Tema 5. Sincronización con Java

1. Exclusión mutua en JAVA

- Java no tiene semáforos a nivel del lenguaje aunque sí en su biblioteca estándar.
- Sin embargo proporciona otro tipo de primitivas con las que podemos gestionar los problemas de concurrencia
- Utiliza un tipo particular de monitor (tema 6) incorporado en su sintaxis
- En un enfoque orientado a objetos la concurrencia se traduce en que muchos hilos pueden estar ejecutando código de un mismo objeto

- Conseguimos la exclusión mutua mediante la palabra reservada `synchronized`
- Un método que lleve el modificador `synchronized` se ejecutará en exclusión mutua
- Cuando un *método sincronizado* se está ejecutando no se ejecutará ningún otro *método sincronizado* del mismo objeto

2. Utilización de *synchronized*

- Un ejemplo:

```java
public class Sincronizados {
    public              void m1() {}
    public synchronized void m2() {}
    public              void m3() {}
    public synchronized void m4() {}
    public              void m5() {}
}
```

- Gráficamente:

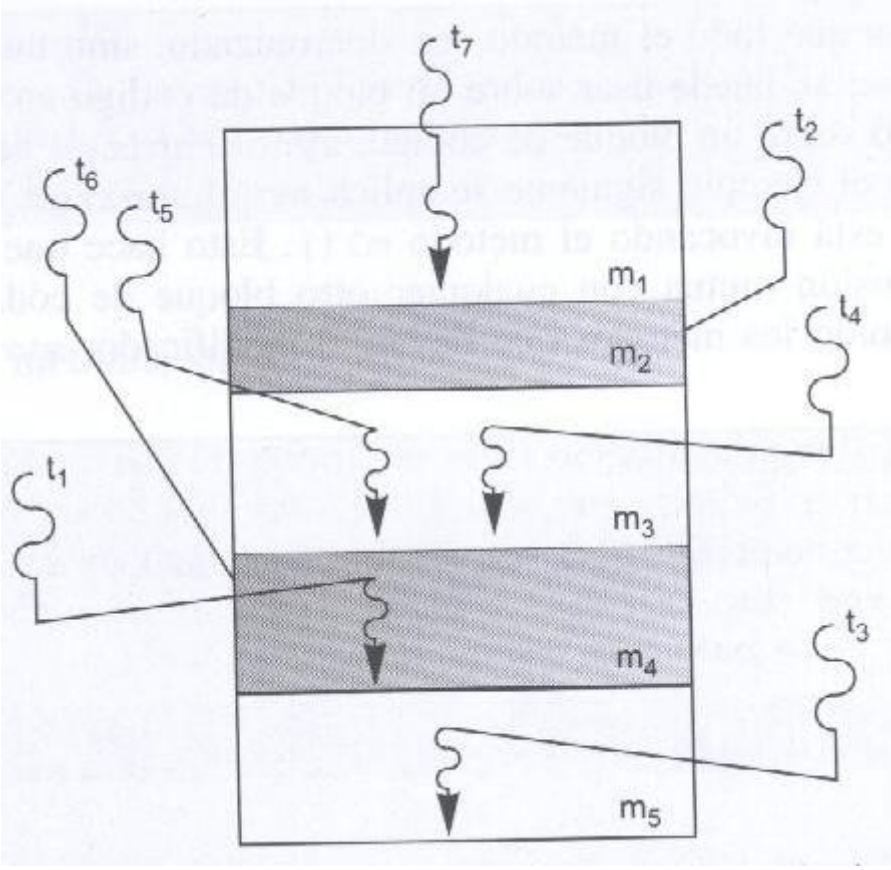

Figure 1: Ejemplo de sincronización en Java.

- Un bloque de código también puede ser sincronizado. En este caso necesitamos hacer referencia a un objeto:

```
1: public class Sincronizada {
2:     public void metodo1() {
3:           //instrucciones
4:
5:           synchronized(this) {
6:              //instrucciones que se ejecutarán en exclusión mutua c
7:              //otro bloque synchronized
8:           }
9:
10:          //instrucciones
11:     }
12: }
```

- El caso anterior lo podemos generalizar a cualquier *objeto*, no sólo `this`:

```
1: public class Sincronizada {
2:     public void metodo1() {
3:           //instrucciones
4:
5:           synchronized(otroObjeto) {
6:              //instrucciones que se ejecutarán en exclusión mutua c
7:              //el mutex de otroObjeto
8:           }
9:
10:          //instrucciones
11:     }
12: }
```

- Podemos sincronizar a nivel de métodos de clase también:

```
1: public class Sincronizada {
2:     public synchronized static void metodo1() {
3:           //instrucciones
4:     }
5: }
6: // O tambien
7: ...
8: synchronized (nombreClase.class) {
9:   //instrucciones
10: }
```

3. Variables volatile

- Las asignaciones entre variables de tipos primitivos se realizan de forma atómica, con lo cual no hace falta poner *synchronized*.
- Las dos clases siguientes son iguales:

```
public class s {
 public void f() {
   synchronized (this) {
     boolean b = true;
   }
 }
}
```

```
public class s1{
 public void f(){
     boolean b = true;
 }
}
```

- Las optimizaciones del compilador pueden dar problemas:

```
public class c {
   private boolean v = false;
   public void cambiaV() {
      v = true;
   }
   public synchronized void esperaVCierto() {
      v = false;
      if (v)
         //Código que no será ejecutado
   }
}
```

- `Synchronized` nos garantiza que ningún otro método sincronizado será ejecutado a la vez, pero el método **cambiaV** *no es sincronizado*, por lo tanto puede ser llamado mientras se está ejecutando **esperaVCierto**.
- El compilador de Java entenderá que el código de dentro del `if` no se ejecutará nunca, ya que en el método **esperaVCierto** hemos inicializado **v** a `false`, por lo tanto eliminará este código.
- Para evitar que el compilador haga estas optimizaciones declararemos **v** de tipo `volatile`
- Declarar una *variable* de tipo `volatile` es decirle al compilador que *otro hilo la puede modificar*

4. Sincronización

wait() :
le indica al hilo en curso que abandone la exclusión mutua (libere el *mutex*) y se vaya al estado de espera hasta que otro hilo lo despierte

notify() :
un hilo, elegido al azar, del conjunto de espera pasa al estado de listo

notifyAll() :
todos los hilos del conjunto de espera pasan a listos

5. Sincronización, guarda boolena

```
synchronized void hacerCuandoCondicion() {
  while(!c)
     try {
       wait();
     } catch(InterruptedException e) {}

  // Aquí irá código que se ejecutará
  // cuando la condicion 'c' sea cierta
}

synchronized void hacerCondicionVerdadera() {
  c = true;
  notify();  // o notifyAll()
}
```

- Vamos a tener dos conjuntos de hilos:
  - los que quieren acceder a la zona de *exclusión mutua*
  - los que están ``dormidos'' esperando a ser despertados por un `notify` o `notifyAll`

- Cuando un hilo hace `wait`, primero se suspende el hilo y luego se libera el ``cerrojo'' (*synchronized*), con lo cual otro hilo puede entrar
- El `while` es necesario, ya que *que se haya despertado un hilo* no nos garantiza que la variable **c** pase a valer `true`

- Un hilo despertado por `notify` tendrá que volver a luchar por conseguir el cerrojo.

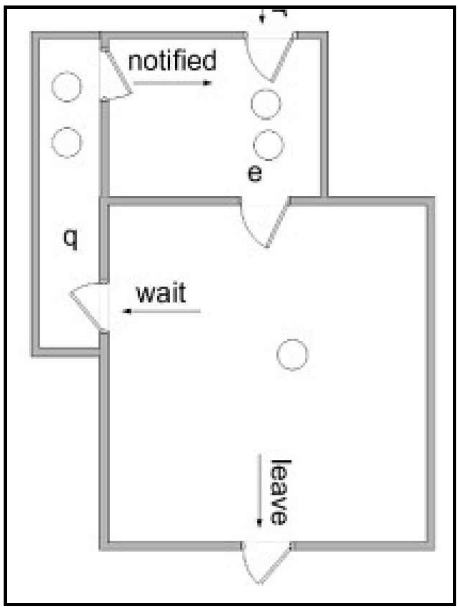

Figure 2: Ejemplo de guarda booleana.

6. Ejemplos de código en Java

6.1. Productor y Consumidor: clase Buffer

```
1: public class Buffer {
2:   private int cima, capacidad, vector[];
3:
4:   public Buffer(int i) {
5:     cima = 0;
6:     capacidad = i;
7:     vector = new int[i];
8:   }
9:
10:  synchronized public int extraer(int id) {
11:    System.out.println(id+": espera para leer");
12:    while (cima == 0)
13:      try {
14:        wait();
15:      } catch (InterruptedException e){}
16:    notifyAll();
17:    return vector[--cima];
18:  }
19:
20:  synchronized public void insertar(int elem,int id) {
21:   System.out.println(id+": espera para insertar");
22:   while (cima == capacidad)
23:     try {
24:      wait();
25:     } catch (InterruptedException e){}
26:   vector[cima]=elem;
27:   cima++;
28:   notifyAll();
29:  }
30: }
```

6.2. Productor y Consumidor: clase Consumidor

```
1: public class Consumidor extends Thread {
2:   int elem, id;
3:   Buffer buffer;
4:
5:   Consumidor(Buffer b, int numHilo) {
6:     buffer = b;
7:     id = numHilo;
8:     System.out.println("CONSUMIDOR " + id +": entra");
9:   }
10:
11:  public void run () {
12:    try {
13:      elem = buffer.extraer(id);
14:      System.out.println(id + ": he leido " + elem);
15:    } catch (Exception e) {}
16:  }
17: }
```

6.3. Productor y Consumidor: clase Productor

```
1: public class Productor extends Thread {
2:   Buffer buffer;
3:   int elem, id;
4:
5:   Productor(Buffer b, int i, int numHilo) {
6:     elem = i;
7:     buffer = b;
8:     id = numHilo;
9:     System.out.println("PRODUCTOR " + id + ": entra");
10:  }
11:
12:  public void run () {
13:    try {
14:      buffer.insertar(elem, id);
15:      System.out.println(id + ": inserta " + elem);
16:    } catch (Exception e) {}
17:  }
18: }
```

6.4. Productor y Consumidor: clase Principal

```
1: public class ProductorConsumidor {
2:   static Buffer buf = new Buffer(3);
3:   static int numcons = 7;
4:   static int numprods = 5;
5:
6:   public static void main(String[] args) {
7:      for(int i = 1; i <= numprods; i++)
8:        new Productor(buf,i,i).start();
9:
10:     for(int k = 1; k <= 1_000_000_000; k++);
11:       System.out.println();
12:
13:     for(int j = 1; j <= numcons; j++)
14:       new Consumidor(buf, j).start();
15:
16:     System.out.println("Fin del hilo main");
17:  }
18: }
```

6.5. Productor y Consumidor: traza

```
| PRODUCTOR 1: entra      | CONSUMIDOR 1: entra | CONSUMIDOR 5: entra
| PRODUCTOR 2: entra      | CONSUMIDOR 2: entra | 4: espera para leer
| 1: espera para insertar | 1: espera para leer | 4: he leido 2
| 1: inserta 1            | 4: inserta 4        | CONSUMIDOR 6: entra
| PRODUCTOR 3: entra      | 1: he leido 3       | 5: espera para leer
| 2: espera para insertar | CONSUMIDOR 3: entra | 5: he leido 1
| 2: inserta 2            | 2: espera para leer | 6: espera para leer
| PRODUCTOR 4: entra      | 5: inserta 5        | CONSUMIDOR 7: entra
| 3: espera para insertar | 2: he leido 4       | Fin del hilo main
| 3: inserta 3            | CONSUMIDOR 4: entra | 7: espera para leer
| PRODUCTOR 5: entra      | 3: espera para leer |
| 4: espera para insertar | 3: he leido 5       |
| 5: espera para insertar |                     |
```

7. Otros problemas clásicos

- Vemos el código fuente de:
  - **ReaderWriter.java**
  - **Philosopher.java**

8. Semáforo binario

```
1: public class SemaforoBinario {
2:   protected int contador = 0;
3:
4:   public SemaforoBinario(int valorInicial) {
5:     contador = valorInicial; // 1 o 0
6:   }
7:
8:   synchronized public void WAIT () {
9:     while (contador == 0 )
10:      try {
11:        wait();
12:      } catch (Exception e) {}
13:    contador--;
14:  }
15:
16:  synchronized public void SIGNAL () {
17:    contador = 1;
18:    notify();
19:  }
20: }
```

9. Semáforo general

```
1: public class SemaforoGeneral extends SemaforoBinario {
2:
3:   public SemaforoGeneral(int valorInicial) {
4:     super(valorInicial); // numero natural
5:   }
6:
7:   synchronized public void  SIGNAL () {
8:     contador ++;
9:     notify();
10:  }
11: }
```

9.1. Otras utilidades en el API de Java

- El API de Java 10:
  - java.util.concurrent
  - java.util.concurrent.atomic

- En los enlaces anteriores encontrarás documentación de los *cierres de exclusión mutua*, *barreras*, *semáforos*, *colecciones concurrentes* (vectores, conjuntos, tablas, colas).

9.2. Cerrojos (Locks) y Variables Condición

- java.util.concurrent.locks
- Interfaces **Lock** y **Condition**
  - **Lock** permite gestionar los cierres de forma explícita
    - `lock()`
    - `unlock()`
    - `tryLock()`
    - `newCondition()`
  - **Condition** permite establecer más de una cola asociada a un cierre
    - `await()`
    - `awaitUntil()`
    - `signal()`
    - `signalAll()`

9.3. Colecciones seguras

BlockingQueue :
Define una estructura FIFO que se bloquea (o espera un tiempo máximo determinado) cuando se intenta añadir un elemento a una cola llena o retirar uno de una cola vacía

ConcurrentMap :
Define una tabla hash con operaciones atómicas

9.4. Variables atómicas

- `java.util.concurrent.atomic`
- Conjunto de clases que permiten realizar operaciones atómicas con variables de tipos básicos
  - `AtomicBoolean`
  - `AtomicInteger`
  - `AtomicIntegerArray`
  - `AtomicLong`
  - `AtomicLongArray`
  - ...

- Por ejemplo, `AtomicInteger` dispone de:
  - addAndGet
  - compareAndSet
  - decrementAndGet
  - getAndAdd
  - getAndDecrement
  - getAndIncrement
  - getAndSet
  - incrementAndGet
  - intValue
  - lazySet
  - set

- Vemos el código de `PhilosopherConditions.java`

9.5. Para saber más ...

- Echa un vistazo al tutorial sobre concurrencia de Java.

9.5.1. Aclaraciones

- **En ningún caso estas transparencias son la bibliografía de la asignatura**, por lo tanto debes estudiar, aclarar y ampliar los conceptos que en ellas encuentres empleando los enlaces web y bibliografía recomendada que puedes consultar en la página web de la ficha de la asignatura y en la web propia de la asignatura.