Programación Concurrente y de Tiempo Real

Guión de prácticas 8: Monitores en Java (API estándar)

Natalia Partera Jaime
Alumna colaboradora de la asignatura

Índice

1. Introducción

En el guión anterior logramos ejecutar varios hilos sincronizados y en exclusión mutua usando los mecanismos más sencillos. En la programación concurrente existen algunos mecanismos más complejos con los que también podemos lograr la sincronización y exclusión mutua. En este guión estudiaremos uno de ellos: el monitor.

2. Monitores

Los monitores son una técnica para lograr sincronización y exclusión mutua y que se basan en la utilización de memoria compartida para conseguir la sincronización necesaria.

2.1. Concepto de monitor

Imagine un elemento donde guarda los datos que deben trabajar en exclusión mutua. Suponga además que ese elemento dispone de una serie de operaciones para trabajar con esos datos, de tal modo que sólo es posible acceder a esos datos a través de esas operaciones. Pues ese es, en general, el concepto de un monitor en la programación concurrente.

Un monitor es un objeto que encapsula las representaciones de recursos abstractos y proporcionan el acceso a estos recursos mediante sus métodos, implementados bajo exclusión mutua y que, además, proveen de sincronización.

Un proceso, o cualquier otro elemento externo, sólo puede acceder a los métodos del monitor. Los métodos que pertenecen a un mismo monitor, se ejecutan en exclusión mutua entre sí. Si un proceso intenta ejecutar un método de un objeto cuando ya hay otro proceso ejecutando algún método de ese mismo objeto, el proceso pasa a una cola de espera (llamada *cola del monitor*). Cuando finaliza la ejecución del método que mantenía ocupado al monitor, se selecciona a uno de los procesos bloqueados en la cola, si hay, y lo desbloquea. Esta es la manera en que los monitores controlan la exclusión mutua.

Para controlar la sincronización, los monitores utilizan las variables de condición. Estas variables de condición permiten bloquear procesos activos (que están ejecutando algún método en el monitor) que no pueden seguir con su ejecución. Las variables de condición también permiten desbloquear a estos procesos cuando la situación que provocó su bloqueo ya no se dé.

Según los métodos que el monitor utilice para avisar a los hilos bloqueados, y según el funcionamiento de estos métodos, se dice que el monitor sigue una política de señalización u otra.

Como puede comprobar, el concepto de monitor teórico se parece bastante a los bloques sincronizados del guión anterior. Sin embargo, el monitor tiene algunas características concretas que no afectaban a los bloques sincronizados.

2.2. El monitor en Java

En Java los monitores se implementan como objetos de una clase, donde sus atributos son privados y sus métodos públicos, que modifican los atributos, son todos `synchronized`. Como recordará del guión anterior, un método `synchronized` impide que otro método `synchronized` del mismo objeto pueda ser ejecutado simultáneamente. Esto lo logra haciendo uso de un cerrojo sobre el objeto, y proporcionando así exclusión mutua sobre los métodos del objeto.

De acuerdo con el concepto de monitor (explicado previamente) y con el uso de los bloques sincronizados (explicados en el guión anterior) sabemos que cuando sobre un objeto se encuentra un método sincronizado en ejecución y se intenta ejecutar otro método sobre el objeto, éste último método debe ir a una cola de espera. En el caso de los monitores, esta cola se llama *cola del monitor*, y existe una cola por cada objeto monitor que se encuentre creado en el sistema.

El funcionamiento de esta cola es similar, por no decir idéntico, al del conjunto de hilos en espera de los objetos sobre los que se establece un cerrojo. Para sincronizar los accesos al monitor se usan los métodos `wait()`, `notify()` y `notifyAll()`.

Antes de seguir profundizando, veamos cómo es la estructura sintáctica de un monitor en Java:

```
class Monitor {
  //Datos protegidos por el monitor.
  private tipoDato1 dato1;
  private tipoDato2 dato2;
  //...

  //Constructor
  public Monitor(){...}

  //Funciones públicas y sincronizadas que acceden a los datos protegidos por el
  //monitor.
  public synchronized tipo1 metodo1(tipoArg1 arg1) throws InterruptedException {
    ...
    notifyAll();
    ...
    while(!condicion1)
      wait();
    ...
  }

  public synchronized tipo2 metodo2() throws InterruptedException {
    ...
    notifyAll();
    ...
    while(!condicion2)
      wait();
    ...
  }
}
```

3. Funcionamiento de los monitores en Java

Como hemos visto, el monitor teórico utiliza variables de condición para conseguir la sincronización. Sin embargo, en Java no podemos desbloquear sólo a un método en función de la condición que éste necesita que se cumpla. Es por ello, que en Java debemos utilizar los métodos `wait()`, `notify()` y `notifyAll()` junto con variables globales (a las que llamaremos *condiciones de guarda*) para simular la sincronización del monitor teórico. Esta técnica es menos selectiva que uso de variables de condición.

Una de las partes más importantes de los monitores es la *cola del monitor*. Sobre ella podemos ejecutar tres métodos. Veamos cómo se comporta según el método que ejecutemos:

- Cuando un método `synchronized` del monitor llama a `wait()` libera la exclusión mutua existente sobre el monitor y encola al hilo que llamó al método en el `wait-set`. Esto se da cuando, por cualquier razón, el método que está siendo ejecutado debe bloquearse. Por ejemplo, cuando el método comprueba durante su ejecución si la condición que necesita se ha cumplido y al no ser así vuelve a bloquearse.
- Cuando otro método del monitor hace `notify()`, un hilo del `wait-set` (Java no especifica cuál) pasará a la cola de hilos que esperan el cerrojo y se reanudará cuando sea planificado. Este método se suele usar cuando se cumpla una condición y dé igual cual sea el siguiente hilo que se ejecute.
- Cuando otro método del monitor hace `notifyAll()`, todos los hilos del `wait-set` pasarán a la cola de hilos que esperan el cerrojo y se reanudarán cuando sean planificados. Este caso se desbloquean todos los hilos y cada uno puede comprobar si se ha cumplido su condición.

Como en Java no es posible señalizar a un hilo en especial, los pasos que se siguen son los siguientes:

- Todos los hilos que quieran ejecutar un método del monitor y no puedan se bloquean en el `wait-set` y serán desbloqueados cuando se produzca un `notifyAll()`.
- Cada vez que un hilo se despierte, continuará con la instrucción que sigue a `wait()`. Habrá entonces que comprobar si se ha cumplido su condición y si no fuera así, volverá a ejecutar `wait()`. Normalmente, se utiliza para ello el siguiente código dentro de los métodos `synchronized`:

```
...
while (!condicion)
  try{
    wait();
  } catch (InterruptedException e) {
    return ;
  }
...
```

- Al ser despertados, los hilos que comprueben su condición y la encuentren verdadera pasarán a la espera del cerrojo sobre el monitor.

En Java, los monitores siguen una política de señalización de `señalar y seguir` (SC), también llamada política de `desbloquear y continuar`. Esto se debe al funcionamiento de los métodos `wait()`, `notify()` y `notifyAll()`. Cuando un proceso desbloquea a otros procesos utilizando `notify()` o `notifyAll()`, continua con su ejecución hasta que sale del monitor. A partir de ese momento, es cuando el proceso que acaba de ser desbloqueado puede acceder al monitor. La desventaja de esta política, es que al seguir con su ejecución el hilo que ha desbloqueado, la variable de condición puede volver a incumplirse antes de que el hilo desbloqueado consiga ser ejecutado.

Ejercicio 1 Complete la siguiente tabla sobre el funcionamiento de los monitores. Cuando finalice, compruebe sus resultados con los que aparecen en el apartado 5.1.

Monitores	Teórico	Java
Protección de datos compartidos		
Exclusión Mutua		
Sincronización		

4. Técnica de diseño de monitores en Java

Para usar monitores en Java como control de la concurrencia debemos cuidar el diseño de los monitores y de los hilos que los usan si queremos garantizarnos su correcto funcionamiento. En este apartado veremos los pasos claves para un correcto diseño de monitores y su uso.

4.1. Diseño de monitores

Los monitores son los objetos de una clase dada que encapsulan la información común que debe ser tratada bajo exclusión mutua. Para diseñar correctamente esta clase, debemos seguir los siguientes pasos:

1. Decidir qué datos debemos encapsular en el monitor.
2. Construir un monitor teórico, utilizando tantas variables de condición como sean necesarias.
3. Usar señalización SC en el monitor teórico.
4. Implementar el monitor teórico en Java. Para ello:
   a) Escribir un método `synchronized` por cada procedimiento.
   b) Implementar los datos encapsulados como `private`.
   c) Sustituir cada

   ```
   wait(variable_condición)
   ```

   por una condición de guarda

   ```
   while(!condición)
     try{
       wait();
     }
     ...
   ```

   d) Sustituir cada

   ```
   send(variable_de_condición)
   ```

   por una llamada a

   ```
   notifyAll();
   ```

   e) Escribir el código de inicialización del monitor en el constructor del mismo.

Veamos un ejemplo. Supongamos un ascensor de un edificio. Cuando alguien llama al ascensor desde un piso, el ascensor acudirá cuando pueda. Si el ascensor está parado, irá inmediatamente. En cambio, si el ascensor está en movimiento, acudirá cuando termine de subir o bajar como tenía previsto. Podemos pensar que el dato importante que controlar es el piso en el que se encuentra el ascensor.

En cuanto a sus métodos, el ascensor subirá o bajará cuando esté parado, pero no cambiará su destino mientras está en movimiento. Parece que con una sola variable de condición que controle si el ascensor está parado, o no, es suficiente.

Observe la siguiente clase que simula al citado ascensor:

```
/**
 * Clase Ascensor.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class Ascensor {

  //Atributo privado
  private int piso;
  private boolean parado;

  //Constructor de la clase
  public Ascensor() {
    piso = 0;
    parado = true;
  }

  //Método que si dada la posición del ascensor y el piso desde el que le llaman,
  //controla si el ascensor sube o baja
  public synchronized void llamar(int p, String mensaje) {
    while(!parado) {
      try {
        wait();
      } catch (InterruptedException e) {
        return ;
      }
    }
    if(piso > p) {
      bajar(p, mensaje);
    }
    else if(piso < p) {
      subir(p, mensaje);
    }
  }

  //Método modificador que simula la subida del ascensor. Muestra un mensaje cuando
  //el ascensor ha llegado al destino.
  public synchronized void subir(int p, String mensaje) {
    while(!parado) {
      try {
        wait();
      } catch (InterruptedException e) {
        return ;
      }
    }
    parado = false;
    while (piso != p) {
      ++piso;
```

```
      System.out.println("Ascensor en el piso " + piso);
    }
    parado = true;
    System.out.println(mensaje);
    notifyAll();
  }

  //Método modificador que simula la bajada del ascensor. Muestra un mensaje cuando
  //el ascensor ha llegado al destino.
  public synchronized void bajar(int p, String mensaje) {
    while(!parado) {
      try {
        wait();
      } catch (InterruptedException e) {
        return ;
      }
    }
    parado = false;
    while (piso != p) {
      --piso;
      System.out.println("Ascensor en el piso " + piso);
    }
    parado = true;
    System.out.println(mensaje);
    notifyAll();
  }

  //Método observador que muestra el piso
  public synchronized void mostrarPiso() {
    System.out.println("El ascensor está en el piso " + piso);
  }

  //Método observador que devuelve el piso
  public synchronized int piso() {
    return piso;
  }

}
```

Hemos creado algunos métodos más para controlar el funcionamiento del ascensor por si fueran necesarios para hacer pruebas. Veamos un programa de prueba con el que comprobar el correcto funcionamiento de los métodos del monitor:

```
/**
 * Programa que prueba el funcionamiento de Ascensor.
 *
 * @author Natalia Partera
 * @version 1.0
 */

class SubirAscensor extends Thread{
  private Ascensor ascensor;
```

```
  public SubirAscensor(Ascensor asc) {
    ascensor = asc;
  }

  public void run() {
    for(;;)
      ascensor.subir(ascensor.piso() + 3, "Sube 3 pisos.");
  }
}

class BajarAscensor extends Thread{
  private Ascensor ascensor;

  public BajarAscensor(Ascensor asc) {
    ascensor = asc;
  }

  public void run() {
    for(;;)
      ascensor.bajar(ascensor.piso() - 2, "Baja 2 pisos.");
  }
}

public class PruebaAscensor {
  //Programa principal
  public static void main (String[] args) {
    Ascensor ascensor;
    ascensor = new Ascensor();

    new SubirAscensor(ascensor).start();
    new SubirAscensor(ascensor).start();
    new SubirAscensor(ascensor).start();
    new BajarAscensor(ascensor).start();
    new BajarAscensor(ascensor).start();
  }
}
```

Ejercicio 2 Diseñe un monitor en Java que sirva para controlar la venta de entradas de un evento. Deberá indicar al menos el inicio de la venta, el fin de la venta y la numeración de la entrada vendida. Si lo desea, puede solicitar más datos o invertir tiempo del procesador para emular que se hacen otras operaciones. Cuando termine, compare su resultado con la clase que se propone en el apartado 5.2.

4.2. Diseño de hilos sobre el monitor

Los métodos del monitor pueden ser invocados desde el programa principal. Pero en ocasiones puede que prefiramos llamar a estos métodos desde varios hilos. Para poder llamar a los métodos del monitor desde un hilo, debemos crear una clase para el hilo en la que tengamos como atributo privado un objeto de la clase monitor.

Este atributo (el objeto de la clase implementada como monitor) debe ser inicializado en el constructor de la clase que representa al hilo. Si varios hilos comparten el objeto implementado como monitor, que suele ser lo común, debemos crear este objeto como variable local del programa principal y pasarlo al constructor de los hilos.

Siguiendo estas indicaciones, veamos cómo crear hilos para la clase `Ascensor` del apartado anterior. Supongamos usuarios que llamarán al ascensor para que les recojan en un piso y le dejen en otro. Varios usuarios pueden llamar al ascensor a la vez desde diferentes sitios, pero el ascensor sólo cambiará su movimiento cuando esté parado. Los usuarios serán los hilos, y tendrán un método para llamar al ascensor. Para ver el comportamiento del ascensor cuando le llaman varias veces, cada uno de los hilos realizará varias llamadas. Este es el código que representa a los usuarios:

```
/**
 * Clase Usuario.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class Usuario extends Thread {

  //Atributo privado
  private String nombre;
  private int piso;
  private Ascensor ascensor;
  private int[] pisos;

  //Constructor de la clase
  public Usuario(Ascensor a, String n) {
    nombre = n;
    piso = 0;
    ascensor = a;
  }

  //Constructor de la clase
  public Usuario(Ascensor a, String n, int p, int[] lp) {
    nombre = n;
    piso = p;
    ascensor = a;
    pisos = lp;
  }

  public void VerPisos() {
    for(int i = 0; i < pisos.length; ++i)
      System.out.println(nombre + " va al piso " + pisos[i]);
  }

  //Método que llama al ascensor para ir de un piso a otro
  public void llamarAscensor(int destino) {
    System.out.println("El usuario " + nombre + " está esperando en el piso " +
      piso + " para ir al piso " + destino);
    String mensaje1 = "El usuario " + nombre + " se ha montado en el ascensor en " +
```

```
      "el piso " + piso;
    ascensor.llamar(piso, mensaje1);
    String mensaje2 = "El usuario " + nombre + " ha llegado al piso " +  destino;
    ascensor.llamar(destino, mensaje2);
    piso = destino;
  }

  //Método run
  public void run() {
    for(int i = 0; i < pisos.length; ++i) {
      llamarAscensor(pisos[i]);
    }
  }
}
```

Hay que recordar, que para que un sólo objeto ascensor dé servicio a varios hilos usuarios, debemos crear este objeto ascensor en el programa principal y pasárselo al constructor de los hilos. A continuación puede ver el programa de prueba:

```
/**
 * Programa que muesta el funcionamiento de Ascensor.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class UsaAscensor {

  static void inicializarPisos(int[] p1, int[] p2, int[] p3) {
    p1[0] = 0;
    p1[1] = 2;
    p1[2] = 1;
    p1[3] = 3;
    p1[4] = 0;
    p1[5] = 5;
    p1[6] = 7;
    p1[7] = 4;
    p1[8] = 6;
    p1[9] = 0;
    p2[0] = 3;
    p2[1] = 1;
    p2[2] = 6;
    p2[3] = 3;
    p2[4] = 8;
    p2[5] = 0;
    p2[6] = 5;
    p2[7] = 2;
    p3[0] = 8;
    p3[1] = 4;
    p3[2] = 6;
    p3[3] = 0;
    p3[4] = 5;
```

```
    p3[5] = 1;
    p3[6] = 7;
    p3[7] = 2;
    p3[8] = 8;
    p3[9] = 0;
    p3[10] = 3;
    p3[11] = 5;
  }

  //Programa principal
  public static void main (String[] args) {
    Ascensor ascensor;
    ascensor = new Ascensor();

    int[] p1 = new int[10];
    int[] p2 = new int[8];
    int[] p3 = new int[12];
    UsaAscensor.inicializarPisos(p1, p2, p3);

    Usuario manolita = new Usuario(ascensor, "Manolita", 0, p1);
    Usuario pepe = new Usuario(ascensor, "Pepe", 3, p2);
    Usuario juan = new Usuario(ascensor, "Juan", 8, p3);

    manolita.start();
    pepe.start();
    juan.start();
  }

}
```

Ejercicio 3 Compile el ejemplo de las clases `Ascensor` y `Usuario` presentadas en este apartado y el anterior. Ejecútelas utilizando el programa de prueba que se expone en este apartado. ¿Qué observa? ¿Refleja el funcionamiento normal de un ascensor? Razone su respuesta.

Ejercicio 4 Diseñe e implemente hilos y un programa de prueba para el monitor del ejercicio 2. Cuando acabe, compruebe que el resultado de ambos ejercicios se ejecuta como cabía esperar y compruebe su código con el que se encuentra en el apartado 5.3.

Ejercicio 5 Realice un programa que ejecute 3 hilos. Estos hilos llamarán al método `comprobar()` de una clase que diseñará como monitor en Java. La clase que actúa como monitor deberá bloquear a los dos primeros hilos que llamen a su función. El tercer hilo que llame a esa función, hará que se desbloqueen los demás. Cuando termine, compruebe su solución con la que puede encontrar en el apartado 5.4.

---

5. Soluciones de los ejercicios

Compruebe los resultados de sus ejercicios con estas soluciones.

5.1. Ejercicio1

A continuación puede ver el cuadro del ejercicio 1 completo:

Monitores	**Teórico**	**Java**
Protección de datos compartidos	Sólo se puede acceder a los datos a través de las operaciones del objeto monitor.	Datos privados, operaciones públicas. Modificación de los datos a través de las operaciones.
Exclusión Mutua	Sólo se puede ejecutar un método del monitor a la vez. Uso de la *cola del monitor*.	Todos los métodos son `synchronized`, sólo se puede ejecutar un método del objeto a la vez. Uso del *cerrojo* y la *cola de espera*.
Sincronización	Uso de las variabes de condión: permiten desbloquear a los procesos cuando la situación que provocó su bloqueo ya no se dé.	Uso de `wait()`,`notify()`, `notifyAll()` y condiciones de guarda: como no es posible avisar sólo al metodo cuya condición se haya cumplido, se avisa a todos (`notifyAll()`) y cada uno comprueba su condición de guarda. Si no se cumple, `wait()`.

5.2. Ejercicio 2

El siguiente código representa la venta de entradas de manera concurrente.

```
/**
 * Clase VentaEntradas que emula la venta concurrente de entradas.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class VentaEntradas {

  //Atributo privado
  private int numeracion;
  private boolean libre;

  //Constructor de la clase
  public VentaEntradas() {
    numeracion = 0;
    libre = true;
  }

  //Método que simula la venta de entradas
  public synchronized void venta() {
    while(!libre) {
```

```
      try {
        wait();
      } catch (InterruptedException e) {
        return ;
      }
    }
    libre = false;
    System.out.println("[Inicio de la venta]");
    ++numeracion;
    //Para emular el tiempo que pudiera tardar en imprimir una entrada o en
    //solicitar y/o registrar otros datos, usamos el siguiente bucle
    for(int i = 0; i < 10000; ++i) {}

    System.out.println("Ha sido vendida la entrada núm. " + numeracion);
    libre = true;
    System.out.println("[Fin de la venta]");
    notifyAll();
  }
}
```

5.3. Ejercicio 4

A continuación puede ver unos hilos que se ejecutan sobre la clase `VentaEntradas` y un programa de prueba.

```
/**
 * Programa que prueba el funcionamiento de la clase VentaEntradas.
 *
 * @author Natalia Partera
 * @version 1.0
 */

class Hilo implements Runnable {
  private VentaEntradas entradas;

  public Hilo(VentaEntradas ent) {
    entradas = ent;
  }

  public void run() {
    for(int i = 0; i < 20; ++i)
      entradas.venta();
  }
}

public class UsaVentaEntradas {
  //Programa principal
  public static void main (String[] args) {
    VentaEntradas entradas;
    entradas = new VentaEntradas();
```

```
    Thread hilo1 = new Thread(new Hilo(entradas));
    Thread hilo2 = new Thread(new Hilo(entradas));
    Thread hilo3 = new Thread(new Hilo(entradas));

    hilo1.start();
    hilo2.start();
    hilo3.start();
  }
}
```

5.4. Ejercicio 5

La clase que representa a un monitor en Java y que contiene el método `comprobar()` es la que sigue:

```
/**
 * Clase Monitor.
 *
 * @author Natalia Partera
 * @version 1.0
 */

public class Monitor {

  //Atributo privado
  private int hilos;
  private boolean despertar, despertando;

  //Constructor de la clase
  public Monitor() {
    hilos = 0;
    despertar = false;
    despertando = false;
  }

  //Método que despierta a todos los hilos si ya han sido suspendidos 2 previamente
  public synchronized void comprobar() {
    if (despertando) {
      if (hilos == 0) {
        despertando = false;
        System.out.println("Todos los hilos están despiertos.");
      }
      else
        notifyAll();
    }
    else {

    if (hilos == 2) {
      System.out.println("Un hilo despierta a los demás.");
      despertar = true;
      despertando = true;
      notifyAll();
```

```
    }
    else {
      System.out.println("Un hilo ha sido suspendido.");
      ++hilos;
      despertar = false;
      while(!despertar) {
        try {
          wait();
        } catch (InterruptedException e) {
          return ;
        }
      }
      System.out.println("Un hilo se despierta.");
      --hilos;
      }
    }
  }
}
```

Los hilos han sido definidos en el mismo fichero que el programa principal, y se ejecutarán hasta que el usuario pare el programa. A continuación puede ver el código de los hilos y el programa principal:

```
/**
 * Programa que prueba el funcionamiento del monitor.
 *
 * @author Natalia Partera
 * @version 1.0
 */

class Hilo implements Runnable {
  private Monitor monitor;

  public Hilo(Monitor mon) {
    monitor = mon;
  }

  public void run() {
    for(;;)
      monitor.comprobar();
  }
}

public class PruebaMonitor {
  //Programa principal
  public static void main (String[] args) {
    Monitor monitor;
    monitor = new Monitor();

    Thread hilo1 = new Thread(new Hilo(monitor));
    Thread hilo2 = new Thread(new Hilo(monitor));
    Thread hilo3 = new Thread(new Hilo(monitor));
```

```
    hilo1.start();
    hilo2.start();
    hilo3.start();
  }
}
```