Implementación de Mecanismos de Sincronización

José Luis Quiroz Fabián

1. Introducción

La sincronización de los hilos se puede realizar mediante diferentes mecanismos: candados, semáforos, barreras, etc. Un candado es una variable que sirve para garantizar exclusión mutua (si se una de forma adecuada). Un *Semáforo* es una variable especial que sirve para restringir o permitir el acceso a recursos compartidos en un entorno de multi-procesamiento. Otro mecanismo de sincronización son las *Barreras*, las cuales permiten detener en un punto a un conjunto de procesos.

2. Candado

Una clase que permite crear un candado en Java es **ReetrantLock**. Esta clase define los métodos **lock** y **unlock** entre otros métodos. La operación **lock** solicita acceso a la zona de código crítica y la operación **unlock** libre la zona crítica. En Java, el hilo que realiza la operación **lock** está obligado a realizar la operación **unlock**. En el archivo **SumaConcurrente.java** se realiza el incremento de una variable compartida llamada suma **N** veces, donde **N** es igual a 200000. Esta suma se realiza de forma concurrente por varios hilos. La suma se protege por un candado (líneas 22-27) lo cual permite obtener el resultado esperado.

```
1
2  import java.util.concurrent.locks.ReentrantLock;
3
4  class Suma implements Runnable{
5    static final int N=200000;
6    static int suma=0;
7    private String nombre;
8    private ReentrantLock re;
9    public Suma(ReentrantLock rt, String n)
10   {
11     re = rt;
12     nombre = n;
13   }
14   public void run()
15   {
16     System.out.println("Hilo: "+nombre);
17     for(int i=0;i<N;i++)
18     {
19
20
21       re.lock();
22
23       suma++;
```

```
24
25
26          re.unlock();
27
28
29      }
30
31    }
32  }
```

micandado/Suma.java

La clase Principal realiza la creación de los hilos (líneas 12-16) y muestra el resultado (línea 29).

```
1  import java.util.concurrent.locks.ReentrantLock;
2
3  public class Principal{
4    static final int MAX_T = 4;
5    public static void main(String[] args)
6    {
7      ReentrantLock rel = new ReentrantLock();
8
9      Thread[] t = new Thread[MAX_T];
10
11
12     for(int i=0;i<MAX_T;i++) {
13
14       t[i]=  new Thread(new Suma(rel, "T"+i));
15       t[i].start();
16     }
17
18     for(int i=0;i<MAX_T;i++){
19
20       try {
21         t[i].join();
22
23       } catch (InterruptedException e) {
24
25         System.out.println("Error: en la espera del hilo");
26       }
27     }
28
29     System.out.println("Resultado final:"+Suma.suma);
30
31
32   }
33 }
```

micandado/Principal.java

Compilación: Ingresando al directorio de los fuentes ejecutar:

```
javac -d . *.java
```

Compilación: Ingresando al directorio de los fuentes ejecutar:

```
java mx.uam.pc.candado.Principal
```

2.1. Implementación

Si bien la mayoría de los lenguaje de programación proporcionan mecanismos de sincronización, ¿qué se requiere para implementar nuestro propio mecanismo?

```
1
2  import java.util.concurrent.ConcurrentLinkedQueue;
3  import java.util.concurrent.atomic.AtomicBoolean;
4  import java.util.concurrent.locks.LockSupport;
5
6  public class Candado{
7
8    private ConcurrentLinkedQueue<Thread> cola = new
       ConcurrentLinkedQueue<Thread>();
9
10
11   private final AtomicBoolean estado;
12   @SuppressWarnings("unused")
13   private Thread propietario=null;
14
15
16
17   public Candado() {
18
19     estado =  new AtomicBoolean(true);
20   }
21
22   public Candado(boolean valor) {
23
24     estado =  new AtomicBoolean(valor);
25   }
26
27
28   public void lock() {
29
30     boolean interrumpido = false;
31     //Hilo actual
32     Thread hilo = Thread.currentThread();
33
34     //Se forma el hilo en la cola concurrente
35     cola.add(hilo);
36
37     //System.out.println("Thread: "+hilo.getId()+"-> Lock 0");
38
39     //El hilo se bloquea si no esta al frente o bien si no puede
       adquirir el candado
40     while (cola.peek() != hilo || !estado.compareAndSet(true, false)) {
41
42       //El método park() deshabilita el hilo actual del planificador
       para propósitos de programación de hilos.
43       //El hilo se habilitara si ocurre:
44       //   1.- Otro hilo ejecuta el método unpark con el el valor del
       hilo actual
45       //   2.- Otro hilo interumpe al hilo actual
46       //   3.- Otra razon propia de eventos en la máquina virtal
47       LockSupport.park( );
48
49     }
50
51     propietario=cola.remove(); //Guardamos el propitario del candado
52
53   }
54
55
56
57
58   public boolean tryLock() {
```

```
59
60      boolean flag=estado.compareAndSet(true, false);
61      if(flag) {
62        propietario=Thread.currentThread();
63        return true;
64      }
65      return false;
66    }
67
68
69
70    public void unlock() {
71
72      estado.set(true);
73      //El metodo unpark (Thread thread) habilita el hilo dado
74      LockSupport.unpark(cola.peek());
75
76
77    }
78
79  }
```

micandado/Candado.java

Pruebe la clase principal con este candado.

2.2. Ejericicio

Realizar algo similar, pero implementando una barrera 0 un semáforo desde 0. Debe proponer una clase para probar su implementación. NO USAR MÉTODOS O BLOQUES **synchronized**.