Problemas y Soluciones de la Programación concurrente (II)

Sincronización Básica: *volatile*

Cuando tenemos múltiples núcleos, los procesadores disponen de técnicas de optimización y algunas de ellas se basan en el uso de cachés. No obstante, en el caso de la programación concurrente, pueden generarse errores si se guarda el valor de una variable en la caché de dos núcleos diferentes. Para evitarlo, las variables compartidas por dos o más hilos se declaran en Java como **volatile**, de forma que se garantiza que exista una única copia de ellas independientemente del número de núcleos del procesador.

```
private volatile static long comparatidaConHilos = 0;
```

Esta solución no resuelve todos los problemas de inconsistencias de memoria, pero sí son apropiadas en los sistemas de un único hilo escritor y varios hilos lectores.

Sincronización Básica: *wait, notify, notifyAll*

Todas las clases Java disponen de los métodos **wait, notify** y **notifyAll** dado que se encuentran en Object. Todos estos métodos se utilizan cuando estamos trabajando con segmentos sincronizados, y obligan a capturar una excepción del tipo **InteruptedException**.

Wait detiene la ejecución del hilo. Por su parte, **notify** y **notifyAll** producen la reactivación de los hilos detenidos. Notify hace continuar a un único segmento al azar de los que están detenidos con wait, mientras que notifyAll hace continuar a todos ellos.

```
Metodo 1 - Ejecucion 0
Metodo 1 - Ejecucion 1
Metodo 1 - Ejecucion 2
Metodo 1 - Ejecucion 3
Metodo 1 - Ejecucion 4
Metodo 1 - Ejecucion 5
Metodo 2 - Ejecucion 1
Metodo 2 - Ejecucion 2
Metodo 2 - Ejecucion 3
Metodo 2 - Ejecucion 4
Metodo 2 - Ejecucion 5
Metodo 2 - Ejecucion 6
Metodo 2 - Ejecucion 7
Metodo 2 - Ejecucion 8
Metodo 2 - Ejecucion 9
Metodo 1 - Ejecucion 6
Metodo 1 - Ejecucion 7
Metodo 1 - Ejecucion 8
Metodo 1 - Ejecucion 9
```

En el siguiente ejemplo (**WaitNotifySimple**) generamos dos hilos que tienen una progresión aritmética. Ambos arrancan a la vez – aunque no lo parezca – pero al llegar a i = 5, el primer Hilo se detiene con un **wait**. No es hasta que termina el segundo Hilo que éste puede reiniciar su ejecución.

Éste es un ejemplo básico de cómo podemos desde el código fuente gestionar problemas de sincronización entre tareas.

```java
public class WaitNotifySimple implements Runnable {
    private volatile boolean ejecutandoMetodo1 = false;

    public synchronized void metodo1() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Metodo 1 - Ejecucion " + i);
            if (i == 5) {
                try {
                    this.wait(); // Paramos el Hilo
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    public synchronized void metodo2() {
        for (int i = 1; i < 10; i++) {
            System.out.println("Metodo 2 - Ejecucion " + i);
        }
        this.notifyAll(); // Reanuda la ejecucion del Hilo detenido
    }

    @Override
    public void run() {
        if (!ejecutandoMetodo1) {
            ejecutandoMetodo1 = true;
            metodo1();
        } else
            metodo2();
    }

    public static void main(String[] args) {
        WaitNotifySimple waitNotifySimple = new WaitNotifySimple();
        new Thread(waitNotifySimple).start();
        new Thread(waitNotifySimple).start();
    }
```

Sincronización Básica: *join*

El método join permite indicar a un hilo que debe suspender su ejecución hasta que otro hilo de referencia finalice. Este método debe de ejecutarse dentro del bloque asíncrono del código, de lo contrario no tendría ningún efecto.

Por ejemplo, en **JoinBasico** tenemos un Hilo2 que está obligado a suspender su ejecución hasta que termine la de Hilo1. Para ello, será necesario pasarle la referencia de Hilo1.

```java
public class JoinBasico extends Thread {

    private int id;
    private boolean suspender = false;
    private Thread hiloReferencia;

    public JoinBasico (int id) {
        this.id = id;
    }

    public void suspenderHilo(Thread hiloReferencia) {
        this.suspender = true;
        this.hiloReferencia = hiloReferencia;
    }

    @Override
    public void run() {
        try {
            for (int i = 0; i < 3; i++) {
                if (suspender) {
                    hiloReferencia.join();
                }
                System.out.println ("Hilo " + id + ", Iteracion " + i);
                Thread.sleep(1000);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        JoinBasico hilo1 = new JoinBasico(1);
        JoinBasico hilo2 = new JoinBasico(2);
        hilo1.start();
        hilo2.start();
        hilo2.suspenderHilo(hilo1);
    }
}
```

Sincronización Avanzada: ***exclusión mutua, synchronized y monitores***

Java nos ofrece un mecanismo para sincronizar segmentos de código mediante la palabra reservada **synchronized**. Un bloque o método marcado como **synchronized** limita el acceso a un único hilo concurrente, de forma que se garantiza la Exclusión Mutua. Esto se cumple siempre para esa instancia de objeto.

Un método **synchronized**:

```java
public synchronized void metodo1() {
```

Adicionalmente, podemos declarar bloques independientes sincronizados, a los que se les conoce como **monitor**. Dicho monitor hace las veces de candado, de forma que al ejecutarse el código gestionado por un monitor éste no puede volver a ser ejecutado hasta que no se libere.

Por ejemplo, el programa **SincronizacionMetodos** proporciona una sincronización simple.

```
public class SincronizacionMetodos implements Runnable {

    public synchronized void metodo1() {
        System.out.println("Metodo 1 - Inicio");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            return;
        }
        System.out.println("Metodo 1 - Fin");
    }

    public synchronized void metodo2() {
        System.out.println("Metodo 2 - Inicio");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            return;
        }
        System.out.println("Metodo 2 - Fin");
    }

    @Override
    public void run() {
        metodo1();
        metodo2();
    }

    public static void main(String[] args) {
        SincronizacionMetodos sincronizacionMetodos = new SincronizacionMetodos();
        new Thread (sincronizacionMetodos).start();
        new Thread (sincronizacionMetodos).start();
    }
}
```

Los métodos se ejecutan uno detrás del otro pese a que se dispone de dos Hilos diferentes. Esto sucede porque es el mismo objeto. Si lanzamos **SinSincronizacionMetodos** veremos que esto no sucede.

Puedes comprobar en **SincronizacionSegmento** cómo se realizaría el mismo programa utilizando **monitores**. Es una solución que, personalmente, no me gusta. Es difícil de gestionar dado que puedes marcar los bloques **synchronized** en cualquier sitio, lo que complica entender lo que estás haciendo.

Sincronización Avanzada: *semáforos*

El uso de **synchronized** está muy bien para la mayoría de los casos, pero a veces ocurre que un recurso puede ser accedido por más de un Hilo a la vez. En estos casos, se hace uso de la clase **Semaphore**. Al construir un semáforo, se le indica el número de Hilos que pueden acceder al recurso que controla de forma concurrente. Cuando un Hilo desea acceder a ese recurso, se le entrega un 'pase' (**aquire** o **tryAcquire**), suponiendo claro que haya alguno disponible. Si no hay, el Hilo quedará bloqueado hasta que se libere uno (**release**).

En el ejemplo **SemaforoBasico** podemos ver un ejemplo de un semáforo que limita a tres Hilos el acceso a una sección crítica.

```
public class SemaforoBasico implements Runnable {

    Semaphore semaphore = new Semaphore(3);

    @Override
    public void run() {
        try {

            semaphore.acquire(); // ¿Puedo pasar?
            System.out.println("Paso1");
            Thread.sleep(1000);
            System.out.println("Paso2");
            Thread.sleep(1000);
            System.out.println("Paso3");
            Thread.sleep(1000);
            semaphore.release(); // He terminado

        } catch (InterruptedException e) {

        }
    }

    public static void main(String[] args) {
        SemaforoBasico semaforoBasico = new SemaforoBasico();
        for (int i = 0; i < 5; i++)
            new Thread(semaforoBasico).start();
    }
}
```