Programación Concurrente
Programación Concurrente Multihilos - Java
Parte I

Programación Orientada a Objetos Concurrente

“hacer mas de una cosa a la vez”

Cuande se habla de concurrencia en programación, se habla de la técnica de hacer que un programa haga mas de una cosa a la vez.
Si bien esquemas mas distribuidos generan una mejor distribución del trabajo, también provocan mayor necesidad de comunicación.

Concurrencia Vs Paralelismo

Un sistema se dice concurrente si puede soportar 2 o mas actividades en progreso a la vez. Un programa concurrente tiene múltiples hilos de control lógicos que pueden o no ejecutarse en paralelo.
Un sistema se dice paralelo si puede soportar 2 o mas actividades ejecutándose simultaneamente. Un programa paralelo puede ejecutarse mas rápidamente que un programa secuencial ejecutando diferentes partes simultáneamente. Puede o no tener mas de un hilo de control lógico.
Se podría pensar que la “concurrencia” es un aspecto relacionado al dominio del problema y el “paralelismo” es un aspecto del dominio de la solución. En un caso se necesita manejar situaciones simultáneamente, y en el otro se quiere hacer un programa mas rápido procesando diferentes porciones del problema en paralelo.

> *Concurrencia no es paralelismo, es mejor. ... Concurrencia es programar como la composición de procesos que se ejecutan de forma independiente. Paralelismo es programar como la ejecución simultánea de computaciones, posiblemente relacionadas.*
> *Concurrencia es sobre tratar con muchas cosas a la vez*
> *Paralelismo es sobre hacer muchas cosas a la vez.*
> *Concurrencia es sobre estructura, y paralelismo es sobre ejecución.*
> *La concurrencia provee una forma de estructurar una solución para resolver un problema que puede ser paralelizable (pero no necesariamente lo es).*
> *La comunicación se usa para coordinar las ejecuciones independientes*

(Rob Pike, http://concur.rspace.googlecode.com/concur.html ver slides14..25)

Consideremos la siguiente situación:

> Un docente, un grupo de alumnos, una clase. El docente es un “master en multitarea”, en cada instante esta realizando una única tarea pero tiene que tratar con varias de forma concurrente: mientras espera que el proyector reconozca a la notebook, escribe un ejercicio en la pizarra, contesta a la pregunta de un alumno, registra que otro alumno lo esta llamando, etc. **Esto es concurrente, pero no es paralelo**. En caso de que haya 2 auxiliares en la clase, el aux1 escribiendo el ejercicio en la pizarra y el aux2 preparando la proyecciòn mientras el docente contesta la pregunta de un alumno entonces sí seria además de concurrente, paralelo.

Una aplicación concurrente puede tener 2 o mas hilos de ejecución en progreso al mismo tiempo. Esto puede significar que la aplicación tiene 2 hilos que estan siendo intercambiados in/out por el sistema operativo en un procesador de un núcleo. Estos hilos estarán en progreso en el medio de su ejecución. En cambio si pensamos en la ejecución en paralelo debe haber multiples núcleos disponibles en la plataforma. Asi a cada hilo se le puede asignar un núcleo separado y se ejecutarán efectivamente de forma simultánea.

Escribir programas correctos es dificil, escribir programas concurrentes correctos es mas dificil. Hay que considerar comunicación y sincronización

El modelo de programación secuencial es intuitivo y natural, modela la forma en que trabajan los humanos: hacer una cosa por vez, mayormente en secuencia. Levantarse de la cama, ponerse la bata de baño, bajar la escalera, tomar el te. Cada una de estas acciones del mundo real es una abstracción para una secuencia de acciones mas finas (abrir la caja de te, seleccionar un saquito de te, poner el saquito de te en la taza, ver si hay suficiente agua en la pava, encender la cocina, poner la pava con agua a hervir, etc.). Mientras se espera que el agua hierva (o este lo suficientemente caliente) se puede hacer algo mas: solo esperar, o hacer otras tareas como preparar una tostada, exprimir naranjas para hacer jugo, encender la computadora para leer el diario, encender el televisor para ver algo, ... mientras se mantiene la atención en la pava que pronto estará lista. Los fabricantes de pavas y tostadoras saben de esta situación, por lo que las producen para que cuando la tarea esta completa (el agua lista o la tostada lista) emitan un sonido para señalar el hecho. Estas tareas envuelven un grado de asincronismo.

Uno de los desafíos de desarrollar programas concurrentes en Java es la discordancia entre las características de concurrencia ofrecidas por la plataforma y como los desarrolladores necesitan pensar acerca de la concurrencia. Los hilos (procesos livianos) permiten que varias corrientes de flujo de control de programas coexistan dentro de un proceso. Los hilos comparten los recursos del proceso (memoria, etc), pero tienen su propio contador, pila y variables locales. SI NO EXISTE **coordinación explícita** entre los hilos, se ejecutan de forma simultánea y asíncrona con respecto a los otros hilos

Hay tres aspectos de la concurrencia que requieren atención:

- distintas tareas compiten por los recursos (se entiende por recurso todo lo que necesita un proceso para operar: espacio de memoria, dispositivos de E/S, etc). En este sentido se debe garantizar que todos los procesos puedan acceder a los recursos necesarios para seguir procesando en un tiempo finito. Es decir se debe asegurar el progreso de todos los procesos: viveza (o liveness)
- respecto a los tipos de métodos, cuando hay concurrencia las operaciones de consulta provenientes de distintas tareas pueden hacerse en cualquier orden sobre el mismo objeto. En cambio, la intervención de operaciones de modificación puede alterar los resultados.
- respecto a los invariantes, puede ser bastante mas difícil de garantizar su cumplimiento cuando varios procesos o hilos pueden acceder simultáneamente a un mismo objeto y modificarlo.

Construir programas concurrentes correctos requiere el uso correcto de hilos y bloqueos. Es necesario proteger los datos de un acceso concurrente descontrolado. Importa COMO es usado el

objeto, no lo que hace.

"Cada vez que mas de un hilo accede a una variable de estado dada, y uno de ellos puede escribir en ella, TODOS deben coordinar su acceso a ella utilizando sincronización..."

"Debe evitarse la tentación de pensar que hay situaciones especiales en las cuales la regla no se aplica. Un programa que omite la sincronización necesaria podría parecer que trabaja, pasar las pruebas, y ejecutarse bien durante años, pero ... aún esta roto y puede fallar en cualquier momento"

Condición de carrera: ¿Quién gana la carrera para …?
¿Qué es una condición de carrera?

El tipo mas comun es **"chequear y luego actuar"**
Ejemplo: supongamos que planeamos encontrarnos con un amigo, al mediodía en XXX, en la Avenida Argentina, pero cuando usted llega alli descubre que hay 2 XXX en esa avenida, una de cada lado de la calle y no está seguro en cual quedaron en encontrarse. A las 12.10 no ve a su amigo en el XXX A, entonces camina hasta el XXX B para ver si está su amigo, pero tampoco está allí.
Hay algunas posibilidades:

- su amigo se retraso, y no está en ningún XXX todavía,
- su amigo llego al XXX A después que usted se fue
- su amigo estaba en el XXX B pero salió a buscarlo y ahora esta rumbo al XXX A

En el caso de la última opción:
Ahora son las 12.15 y ambos están pensando que hacer, volver a ir al otro XXX, ¿cuántas veces ir y venir? A menos que hayan acordado un protocolo, podrán pasar el resto del día caminando a lo largo de la avenida, frustados y con mucho café encima.

El problema: la frase "Yo cruzaré la calle para ver si él está en el otro XXX" es que mientras usted está caminando, cruzando la calle, su amigo puede haberse movido. Usted observa alrededor en XXX A y piensa "él no está aquí", y se va buscándolo, y sucede lo mismo con XXX B... ***pero no al mismo tiempo***. Toma unos minutos cruzar la calle y durante ese tiempo ***el estado del sistema puede haber cambiado.***

Este ejemplo ilustra una condición de carrera, porque alcanzar la salida deseada (encontrarse con su amigo) depende del temporizado relativo de los eventos (cuando usted arriva a XXX A, cuanto espera alli antes de cambiar?). La observación de que “él no esta en XXX A” se torna potencialmente inválida tan pronto como usted sale de XXX A. El podria haber ingresado por otra puerta sin que usted lo sepa.

Esta invalidación de la observación es lo que caracteriza la mayoría de las condiciones de carrera: - observar algo que sea T (archivo X no existe), y tomar una acción basándose en esa observación (crear X), pero la observación puede ser invalidada entre el tiempo que se hizo la observación, y el tiempo en que se actuó sobre ella (alquien mas creo el archivo X), causando un

problema (excepción inesperada, archivo corrupto, datos sobreescritos, etc.).

Otro tipo de condición de carrera: operaciones **"lee-modifica-escribe",** como incrementar un contador, define una transformación del estado del objeto en términos de su estado previo. Para incrementar un contador se debe conocer su valor previo y asegurarse que nadie mas lo cambie o use mientras usted esta en medio de su actualización.

Como la mayoría de los errores de concurrencia, las **condiciones de carrera** no siempre resultan en fallas, se requiere tambien un poco de malasuerte, pero estas pueden causar problemas graves

Aplicaciones concurrentes*:* servidores web, cálculos numéricos pesados, procesamiento de E/S, simulación, aplicaciones basadas en GUI, software basado en componentes, código móvil, sistemas embebidos de tiempo real. …

Planificación de hilos

Los hilos realizan distintas tareas en las aplicaciones, a las que se puede asignar diferentes niveles de prioridad. Cada hilo tiene una prioridad, que es asignada por la máquina virtual al momento de su creación, y que por defecto es la misma prioridad que tiene el hilo desde el que es creado (varia entre 1 y 10). En el caso de Java será la prioridad del hilo que comienza la ejecución, es decir el que ejecuta el main.

Existen los hilos de ejecución "*demonio" que* se ejecutan, normalmente, con prioridad baja y proporcionan servicios básico a un programa cuando la actividad de la máquina es reducida. Por ejemplo el hilo recolector de basura (garbaga collector) es un hilo demonio que es proporcionado por la JVM y se ejecuta continuamente, recuperando las variables que no están en uso, liberando esos recursos.

Puede haber varios hilos en estado listo (runnable), es decir listos para su ejecución, pero sólo 1 se encuentra en ejecución. El hilo que se encuentra en ejecución puede cambiar su estado a *bloqueado* o volver a estado *listo*. El cambio a listo se produce cuando termina su tiempo de CPU y se porduce un desbloqueo. El cambio a bloqueado puede darse en diversas situaciones: por ejemplo porque quiere ejecutar una sección de código para la que no esta habilitado, requiere un recurso qu eno esta disponible, voluntariamente deja la cpu por un período de tiempo ….
Los hilos que están en estado listo se mantienen en una colección, organizados segun su prioridad.
Cuando un hilo es desbloqueado vuelve a esa colección de listos.

Luego, un hilo durante su tiempo de vida puede estar en uno de los siguientes estados:

- **creado**, esperando pasar a estado listo
- **listo/runnable**, habilitado para ser ejecutado
- **en ejecución**, hasta que termina o es bloqueado.
- **bloqueado**, esperando ser desbloqueado.

– **muerto**, cuando termina su ejecución

(Mas adelante veremos distintas situaciones de bloqueo).

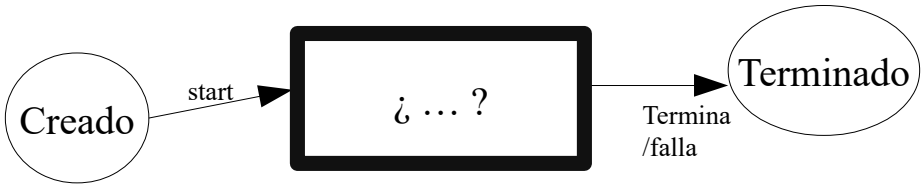
Hilo/Thread: proceso liviano (un hilo de control) La unidad de concurrencia es el Thread, comparte el mismo espacio de variables con los restantes Threads. Se utiliza la clase Thread.

– Un hilo en Java se crea instanciando la clase Thread – El código que ejecuta un hilo está definido por el método run() que tiene todo objeto que sea instancia de la clase Thread

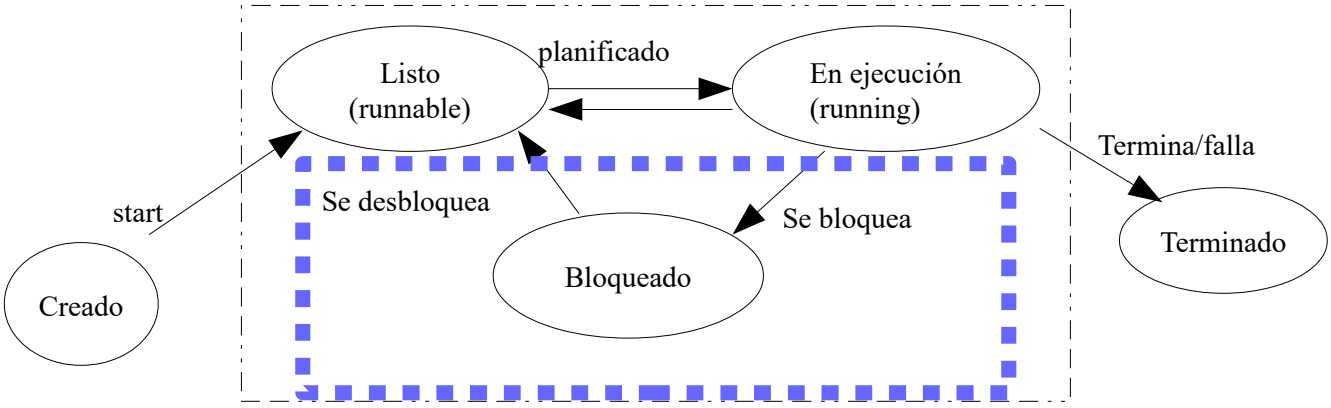
La ejecución del thread se inicia cuando sobre el objeto Thread se ejecuta el método start().

De forma natural, un thread termina cuando en run() se alcanza una sentencia return o el final del método.

(…)



Ahora ampliamos el diagrama, completando la versión anterior con los estados intermedios en los que se puede encontrar un hilo durante su tiempo de vida.



Luego ampliaremos las opciones de “bloqueado”.

Si bien es posible asignar y modificar las prioridades de los hilos, no es aconsejable. No es conveniente basar el diseño de un algoritmo en la prioridad de los hilos. Al escribir código en un ambiente “multithreaded”, que se ejecute de forma correcta en diferentes plataformas, se debe contemplar que “un hilo puede ser desalojado de la CPU en cualquier momento, por lo que se debe trabajar el acceso a los recursos compartidos de manera protegida.

Entonces … como se trabaja …?

Opcion 1: extender la clase Thread.

- Definir una subclase de Thread
- Implementar el método run(), que contendrá las acciones que debe realizar el hilo
- En el main crear las instancias necesarias de la subclase definida

Ejemplo

```
public class HiloAlfaBeta extends Thread {
   int cantidad;

   public HiloAlfaBeta(String nombre, int laCantidad){
      super(nombre);
      this.cantidad = laCantidad;
   }

   public void run(){
      for (int i=1; i<this.cantidad; i++){
         System.out.println(this.getName() + " en ejecución");
         System.out.println(Thread.currentThread() + "-------" +
               Thread.currentThread().getName());
      }
   }
}
```

```
public class TestAlfaBeta {
    public static void main (String[] args){
        HiloAlfaBeta alfa = new HiloAlfaBeta("Hilo Alfa", 100);
        HiloAlfaBeta beta = new HiloAlfaBeta("Hilo Beta", 10);

        alfa.start();
        beta.start();

        System.out.println("Probando hilos");
    }
}
```

Opción 2: utilizar la interfaz Runnable

- Definir una clase que implemente la interfaz Runnable
- Implementar el método run() que contendra las acciones que debe realizar el runnable
- En el main:
  - crear las instancias de runnable necesarias
  - crear las instancias de Thread necesarias a partir de los runnables

```
public class RunnableAlfaBeta implements Runnable{

   int cantidad;

   public RunnableAlfaBeta(int laCantidad){
      this.cantidad = laCantidad;
   }

   public void run(){
      for (int i=1; i<cantidad; i++){
         System.out.println(Thread.currentThread().getName() + " en ejecución");
      }
   }
```

```
public class TestAlfaBeta {
  public static void main (String[] args) {

      RunnableAlfaBeta alfaBetaRunnable_1 = new RunnableAlfaBeta(100);
      RunnableAlfaBeta alfaBetaRunnable_2 = new RunnableAlfaBeta(100);


      Thread alfa = new Thread (alfaBetaRunnable_1, "Hilo Alfa");
      Thread beta = new Thread (alfaBetaRunnable_2, "Hilo Beta");

      alfa.start();
      beta.start();
      System.out.println("entoy saliendo del main");

  }
}
```

Algunas características de los hilos en Java:

• Un objeto Thread mantiene el control de estas actividades.
• Los hilos pueden compartir el acceso a memoria, archivos abiertos, y otros recursos. Se podria decir que son procesos “livianos”.
• un hilo NO puede ser reiniciado.
• las prioridades asignadas (entre 1 y 10) a los hilos afectan al planificador.
• El main tiene prioridad 5. El lenguaje no asegura planificación o justicia, ni siquiera garantiza que los hilos progresen. LA POLITICA EXACTA APLICADA POR EL PLANIFICADOR PUEDE VARIAR SEGUN LA PLATAFORMA, ALGUNAS MAQUINAS VIRTUALES SIEMPRE SELECCIONAN EL HILO CON MAYOR PRIORIDAD CORRIENTE. EXISTEN VARIAS POSIBILIDADES. EL SETEO DE PRIORIDADES NO DEBE SUSTITUIR AL BLOQUEO.
• Para determinar la prioridad actual de un hilo se utiliza el método getPriority
• los métodos sleep, yield, interrupted y currentThread son “estáticos” y se aplican sobre el hilo en ejecución.
• El método yield tiene sentido efectivo en presencia de un planificador que no es “timesliced” preemptivo. Coloca al hilo en ejecución (el llamador) en el pool de hilos listos y da lugar para que otro hilo listo se ejecute.
• Al momento de la implementación debemos independizarnos de la política de planificación que sea utilizada por la JVM.
• Se puede utilizar el método isAlive para determinar si un hilo todavia esta disponible.

… continuará

Bibliografia

• **Concurrent Programming in JAVA: design principles and patterns** - Doug Lea
• **Thinking in JAVA** - Bruce Eckel - President, MindView Inc. - 2008
• **Seven Concurrency Models in Seven Weeks**. When Threads Unravel - Paul Butcher - The Pragmatic Programmers Inc. - 2014
• **Java Concurrency in Practice** - Brian Goetz - et all - Addison Wesley 2006.
• **Documentacion Oracle**: http://docs.oracle.com/javase/tutorial/essential/concurrency/ y el paquete java.util.concurrent.
• **Orientación a Objetos con Java y UML** - Carlos Fontella - nueva libreria nl 2004