Java y JVM: programación concurrente

Adolfo López Díaz
Escuela de Ciencias de la Computación e Informática. Universidad de Costa Rica

Resumen: El lenguaje de programación Java proporciona, sin necesidad de ninguna otra herramienta adicional, la construcción de programas concurrentes (Gómez, 2000) y es este lenguaje junto a la máquina virtual de Java (JVM) la que permiten la ejecución de estos en un ambiente independiente del sistema operativo (Morin). Este trabajo ofrece una breve reseña sobre las bases de la programación concurrente utilizando estas herramientas.

La máquina virtual de java JVM, Silberchatz (2013) la define como un ambiente de programación virtualizado, ya que en este no se virtualiza el hardware real, sino más bien se crea un sistema virtual optimizado.

Este ambiente de programación es específico para el lenguaje Java y permite que los archivos ejecutables de Java sean multiplataforma, ya que el código será interpretado por la JVM de cada plataforma (Morin). Sabiendo lo que es la JVM empezaremos a ver los principales métodos de concurrencia que existen es esta ambiente de programación y su lenguaje.

En la programación concurrente, existen 2 unidades básicas de ejecución: los procesos y los hilos. (Oracle). Dado que en Java los hilos son el modelo principal para la ejecución de programas (Silberchatz, 2013), son los métodos de concurrencia de los hilos los que estudiaremos.

Algo que no hemos dicho todavía es: ¿qué es un hilo?. Un hilo se define como "una sección de código que se ejecuta independientemente" (UNAM)

Otra pregunta interesante de hacer es ¿cuál es el beneficio de utilizar los hilos? La respuesta la encontramos en la obra de Göetz, en donde dice que los hilos son la forma más fácil de aprovechar al máximo los recursos de los sistemas multiprocesador.

Para la creación de hilos se utiliza la clase *Thread*. Como lo menciona Fernández (2012), hay dos formas de crear hilos en Java:

- Utilizando una clase que extienda de la clase *Thread* y sobreescribiendo su método run()
- Construyendo una clase que implemente la interfaz *Runnable* y luego creando un objeto de la clase *Thread* que recibe como argumento el objeto *Runnable*.

A continuación se muestra un ejemplo de la ejecución de un programa simple de Java multihilo, que usa la interfaz *Runnable*.

```
public class Prueba implements Runnable {
@Override
/*
* Este método es el que va a ejecutar el hilo.
* Cada hilo solamente va a imprimir su nombre.
*/
    public void run(){
        System.out.println("Soy el hilo: " + Thread.currentThread().getName());
    }
}
```

```java
public class Main {
    public static void main(String[] args){
        /* se van a crear 5 hilos */
        for (int i=0; i<5; i++){
            Prueba prueba=new Prueba();
            /* se envía como parámetro el objeto de tipo Runnable */
            Thread thread=new Thread(prueba);
            /* esta instrucción inicia la ejecución del hilo */
            thread.start();
        }
    }
}
```

La salida del programa es la siguiente:

```
Output - Threads (run) ×
run:
Soy el hilo: Thread-0
Soy el hilo: Thread-2
Soy el hilo: Thread-1
Soy el hilo: Thread-3
Soy el hilo: Thread-4
BUILD SUCCESSFUL (total time: 0 seconds)
```

Algo importante a notar es que los hilos no se ejecutan en el orden esperado, pero para esto discutiremos más adelante los métodos de sincronización de hilos.

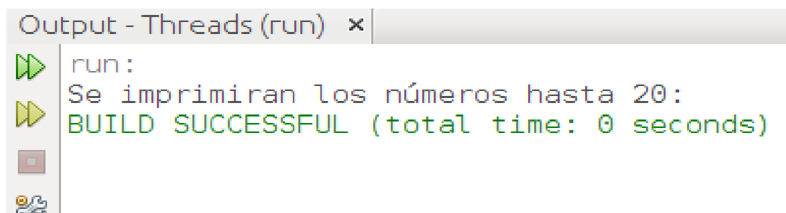
Otro de los métodos importantes en hilos es el join(), este permite a un hilo esperar a que termine la ejecución de otro (Oracle). En el siguiente ejemplo veremos la importancia del join y su uso.

```java
public class Prueba implements Runnable {
    @Override
    public void run(){
        System.out.println("Se imprimiran los números hasta 20:");
        for(int i=1; i<=20; i++){
            System.out.print(i+" ");
        }
    }
}
```

```java
public class Main {
    public static void main(String[] args) throws InterruptedException{
        Prueba prueba = new Prueba();
        Thread thread = new Thread(prueba);
        thread.start();

        System.exit(0);
    }
}
```

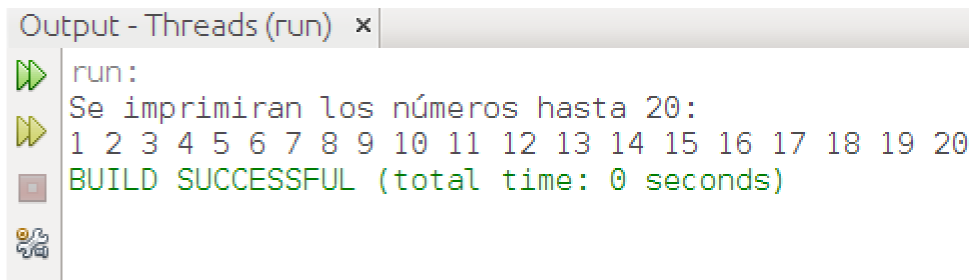
La salida es la siguiente:



Como se puede ver no se imprimieron los números, esto se debe a que el hilo principal (main) no esperó a que el hilo *thread* terminara su ejecución. Para resolver esto utilizaremos el método join() de la forma que se muestra en el código siguiente.

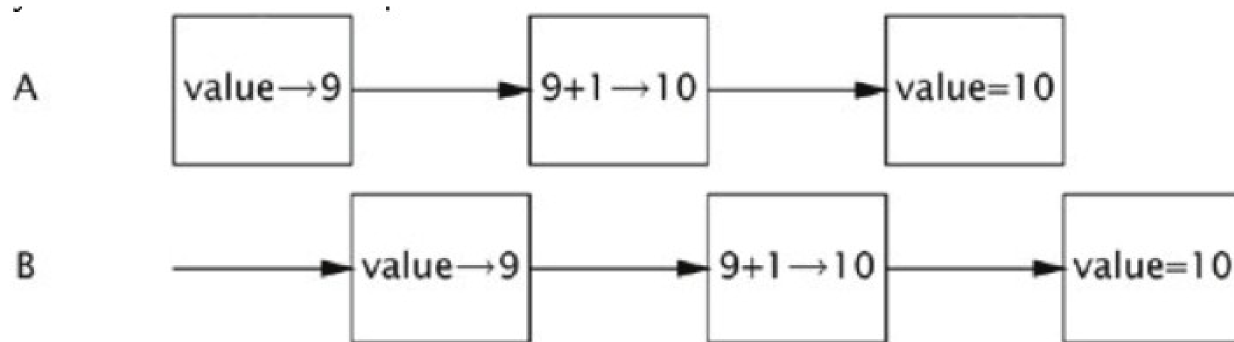
```java
public class Main {
    public static void main(String[] args) throws InterruptedException{
        Prueba prueba = new Prueba();
        Thread thread = new Thread(prueba);
        thread.start();
/* el hilo principal (main) espera a que el hilo thread termine su ejecución */
        try{
            thread.join();
        }catch (InterruptedException e){}
        System.exit(0);
    }
}
```

Ahora la salida del programa es como sigue:



Java provee además para los hilos un atributo de prioridad cuyos valores pueden ser: Thread.MAX_PRIORITY, Thread.MIN_PRIORITY y Thread.NORMAL_PRIORITY.
Para asignar y obtener la prioridad a un hilo se utilizan respectivamente los métodos setPriority(int newPriority) y getPriority() (Fernández, 2012).

Hemos visto como crear hilos y la importancia que tienen para aprovechar el multiprocesamiento pero para una ejecución correcta de un programa multihilo se necesita sincronizar estos, porque sin una sincronización que coordine el acceso a los datos compartidos un hilo podría modificar variables que otro hilo esta modificando y obtener resultados impredecibles (Göetz). Un ejemplo de esto se ve en la siguiente figura tomada de la obra de Göetz.



Para resolver estos problemas Java provee un mecanismo de bloqueo para asegurar la atomicidad en la ejecución de las instrucciones el cual es llamado bloque sincronizado (Göetz).
Un ejemplo de uso de los bloques sincronizados sería de la forma:

```
public synchronized void incrContador() {
        contador = contador + 1;
}
```

Lo anterior permite que un método completo se ejecute atómicamente. También se puede bloquear solo un bloque de instrucciones de la siguiente forma como ha sido indicado por Oracle.

```
public void addName(String name) {
   synchronized(this) {
      lastName = name;
      nameCount++;
   }
   nameList.add(name);
}
```

Lo anterior funciona de la siguiente manera. En java cada objeto tiene asociado un cerrojo y en el momento en que se ejecuta un método o un bloque de instrucciones sincronizado se adquiere el cerrojo para ese objeto, por lo tanto solo un hilo podrá ejecutar un método sincronizado a la vez. (Gómez, 2000)

Horvilleur explica el funcionamiento de los bloques sincronizados de la siguiente forma "El modificador synchronized indica que se debe adquirir el lock al entrar al método y liberarlo al salir del método", por lo tanto el programador no debe preocuparse por tener algún problema por no liberar el cerrojo, ya que "los cerrojos los utiliza la propia máquina virtual, y el compilador, pero el lenguaje impide el acceso a ellos por parte del programador" (Horvilleur).

Vamos a ver la ejecución de un programa de prueba en Java utilizando bloques sincronizados y sin el uso de ellos.

Si no existe sincronización la ejecución queda como sigue:

```
Se imprimiran los números hasta 10:
1 2 3 4 5 6 7 8 9
Se imprimiran los números hasta 10:

Se imprimiran los números hasta 10:
1 2 3 4 5
Se imprimiran los números hasta 10:
1 2 3 4 5 6 7 8
Se imprimiran los números hasta 10:
9
Se imprimiran los números hasta 10:
6 1 10 7 2 3 4 1 10 1 2 3
Se imprimiran los números hasta 10:

Se imprimiran los números hasta 10:
2 5 8 3 6 1 1 4 2 2 3 4 5 6 7 8 9 10 7 4 9 5 6 7 8
Se imprimiran los números hasta 10:
3 5 4 1 9 8 10 9 10 10 2
Se imprimiran los números hasta 10:
5 6 6 1 3 2 7 7 8 3 4 4 9 8 10 5 6 7 8 9 10 5 9 6 10 7 8 9 10 BUILD SUCCESSFUL (total time: 0 seconds)
```

Como se puede ver no existe un orden adecuado en la ejecución ya que algunos hilos están traslapando su ejecución.

Ahora veremos la ejecución con bloques sincronizados:

```
Se imprimiran los números hasta 10:
1 2 3 4 5 6 7 8 9 10
Se imprimiran los números hasta 10:
1 2 3 4 5 6 7 8 9 10
Se imprimiran los números hasta 10:
1 2 3 4 5 6 7 8 9 10
Se imprimiran los números hasta 10:
1 2 3 4 5 6 7 8 9 10
Se imprimiran los números hasta 10:
1 2 3 4 5 6 7 8 9 10
Se imprimiran los números hasta 10:
1 2 3 4 5 6 7 8 9 10
Se imprimiran los números hasta 10:
1 2 3 4 5 6 7 8 9 10
Se imprimiran los números hasta 10:
1 2 3 4 5 6 7 8 9 10
Se imprimiran los números hasta 10:
1 2 3 4 5 6 7 8 9 10
Se imprimiran los números hasta 10:
1 2 3 4 5 6 7 8 9 10 BUILD SUCCESSFUL (total time: 0 seconds)
```

Como vemos la ejecución fue exitosa y en el orden adecuado. El código utilizado es el siguiente:

```
public class Prueba implements Runnable {
    @Override
    public void run(){
        synchronized(this){
        System.out.println("\nSe imprimiran los números hasta 10:");
        for(int i=1; i<=10; i++){
            System.out.print(i+" ");
        }
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) throws InterruptedException{
        Prueba prueba = new Prueba();
        for(int i = 0; i<10; i++){
        Thread thread = new Thread(prueba);
        thread.start();
        }

    }
}
```

A pesar de que la sincronización nos permite evitar ciertos errores, al introducir la sincronización puede ocurrir que uno o más hilos se ejecuten lentamente o no se lleguen a ejecutar (Oracle). Entre estos problemas está el problema de *starvation* y de los interbloqueos o *deadlocks*.

En Oracle se menciona que el problema de *starvation* se da cuando un hilo no puede obtener acceso a recursos compartidos y no puede ejecutarse y el deadlock se da cuando varios procesos esperan por recursos que tienen bloqueados mutuamente. Un ejemplo de un deadlock es el siguiente, tomado de la publicación de Horvilleur.

```
public class Demo6 implements Runnable {
	private Object obj1;
	private Object obj2;
	public Demo6(Object o1, Object o2) {
		obj1 = o1;
		obj2 = o2;
	}
	public void run() {
		for (int i = 0; i < 1000; i++) {
			synchronized (obj1) {
				synchronized (obj2) {
					System.out.println(i);}
			}
		}
	}

	public static void main(String[] args) {
		Object o1 = new Object();
		Object o2 = new Object();
		new Thread(new Demo6(o1, o2)).start();
		new Thread(new Demo6(o2, o1)).start();
	}

}
```

Un problema relacionado con la situación anterior es que “Java no dispone, no obstante, de control sobre las situaciones peligrosas en los programas concurrentes que puedan ocasionar malos funcionamientos, como por ejemplo los interbloqueos. Es responsabilidad del diseñador del sistema dar una correcta estructura al programa que tenga que implementar, de modo que no aparezcan situaciones indeseadas” (Gómez, 2000)

Otros recursos para la sincronizacion ademas de los bloques sincronizados, que se encuentra específicamente en el paquete java.util.concurrent, son los semáforos contadores que permiten a un recurso ser compartido por varios procesos. (Jenkov)

Los principales métodos de los semáforos son acquire() y release(), que permiten adquirir o liberar, respectivamente, un bloqueo sobre un objeto. (Oracle)

Mostraremos el mismo ejemplo que se hizo con los bloques sincronizados ahora con un semáforo.

```java
import java.util.concurrent.Semaphore;

public class Prueba implements Runnable {
    Semaphore semaphore = new Semaphore(1);
    @Override
    public void run(){
        semaphore.acquire();
        System.out.println("\nSe imprimiran los números hasta 10:");
        for(int i=1; i<=10; i++){
            System.out.print(i+" ");
        }
        semaphore.release();
    }
}
```

```java
public class Main {
    public static void main(String[] args) throws InterruptedException{
        Prueba prueba = new Prueba();
        for(int i = 0; i<10; i++){
        Thread thread = new Thread(prueba);
        thread.start();
        }

    }
}
```

Ahora vemos la salida que produce el código anterior.

```
Se imprimiran los números hasta 10:
1 2 3 4 5 6 7 8 9 10
Se imprimiran los números hasta 10:
1 2 3 4 5 6 7 8 9 10
Se imprimiran los números hasta 10:
1 2 3 4 5 6 7 8 9 10
Se imprimiran los números hasta 10:
1 2 3 4 5 6 7 8 9 10
Se imprimiran los números hasta 10:
1 2 3 4 5 6 7 8 9 10
Se imprimiran los números hasta 10:
1 2 3 4 5 6 7 8 9 10
Se imprimiran los números hasta 10:
1 2 3 4 5 6 7 8 9 10
Se imprimiran los números hasta 10:
1 2 3 4 5 6 7 8 9 10
Se imprimiran los números hasta 10:
1 2 3 4 5 6 7 8 9 10
Se imprimiran los números hasta 10:
1 2 3 4 5 6 7 8 9 10 BUILD SUCCESSFUL (total time: 0 seconds)
```

Como en el ejemplo anterior de los bloques sincronizados hemos podido ver que la ejecución ha sido exitosa gracias al buen uso de la sincronización.

CONCLUSIONES

Se ha reseñado muy básicamente los principios de la programación concurrente en Java y hemos visto que tanto la API de este lenguaje como la máquina virtual JVM brindan soporte a este tipo de programación. Todo este soporte se brinda principalmente mediante la clase *Thread*, dando facilidades para crear y administrar hilos. También se mostró como los bloques sincronizados y la clase *Semaphore* permiten una sincronización efectiva de los hilos para evitar algunos de los más importantes problemas que se dan en la programación concurrente.

REFERENCIAS

Fernández, J. (2012). *Java 7 Concurrency Cookbook*. Birmingham, Reino Unido: Packt Publishing.

Göetz,B., Peierls T., Bloch, J.(2006). *Java Concurrency In Practice*. EE.UU: Addison-Wesley

Gómez, P. (2000). *Concurrencia en Java*. Recuperado el 4 de noviembre de 2013, de rt00149b.eresmas.net/Otras/ConcurrenciaJAVA/ConcurrenciaEnJava.PDF

Horvilleur, G. (s.f). *Multithreading y Java.* Recuperado el 4 de noviembre de 2013, de http://www.comunidadjava.org/files/CuartaReunion/JavaYMultithreading.pdf

Jenkov, J. (s.f.). *java.util.concurrent.Semaphore*. Recuperado el 4 de noviembre de 2013, de http://tutorials.jenkov.com/java-util-concurrent/semaphore.html.

Morin, P. (s.f.). *The Java Virtual Machine (JVM)*. Recuperado el 4 de noviembre de 2013, del sitio Web de Carleton University: http://cg.scs.carleton.ca/~morin/teaching/3002/notes/jvm.pdf .

Oracle. (s.f.).*The Java Tutorials: Lesson Concurrency.* Recuperado el 4 de noviembre de 2013, de http://docs.oracle.com/javase/tutorial/essential/concurrency/ .

Silberchatz, A., Baer, P y Gagne, G. (2013). *Operating Systems Concepts* (9a ed). EE.UU: John Wiley & Sons.

UNAM, Posgrado en Ciencia e Ingeniería de la Computación. (s.f). *Concurrencia en Java.* Recuperado el 4 de noviembre de 2013, del sitio Web de la Universidad Nacional Autónoma de México: http://www.matematicas.unam.mx/jloa/concurrencia.pdf