Threads en Java

Introducción

En este artículo voy a explicar cómo se usan los threads en Java (también traducidos como "hilos de ejecución"). La intención no es solamente explicar cuáles son las funciones que hay que llamar si no, también, dar un pantallazo de con qué problemas uno se puede encontrar al crear programas multithread, de qué herramientas se dispone para evitar esos problemas y de cómo utilizarlas.

Cómo se usan

Java tiene un buen soporte de threads. Para usarlo solamente hay que crear un objeto de la clase Thread y pasarle un Runnable. Un Runnable es un objeto que, "implementando" esa interfaz, promete al mundo contar con un método run(). El nuevo thread iniciado comenzará su ejecución saltando a este método y cuando éste termine el thread terminará.

Ejemplo:

```
Thread t = new Thread("Thread para contar", new Runnable() {
	void run()
	{
		for(int i = 1 ; i <= 10 ; i++)
			System.out.println(i);
	}
});
t.start();

/* Acá, para ejemplificar, llamamos a un método que tarda,
 * por ejemplo porque espera que se tipee enter. Mientras
 * tanto, en la pantalla va apareciendo la cuenta hasta diez
 * que sucede en el thread.
 */
in.readLine();
```

¡No confundir el método start con el método run! El método run contiene el código a ser ejecutado “asíncronamente” en otro thread, mientras que el método start es el que crea el thread y en algún punto hace que ese thread ejecute lo que está en run. Este método devuelve el control inmediatamente. Pero si mezclamos todo y ejecutamos directamente run(), el código se ejecutará en el thread actual!

El método start() devuelve el control inmediatamente... mientras tanto, el nuevo thread inicia su recorrido por el método run(). ¿Hasta cuándo? Hasta que termina ese método, cuando sale, termina el thread. Si un thread necesita esperar a que otro termina (por ejemplo el thread padre esperar a que termine el hijo) puede usar el método join(). ¿Por qué se llama así? Bueno, crear un proceso es como una bifuración, se abren dos caminos... que uno espere a otro es lo contrario, una unificación.

Sincronización

La necesidad

La cuestión cuando se trabaja con threads es que la ejecución avanza en varias partes del programa a la vez. Cada una de esas ejecuciones simultáneas pueden tocar los mismos objetos. Eso a veces es un problema. Un ejemplo: Suponga que un thread encola pedidos e incrementa un contador. Existen además 50 threads que se fijan si el contador es mayor que cero y si lo es retiran un pedido, decrementan el contador, y procesan la tarea. Supongamos que hay un pedido en la cola, el contador vale 1, y que sucede lo siguiente:

1. El thread A comprueba que el contador vale más que cero.
2. El thread B comprueba que el contador vale más que cero.
3. Basado en su comprobación el thread B decrementa el contador y toma el pedido.
4. Basado en su comprobación el thread A decrementa el contador y toma el ped... auch, ya no hay pedido!

¿Qué pasó acá? El thread A miró, vio algo y se dispuso a actuar, pero cuando actuó alguien se le había metido en el medio. El mundo ya no era el que era cuando él tomó la decisión de actuar. El problema, generalizado, es el espacio de tiempo que hay entre mirar y actuar, cuando el mundo en el que se mira es compartido por más de un actor. A este tipo de problemas se les llama condición de carrera (en inglés “race condition”), porque son como una competencia.

La solución del problema

Para evitar el caso que expuse lo que se hace es establecer un “lock”, un bloqueo. En Java cada objeto tiene asignado algo que se le llama “monitor”. Mediante la palabra clave synchronized un thread puede “tomar” el monitor. Si otro thread intenta tomar el monitor del mismo objeto, el segundo thread se bloquea hasta que el primero suelte ese monitor.

Entonces, el ejemplo anterior modificado por la sabiduría de los bloqueos quedaría así:

1. El thread A intenta tomar el monitor del objeto de la cola. Lo consigue: es suyo.
2. El thread A comprueba que el contador vale más que cero.
3. El thread B intenta tomar el monitor del objeto de la cola. No lo consigue: lo tiene A, inicia espera.
4. Basado en su comprobación el thread A decrementa el contador y toma el pedido.
5. El thread A libera el monitor.
6. Al quedar el monitor liberado el thread B continúa. Ahora tiene el monitor.
7. El thread B comprueba que el contador vale cero.
8. Basado en su comprobación el thread B ve que no tiene nada que hacer.
9. El thread B libera el monitor.

Sintaxis para establecer locks

Como se dijo, la toma de un monitor se debe hacer con la palabra clave synchronized. Esta palabra tiene dos formas de ser usada. La primera y más básica forma de usar synchronized es la siguiente:

```
synchronized(objeto)
{
	// instrucciones
}
```

El intérprete Java toma el monitor del objeto y ejecuta las instrucciones del bloque. En cualquier caso en el que se salga del bloque el monitor es liberado. Esto incluye la salida por causa de alguna excepción que se produzca. El mecanismo asegura que no exista el riesgo de que se salga de ese bloque sin liberar el monitor, lo que probablemente tarde o temprano haría que se congele todo el programa.

¿Que por qué se llama monitor? Es muy simple, es porqu_ _-_- /-\. '-|-/\-¬ |-\ /-/| --||\|-. '-|-/\-¬ |-\ /-/| \/-¬ --||\|-. Obvio, ¿no?

Es de notar que el uso del monitor de un objeto no tiene nada que ver con el uso del objeto mismo. Mientras el código que deba ser “mutuamente excluido” se sincronice sobre el mismo objeto... ¡no importa qué papel tome ese objeto en el gran esquema de las cosas!

Retomando el ejemplo anterior, una implementación del método que toma un pedido podría ser (antes de aplicar la sabiduría de threads aquí explicada):

```
class MesaDeEntradas
{
	int n = 0;
	Queue<Pedido> cola;

	Pedido retirarPedido()
	{
		if(n > 0)
		{
			n--;
			return cola.remove(0);
		}
		return null;
	}
}
```

Los dos momentos críticos en los que dos threads separados pueden molestarse entre sí son claramente el momento del if y el momento en el que se invoca al método remove(0). Cuando un thread está preguntando el valor de *n* no debe haber ninguno otro ni preguntando el valor de *n*, ni removiendo el valor de la pila. Esto podemos lograrlo tomando el monitor antes de todo este código y liberándolo después:

```
class MesaDeEntradas
{
	int n = 0;
	Queue<Pedido> cola;
```

```
	Pedido retirarPedido()
	{
		synchronized(this)
		{
			if(n > 0)
			{
				n--;
				return cola.remove(0);
			}
			return null;
		}
	}
}
```

Hay dos salidas posibles del bloque sincronizado, ya que hay dos instrucciones return. Pero no hay problema con eso, ya que como se dijo cualquiera sea la forma en que se abandone el bloque, el monitor tomado será liberado correctamente.

Es interesante ver que el hecho de sincronizar sobre this no obedece sino a una convención completamente arbitraria decidida por el programador. Ya que el objeto this es el que está de alguna manera englobando a toda la funcionalidad provista, es lógico usarlo para sincronizar. Pero hay que notar que cualquier otro método de esta clase que use this en un bloque synchronized se sincronizará también con este bloque! Si esto no es lo que se quiere se deberá entonces elegir cualquier otro objeto. A veces tiene sentido crear un objeto solamente para usar su monitor.

Como es muy común querer usar this para sincronizarse, se introdujo una ayudita sintáctica que consiste en una forma abreviada de tomar el monitor de la instancia en la que ese está ejecutando un método. De esta manera...

```
synchronized void m()	{	// ...
}
```

... equivale a ...

```
void m()
{
	synchronized(this)
	{
		// ...
	}
}
```

Por lo tanto podemos reescribir el código anterior como...

```
class MesaDeEntradas
{
	int n = 0;
	Queue<Pedido> cola;
```

```
	synchronized Pedido retirarPedido()
	{
		if(n > 0)
		{
			n--;
			return cola.remove(0);
		}
		return null;
	}
}
```

En el ejemplo que di más arriba teníamos threads que periódicamente se fijaban si un contador era mayor a 0 para saber si había pedidos a procesar. ¿Cada cuánto debería uno fijarse si hay pedidos? ¿A cada minuto? ¿Cada tres? ¿segundos?

En el caso de los ejemplos anteriores, el método retirarPedido devuelve null si no hay todavía ningún pedido esperando (es decir, n == 0). Ahora, supongamos que tenemos 10 threads creados al solo efecto de procesar pedidos. Cada uno de esos threads querrá quedarse esperando si no hay un pedido.

La manera torpe de hacerlo:

```
	Pedido p = null;
	while(true)
	{
		p = m.retirarPedido();
		if(p != null)
			break;
		else
			Thread.sleep(1000); // esperamos un segundo
	}
```

Pero creo que es claro que no es eficiente hacerlo así. Además en muchos casos no conviene esperar todo un segundo para procesar lo que hay que procesar.

Las APIs de sincronización en casi todos los lenguajes y plataformas siempre proveen mecanismos de *espera de condiciones*. En el caso de Java la cosa se hace como explico a continuación. Y acá es donde aparecen wait y notify.

notify:
	Como disparar el tiro que inicia una carrera.
wait:
	El corredor que espera ese tiro para correr.

El thread se “duerme” sobre un monitor. Cuando otro thread “sacude” el monitor, el bello durmiente se despierta y continúa. Esto se hace con las operaciones wait, notify y notifyAll.

Entonces, podría parecer que la cosa se resuelve así: En la "MesaDeEntradas", cuando llega

un pedido, se deberá hacer:

```
class MesaDeEntradas
{
	synchronized void métodoQueRecibeUnPedido(Pedido p)
	{
		cola.add(p);
		notifyAll();
	}
}
```

Y si no somos muy conocedores del tema nos podría parecer que el otro lado del código (el consumidor de la información) se vería algo como esto:

```
	m.wait();
	Pedido p = m.retirarPedido();
```

... pero ¿cómo vamos a esperar sin preguntar antes? ¿Qué pasa si ya había un pedido? Nos quedaríamos esperando en vano, entonces mejor preguntamos:

```
	// si no hay un pedido, esperamos.
	if(!m.hayPedido())
		m.wait();
	Pedido p = retirarPedido();
```

Si algo sacó usted de la lectura de la sección anterior, debería haber descubierto el problema que aparece. Examinamos el mundo (preguntando si hay pedido)... y después actuamos (iniciar la espera). ¿Qué pasa si el pedido llega en el medio y el notify (en otro thread) sucede antes de que nos pongamos a esperar? Nos quedaríamos esperando un notify que nunca llegará.

Por lo dicho es importante tomar "el monitor" del objeto a esperar antes de decidir hacerlo. Pero en el momento inmediatamente después de iniciada la espera el monitor debe ser soltado para que el notify (que intentará tomar ese mismo monitor) pueda suceder. Sería imposible sin ayuda del lenguaje soltar algo mientras estamos en un wait(), ya que la ejecución está suspendida. Por eso, al entrar en el wait se libera atómicamente el monitor. La entrada en el wait, y soltar el "lock" sobre el objeto suceden como una operación indivisible. Repito: Al llamar a wait() sobre un objeto... automáticamente se libera el monitor (¡sin haber salido del synchronized!). Cuando wait termina, automáticamente se recupera el monitor.

Entoces aplicando todo esto correctamente la espera se escribe así:

```
	Pedido p = null;

	synchronized(m)
	{
		m.wait();
		p = retirarPedido();
	}
```