Sincronización de hilos - otro

En la anterior entrada veíamos qué era un hilo, su ciclo de vida y cómo crear uno. En esta entrada estudiaremos algunos de los problemas que existen cuando hay varios hilos en funcionamiento y cómo se solucionan.

Herramientas de planificación de hilos

Java nos proporciona un conjunto de métodos que permiten controlar – en mayor o menor medida – el cambio de un hilo de un estado a otro. A continuación explicaremos uno a uno cada uno de estos métodos.

Método yield()

Un hilo puede ceder el tiempo de CPU que se le ha asignado para que otros hilos puedan ejecutarse. Con la llamada a este método, el hilo pasará del estado en ejecución al estado preparado.

En el caso que no haya otro hilo esperando CPU, el planificador de hilos volverá a cambiar el estado del hilo de preparado a en ejecución, para que vuelva a ocupar tiempo de CPU.

Método sleep(parametroTiempo)

Con la llamada a este método, el hilo pasa al estado bloqueado tantos milisegundos como se le indiquen en el parámetro de entrada parametroTiempo.

Una vez cumplido esa cantidad de tiempo, se pone en estado preparado.

Método join()

Este método permite a un hilo quedar a la espera que termine un segundo hilo.

El método join() suele utilizarse para mantener un orden en la secuencia de los hilos. Así, podemos arrancar una secuencia de hilos llamando a join() para que cada uno finalice en el orden que hemos marcado según la llamada a join().ATENCIÓN

Es obligatorio controlar los métodos join() y sleep() mediante la excepción InterruptedException, para evitar errores de compilación. Por ejemplo, debéis hacer algo como esto:

```
_____

public class UnHilo extends Thread {

public UnHilo(String nombreHilo) {

super(nombreHilo);

}

public void run() {

System.out.println(getName());

}

}
```

_____

Código de UnHilo.java

_____

```
public class TestUnHilo () {

public static void main (String[] args) {

UnHilo hiloUno = new UnHilo(«HiloUno»);

UnHilo hiloDos = new UnHilo(«HiloDos»);

hiloUno.start();

hiloDos.start();

try {

hiloUno.join();

hiloDos.join();

} catch (InterruptedException ie) {

}

System.out.println(«El programa ha finalizado»);
```

}

}

Código de TestUnHilo.java con gestión de excepciones

Sección crítica

El problema de la sección crítica es muy conocido en el ámbito del multiprocesamiento. De hecho, posiblemente ya os habréis enfrentado a algo parecido con el estudio de la concurrencia de datos en el módulo de SGBD (qué ocurre cuando se accede a un mismo dato desde varias sesiones simultáneamente).

De todas maneras, abordaremos el problema para aquellos que no lo hayáis dado o, simplemente, se os haya olvidado.

Observad el siguiente código de programa:

```
public class Contador {

private int contador = 1;

public void setContador(int nContador) {

contador = nContador;

}

public int getContador() {

return contador;

}

}
```

```
/*_____________________

Código de Contador.java

_____________________*/

public class HiloContador extends Thread {

private Contador contador;

public HiloContador(String nNombre, Contador nContador) {

super(nNombre);

contador=nContador;

}

public void run() {

try {

for (int j=0; j<10; j++) {

int i=contador.getContador();

sleep((int) (Math.random()*10));

contador.setContador(i+1);

System.out.println(getName() + » pone el contador a » + i);

}

} catch (InterruptedException e) {

System.out.println(«Error al ejecutar el método sleep»);

}

}

}
```

/*_____

Código de HiloContador.java

_____*/

```
public class TestHiloContador {

public static void main(String[] args) {

Contador cont = new Contador();

HiloContador hc1 = new HiloContador(«HiloUno», cont);

HiloContador hc2 = new HiloContador(«HiloDos»,cont);

hc1.start();

hc2.start();

try{

hc1.join();

hc2.join();

} catch (InterruptedException e) {

System.out.println(«Error al ejecutar el método join»);

}

System.out.println(«El último valor que debería mostrarse es 10*2=20»);

}

}
```

_____

Código de TestHiloContador.java

Tenemos tres clases: Contador, HiloContador y TestHiloContador.

La clase Contador es la que contendrá la sección crítica, en nuestro caso un sencillo contador. Hemos introducido en la clase un atributo de tipo entero inicializado a 1 y sus correspondientes setter y getter.

La clase HiloContador es el hilo que ejecutará la sección crítica. Fijaos que simplemente se trata de la ejecución de un bucle en 10 iteraciones en el que se incrementa en 1 el valor de la variable contador. Así, finalmente, cada hilo habrá aumentado en 10 unidades el valor inicial que tenia el contador.

IMPORTANTE: Hemos introducido un tiempo de espera con la instrucción sleep() entre la recogida del valor de contador (getContador()) y su asignación con su incremento en 1 (setContador()) para que la CPU no realice de una sola vez todo un hilo y después otro. Así podremos comprobar el efecto deseado.

La clase TestHiloContador es la que contiene el método main() que creará un objeto Contador y dos hilos HiloContador, que nos permitirá ver el funcionamiento del programa con dos hilos ejecutándose simultáneamente.

Copiad el código y comprobad el resultado. Podréis apreciar que el resultado no es el esperado y que varía cada vez que ejecutamos el programa. Esto se debe a que existe una región crítica (la variable contador) que no controlamos.

Una solución a este problema es utilizar **synchronized(nombreDeVariableSeccionCritica)** y recoger todo el código de la sección crítica dentro de un bloque. Para entenderlo mejor, fijaos en la solución al problema anterior. Simplemente modificaremos el código de la sección crítica, en nuestro caso, el método run() del hilo. Veámoslo:

```
public void run() {

try {

synchronized (contador) {

for (int j=0; j<10; j++) {

int i=contador.getContador();

sleep((int) (Math.random()*10));

contador.setContador(i+1);

System.out.println(getName() + »  pone el contador a » + i);

}

}

} catch (InterruptedException e) {

System.out.println(«Error al ejecutar el método sleep»);
```

```
}

}
```

Con esta modificación tan sencilla, el resultado siempre será el deseado. La explicación es que ahora el planificador de hilos de Java se encarga de sincronizar que, en cada momento, sólo un hilo pueda acceder al valor de la variable dentro del bloque enmarcado por la palabra reservada synchronized.