

Repaso parte 2 - Sincronización de hilos

Cuando dos o más hilos necesitan acceso a un recurso compartido (como una variable, un registro de una base de datos) necesitamos coordinar su ejecución. A esto se le llama sincronización.

Por ejemplo, cuando un hilo está escribiendo en un archivo, se debe evitar que un segundo hilo lo haga al mismo tiempo y pueda borrar los cambios del primero.

Otra razón para la sincronización es cuando un hilo está esperando un evento causado por otro hilo. En este caso, debemos mantener el primer hilo en suspensión hasta que ocurra el evento y, entonces, reanudar su ejecución.

Monitores

La sincronización a mas bajo nivel en Java se le conoce como monitor, y controla el acceso a un objeto.

Un monitor funciona implementando el concepto de bloqueo. Cuando un objeto está bloqueado por un hilo, ningún otro hilo puede obtener acceso a él. Cuando el hilo termina desbloquea el objeto y este queda disponible para ser utilizado por otros hilos.

En Java, la clase Object es la raíz de todas las herencias, y cualquier clase implementa por defecto los metodos de esta clase. Por lo tanto, todas las clases tienen un monitor, y se pueden sincronizar. La sincronización se realiza con la palabra **synchronized**. Hay dos formas de utilizarla:

Forma 1: Usando métodos synchronized

Añadiendo la palabra synchronized al metodo.

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

Cuando hagamos una instancia de SynchronizedCounter y accedamos a sus metodos **synchronized** generará dos efectos:

Primero, mientras otro hilo este utilizando un metodo synchronized, ningún otro hilo puede ingresar a ese método o cualquier otro método sincronizado de la clase.

Segundo, se garantiza que los cambios que podamos realizar sobre el objeto serán visibles por todos los hilos.

Forma 2: La declaración synchronized

La forma 1 es un medio fácil y efectivo para lograr la sincronización, aunque no funcionará en todos los casos.

Por ejemplo, es posible que se desee sincronizar el acceso a algún método que no tenga la palabra synchronized.

Esto puede ocurrir porque la clase haya sido creada por un tercero, y no tengamos acceso al código fuente. Por lo tanto, no es posible agregar synchronized a los métodos apropiados dentro de la clase.

¿Cómo se puede sincronizar el acceso a un objeto de esta clase? Afortunadamente, la solución a este problema es bastante sencilla: simplemente realiza llamadas a los métodos definidos por esta clase dentro de un bloque synchronized.

Esta es la forma general de un bloque synchronized:

```
synchronized(objref) {  
    // declaraciones a sincronizar  
}
```

Aquí, objref es una referencia al objeto que se sincroniza. Una vez que se ha ingresado un bloque sincronizado, ningún otro hilo puede llamar a un método sincronizado en el objeto referido por objref hasta que se haya salido del bloque.

Siguiendo el ejemplo de la forma 1 podríamos reescribirlo de la siguiente forma:

```
public class SynchronizedCounter {  
    private int c = 0;  
  
    public void increment() {  
        synchronized(this) {  
            c++;  
        }  
    }  
    //... lo mismo con los demás métodos  
}
```

this hace referencia a la propia instancia del objeto. A nivel de ejecución las dos formas serian equivalentes en este caso.

Sin embargo, podemos adaptar esta forma para sacarle mas jugo:

```
public class MsLunch {  
    private long c1 = 0;  
    private long c2 = 0;  
    private Object lock1 = new Object();  
    private Object lock2 = new Object();  
  
    public void inc1() {  
        synchronized(lock1) {  
            c1++;  
        }  
    }  
  
    public void inc2() {  
        synchronized(lock2) {  
            c2++;  
        }  
    }  
}
```

Suponiendo que en la clase MsLunch tenemos dos numeros c1 y c2, y que nunca se utilizan conjuntamente. Todos los incrementos deben de estar sincronizados. Pero no hay razon para que que c1 no sea incrementado mientras c2 esta siendo incrementado. De esta forma reducimos bloqueo de hilos innecesarios.

Comunicación entre hilos

Considera la siguiente situación, un hilo está ejecutando un método synchronized (bloquea al resto de hilos), y dentro de este metodo necesita acceso a un segundo objeto que no está disponible temporalmente. Mientras el hilo está a la espera del segundo objeto, bloquea el acceso de otros hilos. Es una ejecución poco optima porque no aprovecha las ventajas de la concurrencia y alarga el tiempo de ejecución sin hacer nada mas que esperar.

¿Qué debería hacer el hilo entonces?

Una mejor solución es hacer que el hilo renuncie temporalmente al control del primer objeto, permitiendo que se ejecute otro hilo. Cuando el segundo recurso esté disponible, se puede notificar al hilo y reanudar su ejecución.

Tal enfoque se basa en alguna forma de comunicación entre hilos en la que un hilo puede notificar a otro que está bloqueado y recibir una notificación de que puede reanudar la ejecución. Java admite la comunicación entre hilos con los métodos **wait()**, **notify()** y **notifyAll()**.

wait(), notify() y notifyAll()

Los métodos `wait()`, `notify()` y `notifyAll()` son parte de todos los objetos porque están implementados por la clase `Object`. Estos métodos sólo deben invocarse desde un contexto sincronizado.

Dentro de un bloque `synchronized`, cuando un hilo no puede continuar su ejecución se puede pausar temporalmente llamando a `wait()`. Esto ocasiona que el hilo quede en reposo y que se libere el monitor para ese objeto, permitiendo que otro hilo use el objeto.

3 / 11
unidad3.md 3/22/2020

En un momento posterior, el hilo en reposo se activa cuando otro hilo entra al mismo monitor y llama a `notify()` o `notifyAll()`.

Una llamada a `notify()` reanuda un hilo de espera (el que elija el planificador de tareas). Una llamada a `notifyAll()` notifica a todos los hilos, solo un hilo ganara acceso al objeto.

Antes de mirar un ejemplo que usa `wait()`, se necesita hacer un punto importante. Aunque `wait()` normalmente espera hasta que se llame a `notify()` o `notifyAll()`, existe la posibilidad de que, en casos muy raros, el hilo de espera se pueda activar debido a una falsa alarma.

Las condiciones que conducen a una activación falsa son complejas. Sin embargo, Oracle recomienda que, debido a la remota posibilidad de una activación falsa, las llamadas a `wait()` se realicen dentro de un bucle que verifique la condición en la que el hilo está esperando. El siguiente ejemplo muestra esta técnica.

Ejemplo Productor-Consumidor

Se trata de un ejemplo clasico en concurrencia. Tenemos un escritor (productor) que va a enviar una frase, carácter a carácter, a un lector(consumidor). Utilizaran un objeto, buffer, para enviarse cada carácter. La clave de este ejemplo es que el escritor no podrá sobrecribir el carácter hasta asegurarse que el consumidor lo ha leído antes.

Esta es la clase `Buffer` que tienen en común tanto el lector como el escritor. Si os fijais en la lógica de la clase:

Antes de escribir nos aseguramos mediante un booleano que el lector lo ha leído. Sino es así el hilo escritor se esperara hasta ser notificado por el lector.

Antes de leer el lector tiene que asegurarse que hay un dato en el contenedor, o que no es el mismo dato que ya ha leído antes. Sino hay nada nuevo que leer se esperara hasta ser notificado por el escritor.

```
public class Buffer {
    private char contenido;
    private boolean contenedorlleno = false;

    public synchronized char get() throws InterruptedException{
        while (!contenedorlleno){
            wait();
        }
        contenedorlleno = false;
        notify();
        return contenido;
    }
}
```

```
}
```

```
public synchronized void put(char value) throws InterruptedException{  
while (contenedorlleno){  
wait();  
}  
contenido = value;  
contenedorlleno = true;  
notify();  
}
```

4 / 11
unidad3.md 3/22/2020

```
}
```

```
}
```

La clase lector simplemente leera el contenido del buffer y lo mostrara por pantalla.

```
public class Lector extends Thread {  
private Buffer buffer;  
  
public Lector(Buffer buffer) {  
this.buffer = buffer;  
}  
  
@Override  
public void run() {  
try{  
for (int i=0; i<14; i++){  
System.out.println(buffer.get());  
}  
} catch (InterruptedException e) { }  
}  
}
```

La clase escritor simplemente va a escribir caracter a caracter la frase.

```
public class Escritor extends Thread {  
private Buffer buffer;  
  
public Escritor(Buffer buffer) {  
this.buffer = buffer;  
}  
  
@Override  
public void run() {  
String frase = "Frase a enviar";  
try{  
for (int i=0; i<frase.length(); i++){  
buffer.put(frase.charAt(i));  
}  
} catch (InterruptedException e) { }  
}
```

```
}  
}
```

La clase que ejecuta todos los componentes.

```
public class Main {  
    public static void main(String[] args) {  
        Buffer buffer = new Buffer();
```

5 / 11
unidad3.md 3/22/2020

```
        new Escritor(buffer).start();  
        new Lector(buffer).start();  
    }  
}
```

Problemas en las ejecución

La ejecución concurrente de hilos favorece el mejor aprovechamiento de la CPU, y permite la multitarea. Sin embargo, requiere de una atención especial a los posibles problemas que puede generar

Problema 1: Race conditions

Se produce cuando dos o más hilos quieren acceder a un mismo recurso compartido con la intención de modificar su valor.

Dado que el algoritmo para dar paso a un hilo u otro es impredecible no sabemos en qué orden accederán los hilos al recurso, y puede provocar incongruencias o lecturas sucias.

Ejemplo sin sincronización:

```
public class Monedero {  
  
    int cantidad;  
  
    public void ingresar (int dinero) throws InterruptedException {  
        this.cantidad += dinero;  
    }  
}
```

```
public class Persona extends Thread{  
  
    private Monedero monedero;  
  
    public Persona(Monedero monedero) {  
        this.monedero = monedero;  
    }
```

```

@Override
public void run() {
    for(int i=0; i<100; i++){ // Cada persona ingresa 1€ 100 veces    try {
        monedero.ingresar(1);
    } catch (InterruptedException e) { }
    }
}
}
}

```

6 / 11
 unidad3.md 3/22/2020

```

public class Main {
    public static void main(String[] args) {
        Monedero m = new Monedero(0);
        for (int i=0; i<100; i++){ //Creo 100 personas
            new Persona(m).start();
        }
        try {
            Thread.sleep(2000); // Lo mejor seria hacer un join()
            System.out.println("Dinero final: "+m.cantidad);
        } catch (InterruptedException e) { }
    }
}

```

Si haceis las cuentas la ejecución debería arrojar un resultado de: 1000€. Sin embargo, **el resultado son unos cuantos euros menos.**

La solución a este problema es utilizar **synchronized** que garantiza que mientras otro hilo está modificando el recurso no habrá un segundo hilo que lo modifique. Esta medida aporta seguridad aunque afecta de forma notable al rendimiento de nuestra aplicación y al concepto de la programación concurrente.

En este caso el recurso sería Monedero, y aplicaríamos la palabra reservada synchronized al metodo ingresar(). De esta forma, mientras una persona ingresa dinero las demas deberan esperar hasta que esta hubiera terminado de ingresar.

```

....
    public synchronized void ingresar (int dinero) throws InterruptedException {
        this.cantidad += dinero;
    }
...

```

Problema 2: Starvation

Esta situación sucede cuando en una aplicación con varios hilos, alguno de los hilos tiene poco acceso al recurso compartido (en comparación con el resto de hilos) y por lo tanto "progresar" en su ejecución.

Ejemplo: Se trata de un monedero compartido por 10 Personas. Cada persona intenta sacar 10€ x 4 veces.

Se utiliza synchronized para asegurarnos la integridad de la cantidad de dinero.

```
public class Monedero {  
  
    private int cantidad;  
  
    public Monedero(int cantidad) {  
        this.cantidad = cantidad;  
    }  
  
    public synchronized int sacarDinero(int dineroAsacar) throws
```

7 / 11
unidad3.md 3/22/2020

```
InterruptedException {  
    if (cantidad >= dineroAsacar){ // Comprobamos que haya suficiente dinero  
        cantidad -= dineroAsacar;  
        return dineroAsacar;  
    } else {  
        return 0;  
    }  
}  
}
```

```
public class Persona extends Thread{  
  
    private int id;  
    private Monedero monedero;  
    private int miDinero;  
  
    public Persona(Monedero monedero, int id) {  
        this.monedero = monedero;  
        this.id = id;  
        this.miDinero = 0;  
    }  
  
    @Override  
    public void run() {  
        for(int i=0; i<4; i++){  
            try {  
                miDinero += monedero.sacarDinero(10);  
            } catch (InterruptedException e) { }  
        }  
        System.out.println(id+" "+miDinero); //Al finalizar muestra el dinero que cada  
        hilo ha podido sacar  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {
```

```

Monedero m = new Monedero(100); // Saldo inicial 100€
for (int i=0; i<10; i++){ //Creamos e iniciamos a 10 personas new
Persona(m, i).start();
}
}
}

```

En su ejecución comprobaremos que 2 hilos han sacado 40€ y 1 hilo 20€. En vez de estar el dinero mas repartido la ejecución a priorizado (por las razones que sean) a esos 3 hilos.

Para conseguir un reparto mas equitativo podemos añadir la funcion `yield()` al codigo. **`yield()`** le dice al planificador de tareas que puede elegir otro hilo para ejecutar, o seguir ejecutando el mismo.

8 / 11
unidad3.md 3/22/2020

```

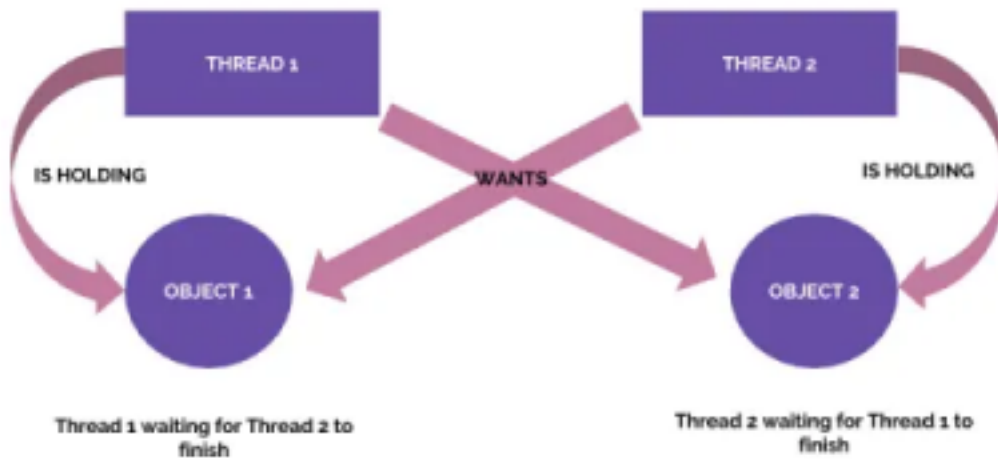
public class Persona extends Thread{
...
@Override
public void run() {
for(int i=0; i<4; i++){
try {
miDinero += monedero.sacarDinero(10);
} catch (InterruptedException e) {
e.printStackTrace();
}
this.yield(); // Lo añadido
}
System.out.println(id+" "+miDinero);
}
}

```

En este caso nos hubiera dado igual resultado utilizar **`sleep(100)`** o **`wait(100)`**.

Problema 3: Dead Lock

Es una situación que se produce cuando dos o más hilos están bloqueados de forma permanente esperándose mutuamente para desbloquearse. Un programa Java multihilo puede sufrir esta situación cuando se utiliza los bloques `synchronized`.



Ejemplo:

```
public class Cuenta {
    int saldo;

    public Cuenta(int saldo) {
```

9 / 11
unidad3.md 3/22/2020

```
    this.saldo = saldo;
}
```

```
public void sacarDinero(int dinero) {
    this.saldo -= dinero;
}
```

```
public void ingresarDinero(int dinero) {
    this.saldo += dinero;
}
}
```

```
public class Persona extends Thread {
```

```
    private Cuenta miCuenta;
    private Cuenta otraCuenta;
```

```
    public Persona(Cuenta miCuenta, Cuenta otraCuenta) {
        this.miCuenta = miCuenta;
        this.otraCuenta = otraCuenta;
    }
```

```
    @Override
    public void run() {
        synchronized (miCuenta){ //Bloqueo miCuenta
```

```

try {
    Thread.sleep(100); // Fuerzo el DeadLock
} catch (InterruptedException e) {
    e.printStackTrace();
}
synchronized (otraCuenta){ //Bloque la otraCuenta
miCuenta.sacarDinero(100);
    otraCuenta.ingresarDinero(100);
}
}
System.out.println("Ingreso realizado");
}
}

```

```

public class Main {

    public static void main(String[] args) {
        Cuenta cuenta1 = new Cuenta(1000);
        Cuenta cuenta2 = new Cuenta(1000);

        /* A cada persona le doy primero su cuenta propia, y después la cuenta de
        destino.
        Por eso es distinto para cada una de las personas.*/ new
        Persona(cuenta1, cuenta2).start();
    }
}

```

10 / 11
 unidad3.md 3/22/2020

```

new Persona(cuenta2, cuenta1).start();
}
}

```

No existe una fórmula o una solución única para Deadlock. Las opciones pueden ser:

Cambiando el orden de los bloqueos en los hilos, es decir, reconfigurando la lógica de tu aplicación.
 Utilizando herramientas avanzadas de concurrencia que proporcione el lenguaje de programación.
 En Java, Lock con el tryLock(), ExecutorService, ForkJoin, etc..

