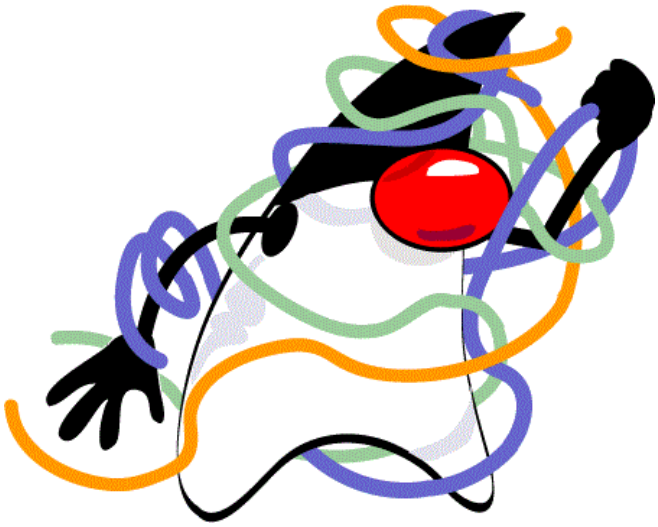
UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS



Concurrencia.

2013
Transversal Programación Orientada a Objetos
Proyecto Curricular de Ingeniería de Sistemas

Introducción

Cuando programamos procesos que se puedan ejecutar al mismo tiempo "en paralelo", qué mecanismo podemos implementar para poder controlar estos procesos.

En la programación orientada a objetos se puede dividir un programa en secciones independientes mediante los objetos, pero esto muchas veces no es suficiente y se hace necesario dividirlos también en subtareas separadas se puedan ejecutar independientemente y en paralelo, cada una de estas subtareas independientes recibe el nombre de hilo, y la idea es programar cada hilo como si se ejecutara por sí mismo y tuviera todos los recursos de máquina para él sólo.

En java existen mecanismos que divide el tiempo de CPU entre ellos sin que el programador tenga qué pensar en ello, lo que hace de la programación de hilos múltiples una tarea más sencilla.

1. Crear un nuevo hilo.

Cuando se inicia un programa Java, la máquina virtual crea un hilo inicial, que se encargará de llamar al método `public static void main(String[] args)` de la clase que se comienza a ejecutar.

Si se desea crear un nuevo hilo, es suficiente construir un nuevo objeto de la clase `Thread` y llamar a su método `start()`. Esa llamada ocasionará que se llame al método `run()` del hilo, pero en un hilo de ejecución nuevo, diferente al que ejecutó la llamada al método `start()`. La ejecución del método `run()` será, por lo tanto, "simultánea" a la ejecución del código que siga a la llamada a `start()`.

La implementación del método `run()` implementada en la clase Thread es vacía, no hace nada. Para construir un hilo un poco más interesante, habrá que implementar una clase que extienda a la clase `Thread`, y sobreescriba el método `run()`. Debido a la invocación dinámica propia de la programación orientada a objetos, la llamada al método `start()` de la clase recién creada llamará al nuevo método `run()` implementado. Para construir un hilo, por lo tanto, bastará con meter en el método `run()` el código que deseemos que se ejecute de forma independiente en un nuevo hilo.

Veamos un ejemplo. El siguiente programa:

```
public class DosHilosBasicas extends Thread {
	int cont;
	DosHilosBasicas(int c) {
		cont = c;
	}
	public void run() {
		while (true) {
			system.out.println(cont);
		}
	}
	public static void main(String[] args) {
		new DosHilosBasicas(0).start();
		new DosHilosBasicas(1).start();

	}

}
```

La ejecución de la aplicación anterior causa la salida por pantalla de líneas con 0's y 1's entrelazados. Eso se debe a que la máquina virtual distribuye el tiempo del procesador entre los dos hilos, de modo que según quién tenga el procesador se escribirá un 0 o un 1.

Este modo de implementar nuevos hilos tiene una desventaja. Debido a la restricción de herencia simple impuesta por Java, sería imposible hacer clases ejecutables en hilos independientes que hereden de cualquier otra clase. Esto ocurre por ejemplo en applets. No es posible implementar la aplicación anterior en un applet de forma directa, debido a que la clase tendría que heredar simultáneamente de la clase Thread y de la clase Applet.

Afortunadamente existe el interfaz Runnable. Una clase que desee implementar este interfaz deberá disponer del método: public void run();

Es posible comenzar la ejecución del método run() de cualquier objeto que implemente el interfaz en un hilo independiente. Para eso, es suficiente crear un nuevo objeto de la clase Thread, pasándole en el constructor el objeto que se desea ejecutar. Cuando se llame al método start() de ese hilo, el método que se ejecutará será, realmente, el método run() del objeto pasado en el constructor. El ejemplo anterior se implementaría:

```
public class DosHilosBasicas implements Runnable {
	int cont;
	DosHilosBasicas(int c) {
		cont = c;
	}
	public void run() {
		while (true) {
			system.out.println(cont);
		}
	}
	public static void main(String[] args) {
		new Thread (new DosHilosBasicas(0)).start();
		new Thread (new DosHilosBasicas(1)).start();
	}
}
```

Ahora la clase no hereda de ninguna otra, por lo que la herencia queda libre para heredar, por ejemplo, de la clase Applet. También se modifica la forma de crear los hilos. Ahora es necesaria la creación de dos objetos: el objeto que implementa el interfaz, y el propio hilo que lo ejecutará.

El método start() del hilo realmente sigue ejecutando el código del método run() del objeto de la clase Thread. Se ejecutará el método run() del objeto que implementa el interfaz Runnable porque el código del run() del hilo no está realmente vacío. En el siguiente código se esboza la parte del código de la clase Thread que nos interesa ahora mismo:

```
public class Thread implements Runnable {
	/* ... */
	protected Runnable _target;
	/* ... */
Thread(Runnable target) {
	_target = target;
}
/* ... */
public void run() {
	if (_target != null)
		_target.run();
	}
/* ... */
}
```

Puede verse que el método run() del hilo se encarga de llamar al método run() del objeto que implementa el interfaz Runnable, si es que se ha especificado alguno en el constructor. Si no es así, el método no hace nada.

También puede verse que la clase Thread implementa el interfaz Runnable, y contiene el método run() que es el que se llama cuando se invoca al método start(), tal y como se ha dicho antes. Pero eso no debe llevar a confusión con el uso del interfaz Runnable en cualquier otra clase.

La ejecución es completamente igual en los dos casos. No hay reglas de cuando usar una u otra técnica. No obstante, el uso de la herencia es solo válido cuando la clase que se desea ejecutar no tiene que heredar de ninguna otra clase. Además, se aconseja que se use herencia únicamente si hay que sobreescribir algún otro método, además del run(). Por su parte, el uso del interfaz Runnable requiere la construcción de dos objetos (el hilo y el propio objeto) para poder realizar la ejecución.

2. Finalización de un hilo.

La finalización de un hilo suele ocurrir cuando se termina de ejecutar el método run(). No obstante, hay otras tres razones por las que un hilo puede terminar:

- En algún momento de la ejecución del método run() se ha generado una excepción que nadie ha capturado. La excepción se propaga hasta el propio método run(). Si tampoco éste tiene un manejador para la excepción, el método run() finaliza abruptamente, terminando la ejecución del hilo.
- Si se llama al método stop() o stop(excepción) del hilo. Estos dos métodos originan que el hilo termine, y son en realidad un caso particular del anterior.
- Cuando se llama al método destroy() del hilo.

En cualquiera de los casos, el hilo finaliza su ejecución, y deja de ser tenida en cuenta por el planificador.

Prioridades

El planificador es la parte de la máquina virtual que se encarga de decidir qué hilo ejecutar en cada momento. La especificación de la máquina virtual no fuerza al uso de ningún algoritmo particular en la planificación de hilos. De hecho, es problema de cada implementación particular decidir si implementar por sí misma el planificador, o apoyarse en el sistema operativo subyacente.

En cualquier caso si se fuerzan ciertas características que debe tener el planificador. En concreto, todos los hilos tienen una prioridad, de modo que el planificador dará más ventaja a los hilos con mayor prioridad que a las de menos.

En principio, se obliga a que exista expropiación entre hilos de igual prioridad, de modo que si hay varios hilos con igual prioridad todos se ejecuten en algún momento. La especificación de la máquina virtual no obliga, sin embargo, a la expropiación de hilos de una prioridad mayor si el hilo que va a pasar a ejecutarse es de prioridad menor. Es decir, no se garantiza que hilos de prioridad baja pasen a ejecutarse si existe algún hilo de mayor prioridad.

Un sencillo ejemplo que puede utilizarse para comprobar esto es el siguiente:

```
public class ComprobarPrioridad implements Runnable {
	int num;
	boolean parar = false;

	ComprobarPrioridad(int c) { num = c; }
		public void run() {
			while (!parar) {
				system.out.println(num);
			}
		}
public static void main(String[] args) {
	Thread nueva;
	for (int c = 0; c < 10; c++) {
		nueva = new Thread(new ComprobarPrioridad(c));
			if (c == 0) nueva.setPriority(Thread.MAX_PRIORITY);
				nueva.start();
		}
}

}
```

Si la máquina virtual que ejecute este ejemplo no realiza expropiación en hilos de mayor prioridad para ejecutar hilos de menor prioridad, solo se ejecutará el hilo número 0, por lo que sólo se mostrará ese número en la salida estándar. Si sí se realiza expropiación, otras hilos se ejecutarán, pero, en principio, una menor cantidad de veces que el hilo 0.

Del código anterior se pueden deducir algunas constantes y funciones definidas en la clase Thread para controlar la prioridad. Se definen dos funciones:

- setPriority(int): establece la prioridad del hilo. Puede ocasionar la generación de una excepción de seguridad si el hilo que solicita el cambio de prioridad de otra no está autorizada a hacerlo.
- getPriority(): devuelve la prioridad del hilo.

Para establecer los posibles valores en el parámetro de setPriority o como resultado de getPriority, la clase Thread define tres constantes estáticas a la clase:

- MAX_PRIORITY (= 10): es el valor que simboliza la máxima prioridad.
- MIN_PRIORITY (= 1): es el valor que simboliza la mínima prioridad.
- NORM_PRIORITY (= 5): es el valor que simboliza la prioridad normal, la que tiene el hilo creado durante el arranque de la máquina virtual y que se encarga de ejecutar la función main().

Las librerías gráficas de Java AWT y Swing construyen su propio hilo, que se encargará de atender los eventos del usuario (ya sean del ratón o del teclado). Cuando, por ejemplo, se pulsa un botón, es ese hilo el que lo recoge, y la que se encarga de ejecutar el código asociado al evento.

El usuario espera que sus órdenes se ejecuten instantáneamente, lo que da la impresión de un sistema rápido. Al menos, es deseable que cuando se pulse un botón, éste modifique momentáneamente su aspecto para darle al usuario la sensación de que el sistema se ha dado cuenta de su acción. Para que esto pueda realizarse, los desarrolladores de las librerías anteriores decidieron establecer a el hilo que crean una prioridad ligeramente superior a la normal (=6). Gracias a eso se logra que la interfaz gráfica responda inmediatamente, incluso aunque haya otros hilos normales ejecutándose. Cuando el usuario no realiza ninguna acción, el hilo del interfaz estará suspendido, esperando a que lleguen eventos, momento en el que se permite a esos otros hilos ejecutarse.

Naturalmente, si el programa crea un hilo con una prioridad mayor que la establecida al hilo del interfaz, ésta podría dejar de responder rápidamente.

Otros métodos para el planificador.

Además del uso de las prioridades, la clase Thread implementa otros métodos con los que se puede conseguir algo de control sobre el comportamiento que toma el planificador respecto a hilos independientes. Esos métodos son:

- void sleep(long milis): duerme al hilo durante al menos <milis> milisegundos. Transcurrido el tiempo, el hilo pasará a estar preparado para ejecutarse, pero eso no implica que pase inmediatamente a hacerlo (dependerá del planificador), de ahí que pueda estar más tiempo del que se especifica sin ejecutarse.
- void sleep(long milis, int nanos): duerme al hilo durante al menos <milis> milisegundos y <nanos> nanosegundos. Sirve como una implementación con más precisión que la anterior. En la práctica, la implementación actual no permite tanta parecisión, y se limita a redondear los <milis> en función de los <nanos> y a llamar al método sleep anterior con el valor obtenido.
- void yield(): cede el procesador. Pasará a ejecutarse de nuevo el planificador, que decidirá qué otro hilo ejecutar.

Los dos métodos sleep(...) anteriores pueden ocasionar la generación de la excepción InterruptedException. Ésta salta si otro hilo llama al método interrupt() del hilo que está dormido. Cuando eso ocurre, el hilo que estaba dormido pasa inmediatamente a estar preparado, de modo que el planificador volverá a tenerlo en cuenta. Para que el hilo que ejecuta el sleep(...) pueda diferenciar si ha pasado a ejecutarse por culpa de que el tiempo solicitado ha pasado, o porque alguien ha interrumpido su sueño, en este último caso el método retornará con la excepción.

Con el método yield() puede verse qué estrategia sigue el planificador. Para ello se hace que cada hilo escriba su identificador, e inmediatamente ceda el control al planificador.

El ejemplo siguiente hace eso, además de añadir un hilo extra que escribe su identificador y se duerme un tiempo aleatorio, para comprobar también el funcionamiento de sleep().

```
package Ejemplos;
import java.lang.Math;

public class YieldSleep extends Thread {
   int num;
   boolean yield;
   static YieldSleep[] hilos = new YieldSleep[10];
   public YieldSleep(int c, boolean yield) {
      num = c;
      this.yield = yield;
   }
   public void run() {
       while (true) {
         if (yield)
             Thread.currentThread().yield();
         else {
             try {
                  Thread.currentThread().sleep(
                   (long)(Math.random()*1000.0));
               } catch (Exception e) {}
         }
         system.out.println(num);
       }
    }

    public static void main(String[] args) {
   `    // Establecemos el hilo actual como de máxima prioridad, para
        // que no pueda comenzar a ejecutarse ninguna de los hilos
        // hasta que no esten todas creadas.
        Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
        for (int c = 0; c < hilos.length; c++) {
           hilos[c] = new YieldSleep(c, (c != hilos.length - 1));
           hilos[c].setPriority(Thread.NORM_PRIORITY);
           hilos[c].start();
        }
        system.out.println("Todas creadas");
    }

}
```

Este ejemplo solo sirve para ver el modo en el que la máquina virtual realiza la planificación si se ejecuta en una consola. Si se ejecuta desde un applet o una aplicación, existirá un hilo añadido, la del AWT, con mayor prioridad, y que saltará cada vez que algún hilo trate de mostrar su identificador en un cuadro de texto, por ejemplo. El planificador tendrá por lo tanto un hilo más, por lo que el resultado puede ser desconcertante.

Bibliografía.

- Eckel, Bruce. Thinking in Java. Prentice Hall. 1998. Estados Unidos.
- Bertrand Meyer. Construcción de software orientado a objetos .Prentice Hall.