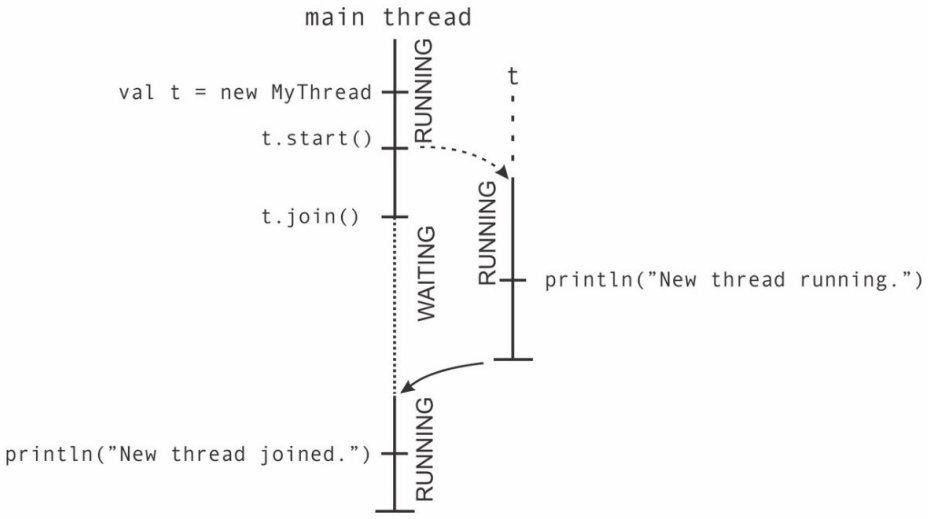
```
object ThreadsCreation extends App {
  class MyThread extends Thread {
    override def run(): Unit = {
      println("New thread running.")
    }
  }
  val t = new MyThread
  t.start()
  t.join()
  println("New thread joined.")
}
```

Cuando arranca una aplicación JVM, crea un thread especial llamado **main thread** que ejecuta el método llamado `main` de la clase especificada, en este caso, el objeto `ThreadsCreation`. Cuando se extiende la clase `App`, el método `main` se genera automáticamente con el cuerpo del objeto. En este ejemplo, el thread principal primero crea otro thread del tipo `MyThread` y lo asigna a `t`.

Después, el thread principal arranca `t` llamando al método `start`. Esta llamada al método `start` en algún momento ejecuta el método `run` del nuevo thread. Primero, se notifica al SO que `t` debe empezar a ejecutarse. El SO deberá asignar un procesador al nuevo thread en algún momento, pero esto está fuera del control del programador. Después de que el main thread arranque el nuevo thread `t`, llama al método `join`. Este método detiene la ejecución del main thread hasta que `t` completa su ejecución. Decimos que la operación `join` pone al main

thread en el **waiting state** hasta que `t` termina. El thread en espera libera el control sobre el procesador, y el SO puede asignarlo a otro thread.

Mientras, el SO busca un procesador disponible que ejecute el thread hijo y le ordena ejecutarlo. Las instrucciones que un thread debe ejecutar se especifican sustituyendo (overriding) su método `run`. La instancia `t` de la clase `MyThread` empieza imprimiendo el texto "New thread running." en la salida estándar y después acaba. En este punto, se notifica al sistema operativo que `t` ha terminado y en algún momento deja que el main thread continúe su ejecución. El SO pasa entonces al main thread de nuevo al estado running, y el main thread imprime "New thread joined." Esto se muestra en el siguiente diagrama:



Es importante notar que las dos salidas "New thread running." y "New thread joined." se imprimen siempre en este orden. Esto es porque la llamada a `join` asegura que la terminación del thread `t` ocurre antes que las instrucciones que siguen a la llamada a `join`.

La ejecución del programa es tan rápida que las dos sentencias `println` actúan casi simultáneamente. ¿Es posible que la ordenación de las sentencias `println` dependa de cómo decide el SO ejecutar los threads? Para verificar la hipótesis de que el main thread realmente espera a `t` y que esa salida no se debe solo a que el SO se decanta por `t` en este ejemplo concreto, podemos experimentar afinando el plan de ejecución. Antes de hacer esto, introduciremos una forma más corta de crear y arrancar un nuevo thread, ya que la sintaxis actual es demasiado verbosa. El nuevo método `thread` simplemente ejecuta un bloque de código en un thread recién creado. Esta vez crearemos un nuevo thread usando una clase thread anónima declarada en la propia instanciación.

Creamos un nuevo Scala Packet Object, llamado `de`, tal y como se indica a continuación:

```
package object de {
  def log(msg: String): Unit =
    println(s"${Thread.currentThread.getName}: $msg")
  def thread(body: =>Unit): Thread = {
    val t = new Thread {
      override def run() = body
    }
    t.start()
    t
  }
}
```

El método `thread` toma un bloque de código, crea un nuevo thread que lo ejecuta en su método `run`, arranca el thread, y devuelve una referencia al nuevo thread para que los clientes puedan llamar a `join` con ella. Crear y arrancar threads usando la sentencia `thread` es mucho menos verboso. Para hacer los ejemplos más concisos usaremos la sentencia `thread` a partir de ahora.

Ahora podemos experimentar con el SO para asegurarnos de que todos los procesadores están disponibles. Para ello, usaremos el método estático `sleep` de la clase `Thread`, que pospone la ejecución del thread actual durante el número especificado de milisegundos. Este método pone el thread en el estado **timed waiting**. El SO puede reusar el procesador para otros threads cuando se llama a `sleep`. Necesitaremos un tiempo de parada mucho mayor que la rodaja de tiempo (time slice) de un SO típico, que varía entre 10 y 100 milisegundos. El siguiente código es un ejemplo de esto:

```
import de._
object ThreadsSleep extends App {
  val t = thread {
    Thread.sleep(1000)
    log("New thread running.")
    Thread.sleep(1000)
    log("Still running.")
    Thread.sleep(1000)
    log("Completed.")
  }
  t.join()
  log("New thread joined.")
}
```

El main thread de la aplicación `ThreadSleep` crea y arranca un nuevo thread `t`, que se duerme un segundo y después visualiza algún texto, y lo repite varias veces antes de terminar. El main thread llama a `join` como antes y después imprime "New thread joined." El método `log` imprime el valor string especificado junto al nombre del thread que llama al método `log`.

Independientemente de cuantas veces ejecutes la aplicación anterior, la última salida será siempre "New thread joined.". Este programa es **determinista**: dada una entrada, siempre produce la misma salida, independientemente del plan de ejecución elegido por el SO.

Sin embargo, no todas las aplicaciones que usan threads darán siempre la misma salida para la misma entrada. El código siguiente es un ejemplo de aplicación **no-determinista**:

```
import de._
object ThreadsNondeterminism extends App {
  val t = thread {log("New thread running.")}
  log("...")
  log("...")
  t.join()
  log("New thread joined.")
}
```

No hay garantía de que las sentencias log("...") del main thread ocurran antes o después de la llamada a log del thread t. La ejecución repetida de la aplicación en un procesador multicore imprime "..." antes, después, o en medio de la salida del thread t. Ejecutando el programa, obtenemos la siguiente salida:

```
main: ...
Thread-0: New thread running.
main: ...
main: New thread joined.
```

Ejecutándolo de nuevo, los resultados salen en diferente orden:

```
main: ...
main: ...
Thread-0: New thread running.
main: New thread joined.
```

Atomic Execution

Ya hemos visto la forma básica en que los threads se comunican: se esperan uno a otro para acabar. La información que el joined thread entrega es que ha terminado. Además, el método `join` de los threads tiene una propiedad adicional. Todas las escrituras a memoria realizadas por el thread al que se hace join ocurren antes de que la llamada `join` retorne y son visibles para los threads que llaman al método `join`. Esto se ilustra en el siguiente ejemplo:

```
import de._
object ThreadsCommunicate extends App {
  var result: String = null
  val t = thread {result = "\nTitle\n" + "=" * 5 }
  t.join()
  log(result)
}
```

El main thread nunca escribirá `null`, ya que la llamada a `join` siempre ocurre antes que la llamada a `log`, y la asignación a `result` ocurre antes de la terminación de `t`. Este patrón es una forma muy básica en que los threads pueden usar sus resultados para comunicarse entre ellos. Observe que la variable `result` es compartida por ambos threads.

Sin embargo, este patrón solo permite una comunicación restringida en una sola dirección, y no permite a los threads comunicarse entre ellos durante su ejecución. Hay muchos casos de uso para una comunicación no restringida bidireccional. Un ejemplo es asignar identificadores únicos, en que un conjunto de threads eligen números concurrentemente de forma que dos threads diferentes no puedan producir el mismo número. Podríamos estar tentados de actuar como en el siguiente ejemplo incorrecto. Empezamos mostrando la primera mitad del programa:

```
object ThreadsUnprotectedUid extends App {
  var uidCount = 0L
  def getUniqueId() = {
    val freshUid = uidCount + 1
    uidCount = freshUid
    freshUid
  }
```

En el trozo de código anterior, primero declaramos una variable `uidCount` que contiene el último identificador único cogido por un thread. Los threads llamarán al método `getUniqueId` para calcular el primer identificador no usado y después actualizar la variable `uidCount`. En este ejemplo, la lectura de `uidCount` para inicializar `freshUid` y la asignación de `freshUid` de vuelta a `uniqueUid` no ocurren necesariamente juntas. Decimos que las dos sentencias no ocurren de forma atómica **(atomically)** ya que las sentencias de los otros threads pueden mezclarse de forma arbitraria. A continuación, definimos un método `printUniqueIds` tal que, dado un número `n`, el método llama a `getUniqueId` para obtener `n` identificadores únicos e imprimirlos. Usamos un `for` de Scala para mapear el rango `0 until n` a identificadores únicos. Finalmente, el main thread arranca un nuevo thread `t` que llama al método `printUniqueIds`, y después llama a `printUniqueIds` de forma concurrente con el thread `t` como sigue:

```
import de._
object ThreadsUnprotectedUid extends App {
```

```
  var uidCount = 0L
  def getUniqueId() = {
    val freshUid = uidCount + 1
    uidCount = freshUid
    freshUid
  }
  def printUniqueIds(n: Int): Unit = {
    val uids = for (i<- 0 until n) yield getUniqueId()
    log(s"Generated uids: $uids")
  }
  val t = thread {printUniqueIds(5)}
  printUniqueIds(5)
  t.join()
}
```

La ejecución de este programa varias veces revela que los indentificadores generados por los dos threads no son necesariamente únicos. A continuación se muestra lo que imprimen dos ejecuciones de la aplicación:

```
Thread-0: Generated uids: Vector(1, 2, 3, 4, 6)
main: Generated uids: Vector(1, 2, 3, 5, 7)
```

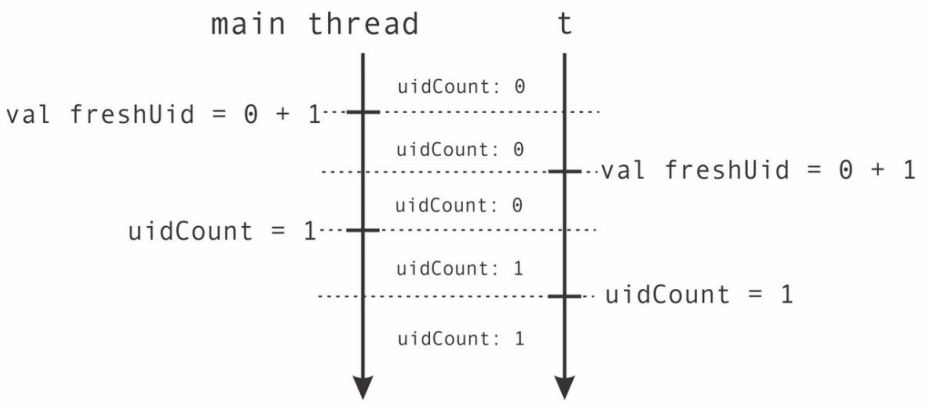
```
Thread-0: Generated uids: Vector(1, 3, 4, 5, 6)
main: Generated uids: Vector(2, 3, 4, 5, 6)
```

Las salidas del programa dependen de la temporización con que se ejecutan las sentencias de dos threads separados.

Nota

Una condición de carrera **(race condition)** es un fenómeno en que la salida de un programa concurrente depende de la planificación de ejecución de las sentencias del programa.

Una condición de carrera no es necesariamente un comportamiento incorrecto del programa. Sin embargo, si alguna planificación de la ejecución provoca una salida no deseada del programa, la condición de carrera se considera un error del programa. La condición de carrera del ejemplo anterior es un error del programa porque el método `getUniqueId` no es atómico. El thread `t` y el main thread a veces llaman concurrentemente a `getUniqueId`. En la primera línea, leen concurrentemente el valor de `uidCount`, que inicialmente es `0`, y concluyen que su variable `freshUid` debe ser `1`. La variable `freshUid` es local, por lo que se sitúa en la pila del thread; cada thread ve una instancia distinta de la variable. En este punto, los threads deciden escribir el valor `1` de vuelta a `uidCount` en cualquier orden, y ambos retornan un identificador no único `1`. Esto se ilustra en la figura siguiente:



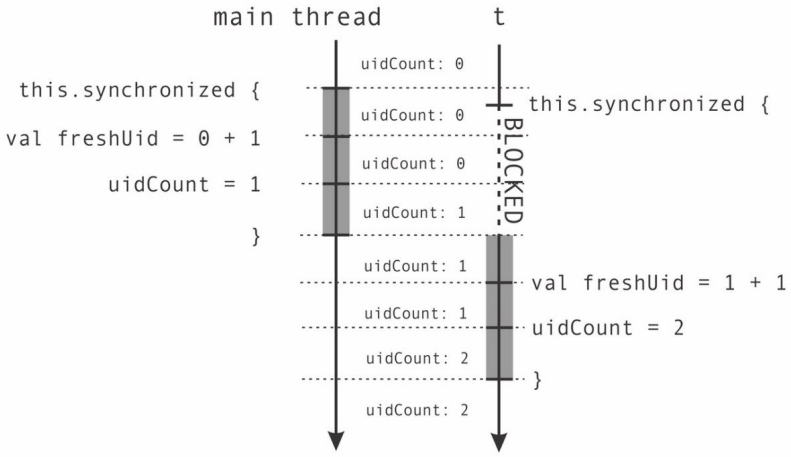
Hay una falta de correspondencia entre el modelo mental que la mayoría de programadores heredan de la programación secuencial y la ejecución del método `getUniqueId` cuando se

ejecuta concurrentemente. Esta diferencia se basa en la suposición de que `getUniqueId` se ejecuta atómicamente. La ejecución atómica de un bloque de código significa que las sentencias individuales del bloque ejecutado por un thread no pueden entremezclarse con las sentencias ejecutadas por otro thread. En la ejecución atómica, las sentencias solo pueden ejecutarse todas seguidas, que es exactamente como debería modificarse el campo `uidCount`. El código de la función `getUniqueId` lee, modifica, y escribe un valor, acciones que no son atómicas en la JVM. Se necesita un mecanismo especial en el lenguaje para garantizar la atomicidad. El mecanismo fundamental de Scala para garantizar la ejecución atómica es la sentencia `synchronized`, y puede utilizarse con cualquier objeto. Esta nos permite definir `getUniqueId` como sigue:

```
def getUniqueId() = this.synchronized {
  val freshUid = uidCount + 1
  uidCount = freshUid
  freshUid
}
```

La llamada a `synchronized` asegura que el bloque siguiente de código solo puede ejecutarse si no hay ningún otro thread ejecutando simultáneamente este bloque de código sincronizado y ningún otro bloque sincronizado que use el mismo objeto `this`. En nuestro caso, el objeto `this` es el singleton object que lo encierra, en este caso `ThreadsUnprotectedUid`, aunque en general, puede ser una instancia de la clase o trait que lo encierra.

La siguiente figura muestra dos llamadas concurrentes al método `getUniqueId`:



La JVM asegura que el thread que ejecuta una sentencia `synchronized` sobre un objeto `x` es el único thread que ejecuta una sentencia `synchronized` en ese objeto `x` particular. Si un thread `T` llama a `synchronized` sobre `x`, y hay otro thread `S` que llama a `synchronized` sobre `x`, el thread `T` se pone en estado **blocked**. En cuanto el thread `S` completa la sentencia `synchronized`, la JVM puede elegir el thread `T` para ejecutar su propia sentencia `synchronized`.

Los objetos creados dentro de la JVM tienen un elemento especial llamado **intrinsic lock** o **monitor**, que se usa para asegurar que solo un thread está ejecutando un bloque `synchronized` sobre ese objeto. Cuando un thread empieza a ejecutar un bloque `synchronized`, decimos que el thread **gains ownership** del monitor de `x`, o también, que lo **adquiere**. Cuando un thread completa el bloque `synchronized`, decimos que **libera (releases)** el monitor.

La sentencia `synchronized` es uno de los mecanismos fundamentales para la comunicación inter-thread en Scala y en la JVM. Cuando hay posibilidad de que varios threads accedan y modifiquen un campo de algún objeto, debe usarse la sentencia `synchronized`.

Reordering

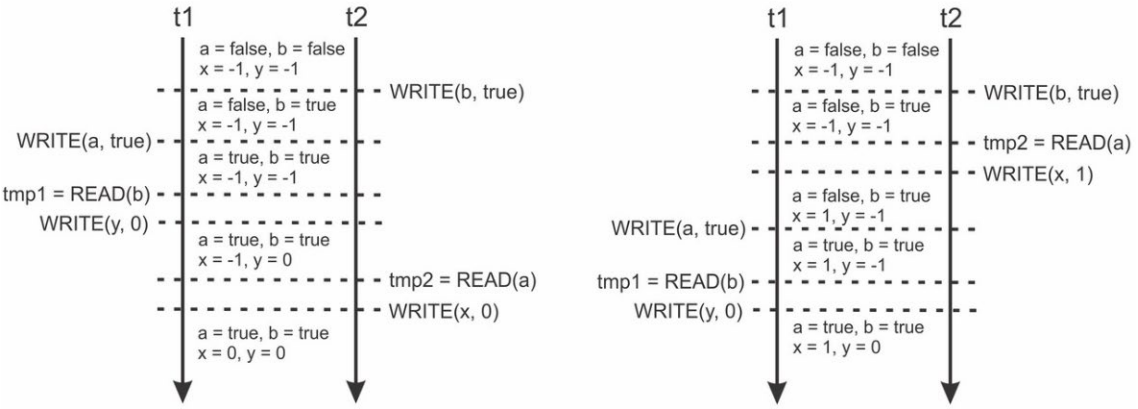
La sentencia `synchronized` tiene un precio: escribir en campos como `uidCount`, protegidos por la sentencia `synchronized`, es normalmente más caro que una escritura sin protección. La penalización en rendimiento de la sentencia `synchronized` depende de la implementación de la JVM, pero normalmente no es muy grande. Puedes sentirte tentado de evitar el uso de `synchronized` cuando piensas que no hay entrelazados (interleaving) peligrosos entre sentencias del programa como los que vimos en el ejemplo del unique identifier. ¡No lo hagas! Vamos a mostrarte un ejemplo mínimo que puede llevar a errores serios.

Consideremos el siguiente programa, en que dos threads, `t1` y `t2`, acceden a un par de variables Boolean, `a` y `b`, y a un par de variables Integer, `x` e `y`. El thread `t1` pone la variable `a` a `true`, y después lee el valor de la variable `b`. Si el valor de `b` es `true`, el thread `t1` asigna `0` a `y`, y si no, asigna `1`. El thread `t2` hace lo contrario: primero asigna `true` a la variable `b`, y después asigna `0` a `x` si `a` es `true`, y `1` si no. Esto se repite `100000` veces en un bucle, como se muestra en el siguiente trozo de código:

```
object ThreadSharedStateAccessReordering extends App {
  for (i <- 0 until 100000) {
    var a = false
    var b = false
    var x = -1
    var y = -1
    val t1 = thread {
      a = true
      y = if (b) 0 else 1
    }
    val t2 = thread {
      b = true
      x = if (a) 0 else 1
    }
    t1.join()
    t2.join()
    assert(!(x == 1 && y == 1), s"x = $x, y = $y")
  }
}
```

Este programa es algo sutil, por lo que debemos considerar diferentes escenarios muy cuidadosamente. Analizando los posibles entrelazados de las instrucciones de los threads `t1` y `t2`, podemos concluir que si ambos threads asignan simultáneamente a `a` y `b`, asignarán `0` a `x` y a `y`.

Esta salida indica que ambos threads empezaron casi al mismo tiempo, como se muestra a la izquierda de la figura siguiente:



Ahora, supongamos que el thread `t2` ejecuta más rápido. En este caso, el thread `t2` pone la variable `b` a true, y procede a leer el valor de `a`. Esto ocurre antes de la asignación a `a` por el thread `t1`, de forma que el thread `t2` lee el valor `false`, y asigna `1` a `x`. Cuando el thread `t1` se ejecuta, ve que el valor de `b` es `true`, y asigna `0` a `y`. Esta secuencia de eventos se muestra en la parte derecha de la figura anterior. Observa que el caso en que el thread `t1` empieza antes, lleva a una asignación similar donde `x = 0` e `y = 1`, por lo que no se muestra en la figura.

La conclusión es que, independientemente de cómo se ordene la ejecución de sentencias de los threads t1 y t2, la salida del programa no debería ser nunca x = 1 e y = 1 simultáneamente. Por ello, la aserción al final del bucle nunca debería provocar una excepción.

Sin embargo, tras ejecutar el programa unas cuantas veces, obtenemos la siguiente salida, que muestra que puede asignarse el valor 1 simultáneamente a x e y:

```
Exception in thread "main" java.lang.AssertionError: assertion failed: x = 1, y = 1
```

Este resultado choca y puede considerarse que atenta contra el sentido común. ¿Por qué no podemos razonar sobre la ejecución del programa como lo hemos hecho? La respuesta es que por la especificación de la JMM, la JVM tiene permitido reordenar ciertas sentencias de un programa ejecutadas por un thread siempre que no cambie la semántica secuencial de dicho thread. Esta es la razón por la que ciertos procesadores no siempre ejecutan las instrucciones en el orden del programa. Además, los threads no necesitan escribir todos sus cambios a memoria de forma inmediata, sino que pueden guardarlos temporalmente en registros caché dentro del procesador. Esto maximiza la eficiencia del programa y permite mejores optimizaciones del compilador.

¿Cómo deberíamos razonar sobre los programas multithread? El error que cometimos al analizar el ejemplo es que supusimos que las escrituras de un thread son inmediatamente visibles a los otros threads. Sin embargo, tenemos que aplicar la sincronización apropiada para asegurar que las escrituras de un thread son visibles a otro thread.

La sentencia `synchronized` es uno de los mecanismos fundamentales para conseguir la sincronización adecuada. Las escrituras de cualquier thread que ejecute la sentencia `synchronized` en un objeto `x` no solo son atómicas, sino también visibles a los threads que ejecuten `synchronized` sobre `x`. Encerrar todas las asignaciones de los threads `t1` y `t2` en una sentencia `synchronized` hace que el programa se comporte como esperamos.

Tip

Usa la sentencia `synchronized` sobre un objeto `x` cuando accedas (para leer o modificar) a un estado compartido entre varios threads. Esto asegura que como mucho un thread está

ejecutando en un momento dado una sentencia `synchronized` sobre `x`. Además, asegura que todas las escrituras a memoria del thread `T` son visibles a todos los otros threads que ejecuten después `synchronized` sobre el mismo objeto `x`.

Volatile variables

La JVM ofrece un mecanismo más ligero de sincronización que el bloque synchronized, llamado **volatile variables**. Las volatile variables pueden leerse y modificarse atómicamente, y se usan como status flags; por ejemplo, para señalar que una computación se ha completado o cancelado. Dichas variables tienen dos ventajas. La primera es que las escrituras y lecturas de volatile variables no pueden reordenarse dentro de un thread. La segunda es que la escritura en una volatile variable es inmediatamente visible a todos los otros threads.

Observa esta aplicación:

```
//from Professional Scala, p.184
object Volatile extends App {
 class CustomThread extends Thread {
   var flag = true
   override def run(): Unit = {
     while(flag) { }
     println("Thread terminated")
   }
 }
 val thread = new CustomThread
 thread.start()
 Thread.sleep(2000)
 thread.flag = false
 println("App terminated")
}
```

Hay un thread principal que lanza otro custom thread. En el método `run()` del custom thread puedes observar un `while loop` infinito, que no para hasta que alguien se lo dice. Esto se denomina “busy-waiting” y mantiene ocupado al CPU core. Por ello, no debes usar nunca un while dentro de un thread para esperar por algo.

El problema en esta aplicación es que cuando ejecutas `thread.flag = false`, el thread no verá el cambio y continuará esperando (“busy-wait”) a que algo cambie. Sin embargo, funcionará como se espera si, en vez de esto, marcas la variable con la anotación `@volatile`, como sigue: `@volatile var flag = true`. En este caso, el custom thread verá con éxito el cambio en la variable flag y terminará su ejecución.

Ejercicios

1. [AU] Implementa un método parallel que tome dos bloques de computación a y b y arranque cada uno en un thread. El método debe devolver una tupla con los valores resultantes de ambas computaciones. El método debe tener la siguiente sintaxis (signature):

```
def parallel[A, B](a: =>A, b: =>B): (A, B)
```

Los tipos `A` y `B` son genéricos y son el tipo del valor devuelto por cada bloque. Los dos parámetros de entrada, `a` y `b`, son funciones que no tienen parámetros y que devuelven un valor de tipo `A` (o `B`). En el programa principal puedes poner las siguientes expresiones para probar la corrección del programa:

```
val task1 = {3+1}
val task2 = {3*1}
println (parallel (task1, task2))
log ("FIN")
```

El `threadA` puede dejar su resultado sobre la variable `resultA` y el otro thread sobre otra variable `resultB`. Para devolver ambos valores (como una tupla) basta con poner, al final del método `parallel`, la expresión: `(resultA, resultB)`. Como ambas variables son genéricas, cuando se declaren puede inicializarse como:

```
var resultA: A = 1.asInstanceOf[A]
```

Sigue el siguiente esquema:

```
import de._
object ParallelBlock extends App {
  def parallel[A, B](a: => A, b: => B): (A, B) = {
    var resultA: A = 1.asInstanceOf[A]
    var resultB: B = 1.asInstanceOf[B]
    val threadA = thread {
      …
    }
    val threadB = thread {
      …
    }
    …
    (resultA, resultB)
  }
  val task1 = {3+4}
  val task2 = {3*1}
  println (parallel (task1, task2))
  log ("FIN")
}
```

2. [LA] Implementa un método periodically que tome un intervalo de tiempo duration en milisegundos y un bloque de computación b. El método arranca un thread (`worker`) que ejecuta el bloque de computación b cada duration milisegundos. Deberá tener la siguiente sintaxis:

```
def periodically(duration: Long, f: => Unit): Unit
```

Sigue el siguiente esquema:

```
object Periodically extends App {
  def periodically(duration: Long, f: => Unit): Unit = {
    val worker = new Thread {
      override def run(): Unit = {
      …
      }
```

```
    }
    worker.setName("Worker")
    worker.start()
  }
  periodically (1000,{println("ha pasado 1 segundo")})
}
```

3. [AV] A continuación, se presenta un método para ordenar una lista de elementos de tipo `T` en un orden dado (`less`)

```
def msort[T](less: (T,T) => Boolean)(xs: List[T]): List[T] = {
    def merge (xs: List[T], ys: List[T]): List[T] = {
      (xs, ys) match {
        case (Nil, _) => ys
        case (_, Nil) => xs
        case (x::xs1, y::ys1) =>
          if (less(x,y)) x::merge(xs1,ys) else y::merge(xs, ys1)
      }
    }
    val n = xs.length / 2
    if (n==0) xs
    else {
      val (ys, zs) = xs splitAt n
      merge (msort(less)(ys), msort(less)(zs))
    }
}
```

El algoritmo (merge sort) divide la lista en dos mitades, ordena cada mitad y mezcla ambas mitades ordenadamente. Queremos hacer esta ordenación concurrente por lo que se propone que cada mitad de la lista la ordene un thread distinto.

Genera una Scala App que, dado una lista de tipo T, la ordene según un criterio de entrada y que dicha ordenación proceda de manera concurrente. Por ejemplo:

```
val r = msort ((x:Int, y:Int)=> x < y)(List(5,7,1,3))
println (r)
List(1, 3, 5, 7)
```

Inicialmente puedes suprimir el tipo genérico y hacer una ordenación ascendente (de menor a mayor).

¿Cuántos threads participan en la ordenación? ¿Por qué?

Currying. Esto solo tiene que ver con el lenguaje Scala y no con aspectos de concurrencia. Este concepto se refiere a la forma de expresar los parámetros en la función `msort`. Observa que hay dos listas de parámetros en lugar de una lista con dos parámetros.

```
def msort[T](less: (T,T) => Boolean)(xs: List[T]): List[T]
```

Se indica que los elementos a ordenar son de un tipo genérico `T`. La primera lista indica la función de ordenación: una función que toma dos elementos de tipo `T` y devuelve un resultado `Boolean`. La segunda lista indica la lista de números a ordenar según el criterio especificado en la primera lista. Un ejemplo de función que ordena de menor a mayor se indicó anteriormente:

```
((x:Int, y:Int)=> x < y)
```

El tipo de los elementos a ordenar es `Int` y la expresión que devuelve el boolean es: `x < y`. En la segunda lista de parámetros aparecen los números a ordenar: `List(5,7,1,3)`.

Al expresar los parámetros en dos listas, podemos definir la función `msort` especificando inicialmente solo el método de ordenación. Por ejemplo:

```
scala > val intSort = msort ((x:Int, y:Int)=> x < y) _
intSort: List[Int] => List[Int] = <function1>
```

La variable `intSort` se refiere a una función que toma una lista de enteros y los ordena ascendentemente. Observa que no se especifica el segundo parámetro (_). Cuando se quiera ordenar una lista en orden ascendente basta con invocar:

```
scala> val numeros = List (4,1,9,5)
numeros: List[Int]=List(4,1,9,5)
scala> inSort(numeros)
res0=List[Int]=List(1,4,5,9)
```