Sincronización de Hilos

José Luis Quiroz Fabián

1. Introducción

Como hemos estudiado en este curso, la concurrencia en muchos casos puede mejorar el desempeño de nuestras aplicaciones ya que nos permite realizar varias tareas de forma simultánea. Ejemplos de lo anterior es cuando utilizamos varias pestañas en nuestro navegador web, o bien cuando nuestras aplicaciones de oficina revisan al momento el texto que estamos escribiendo. Sin embargo, el precio a pagar al desarrollar aplicaciones concurrentes es sin duda la complejidad subyacente en las mismas. No obstante, se han implementado diferentes mecanismos de sincronización a fin de garantizar un resultado çorrecto.[en] la ejecución de nuestras aplicaciones.

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. Además de los semáforos y barreras se tienen los *SpinLock*.

2. Ejemplos de mecanismos de sincronización en Java

Java cuenta con varios mecanismos de sincronización. En esta sección solo se muestra dos ejemplo usando el mecanismo candado y el semáforo.

2.1. Candados

En el archivo **Suma.java** realiza el incremento de una variable compartida llamada suma (línea 5 y 19) **N** veces, donde **N** es igual a 200000. Esta suma se realiza de forma concurrente por varios hilos por lo que al no estar protegida el resultado final de suma es incierto o desconocido.

```
1 package mx.uam.pc.candado;
2
3 class Suma implements Runnable{
4   static final int N=200000;
5   static int suma=0;
6   private String nombre;
7
8   public Suma(String n)
9   {
```

```
10
11      nombre = n;
12    }
13    public void run()
14    {
15      System.out.println("Hilo: "+nombre);
16      for(int i=0;i<N;i++)
17      {
18
19          suma++;
20
21      }
22
23    }
24  }
```

candado/Suma.java

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** la suma se protege por un candado (líneas 22-27) lo cual permite obtener el resultado esperado.

```
1  package mx.uam.pc.candado;
2
3  import java.util.concurrent.locks.ReentrantLock;
4
5  class SumaConcurrente implements Runnable{
6    static final int N=200000;
7    static int suma=0;
8    private String nombre;
9    private ReentrantLock re;
10   public SumaConcurrente(ReentrantLock rt, String n)
11   {
12     re = rt;
13     nombre = n;
14   }
15   public void run()
16   {
17     System.out.println("Hilo: "+nombre);
18     for(int i=0;i<N;i++)
19     {
20
21
22       re.lock();
23
24       suma++;
25
26
27       re.unlock();
28
29
30     }
31
32   }
33 }
```

candado/SumaConcurrente.java

La clase Principal realiza la creación de los hilos (líneas 14-16) y muestra el resultado sin sincronización (línea 29) y con sincronización (línea 49).

```
1  package mx.uam.pc.candado;
2
3  import java.util.concurrent.locks.ReentrantLock;
4
5  public class Principal{
6    static final int MAX_T = 4;
7    public static void main(String[] args)
8    {
9      ReentrantLock rel = new ReentrantLock();
10
11     Thread[] t = new Thread[MAX_T];
12
13
14     for(int i=0;i<MAX_T;i++) {
15       t[i]=  new Thread(new Suma("T"+i));
16       t[i].start();
17     }
18
19     for(int i=0;i<MAX_T;i++){
20
21       try {
22         t[i].join();
23
24       } catch (InterruptedException e) {
25
26         System.out.println("Error: en la espera del hilo");
27       }
28     }
29     System.out.println("Resultado final sin sincronizacion:"+Suma.suma)
       ;
30
31
32     for(int i=0;i<MAX_T;i++) {
33
34       t[i]=  new Thread(new SumaConcurrente(rel, "T"+i));
35       t[i].start();
36     }
37
38     for(int i=0;i<MAX_T;i++){
39
40       try {
41         t[i].join();
42
43       } catch (InterruptedException e) {
44
45         System.out.println("Error: en la espera del hilo");
46       }
47     }
48
49     System.out.println("Resultado final con sincronizacion:"+
       SumaConcurrente.suma);
50
51
52
53   }
54 }
```

candado/Principal.java

2.2. Compilar el paquete desde la terminal

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
```

2.3. Candados no bloqueantes

Este ejemplo es similar al anterior solo que se utilizan candados no bloqueantes. En la clase SumaConcurrente.java en la línea 20 se puede observar el uso del candado.

```
1  package mx.uam.pc.spinlock;
2
3  import java.util.concurrent.locks.ReentrantLock;
4
5  class SumaConcurrente implements Runnable{
6    static final int N=200000;
7    static int suma=0;
8    private String nombre;
9    private ReentrantLock re;
10   public SumaConcurrente(ReentrantLock rt, String n)
11   {
12     re = rt;
13     nombre = n;
14   }
15   public void run()
16   {
17     System.out.println("Hilo: "+nombre);
18     for(int i=0;i<N;i++)
19     {
20       while(!re.tryLock()) {}
21
22
23         suma++;
24
25       re.unlock();
26
27
28     }
29
30   }
31 }
```

spinlock/SumaConcurrente.java

```
1  package mx.uam.pc.spinlock;
2
3  import java.util.concurrent.locks.ReentrantLock;
4
5  public class Principal{
6    static final int MAX_T = 4;
7    public static void main(String[] args)
8    {
9      ReentrantLock rel = new ReentrantLock();
10
11     Thread[] t = new Thread[MAX_T];
```

```
12
13
14
15      for(int i=0;i<MAX_T;i++) {
16
17        t[i]=  new Thread(new SumaConcurrente(rel, "T"+i));
18        t[i].start();
19      }
20
21      for(int i=0;i<MAX_T;i++){
22
23        try {
24          t[i].join();
25
26        } catch (InterruptedException e) {
27
28          System.out.println("Error: en la espera del hilo");
29        }
30      }
31
32      System.out.println("Resultado final con sincronizacion (trylock):"+
        SumaConcurrente.suma);
33
34
35
36    }
37  }
```

spinlock/Principal.java

2.4. Semáforos

La nueva clase **SumaConcurrente.java** utiliza un semáforo para proteger la escritura en la variable suma (líneas 20-32). Este ejemplo es solo para fines académicos, si bien el semáforo se puede utilizar para exclusión mutua, no es recomendado dado que su implementación es más costosa que la de un candado y conlleva mayor tiempo de CPU.

```
1  package mx.uam.pc.semaforo;
2
3  import java.util.concurrent.Semaphore;
4
5  class SumaConcurrente implements Runnable{
6    static final int N=200000;
7    static int suma=0;
8    private String nombre;
9    private Semaphore s=null;
10   public SumaConcurrente( Semaphore s, String n)
11   {
12     this.s = s;
13     nombre = n;
14   }
15   public void run()
16   {
17     System.out.println("Hilo: "+nombre);
18     for(int i=0;i<N;i++)
19     {
20       try {
21         s.acquire();//Wait
22       } catch (InterruptedException e) {
```

```
23
24          e.printStackTrace();
25        }
26
27
28          suma++;
29
30
31        s.release();//Signal
32
33
34      }
35
36    }
37  }
```

semaforo/SumaConcurrente.java

En la clase **Principal.java** se crea el semáforo (línea 10) y se crean los hilos pasándoles como parámetro el semáforo que se va a utilizar (líneas 15-19).

```
1  package mx.uam.pc.semaforo;
2
3  import java.util.concurrent.Semaphore;
4
5
6  public class Principal{
7    static final int MAX_T = 4;
8    public static void main(String[] args)
9    {
10     Semaphore s = new Semaphore(1);
11
12     Thread[] t = new Thread[MAX_T];
13
14
15     for(int i=0;i<MAX_T;i++) {
16
17       t[i]=  new Thread(new SumaConcurrente(s, "T"+i));
18       t[i].start();
19     }
20
21     for(int i=0;i<MAX_T;i++){
22
23       try {
24         t[i].join();
25
26       } catch (InterruptedException e) {
27
28         System.out.println("Error: en la espera del hilo");
29       }
30     }
31
32     System.out.println("Resultado final con sincronizacion (semaforo):"
       +SumaConcurrente.suma);
33
34
35
36   }
37 }
```

semaforo/Principal.java

2.5. Berrera

La nueva clase **SumaConcurrente.java** utiliza una barrera para que todos los hilos muestren el resultado de la suma (línea 36).

```
1  package mx.uam.pc.barrera;
2
3  import java.util.concurrent.BrokenBarrierException;
4  import java.util.concurrent.CyclicBarrier;
5  import java.util.concurrent.locks.ReentrantLock;
6
7  class SumaConcurrente implements Runnable{
8    static final int N=200000;
9    static int suma=0;
10   private String nombre;
11   private ReentrantLock re;
12   private CyclicBarrier barrera;
13   public SumaConcurrente(ReentrantLock rt,CyclicBarrier barrera,String
       n)
14   {
15     re = rt;
16     this.barrera=barrera;
17     nombre = n;
18   }
19   public void run()
20   {
21     System.out.println("Hilo: "+nombre);
22     for(int i=0;i<N;i++)
23     {
24       re.lock();
25
26
27         suma++;
28
29
30
31       re.unlock();
32
33
34     }
35     try {
36       barrera.await();
37     } catch (InterruptedException | BrokenBarrierException e) {
38       // TODO Auto-generated catch block
39       e.printStackTrace();
40     }
41     System.out.println("Resultado final con sincronizacion (barrera):"+
       suma);
42
43   }
44 }
```

barrera/SumaConcurrente.java

En la clase **Principal.java** se crea la barrera (línea 11) y se crean los hilos pasándoles como parámetro la barrera que se va a utilizar (líneas 17 y 18).

```
1  package mx.uam.pc.barrera;
2
3  import java.util.concurrent.CyclicBarrier;
4  import java.util.concurrent.locks.ReentrantLock;
5
6  public class Principal{
```

```
7   static final int MAX_T = 4;
8   public static void main(String[] args)
9   {
10    ReentrantLock rel = new ReentrantLock();
11    CyclicBarrier barrera = new CyclicBarrier(MAX_T);
12    Thread[] t = new Thread[MAX_T];
13
14
15    for(int i=0;i<MAX_T;i++) {
16
17      t[i]=  new Thread(new SumaConcurrente(rel,barrera,"T"+i));
18      t[i].start();
19    }
20
21    for(int i=0;i<MAX_T;i++){
22
23      try {
24        t[i].join();
25
26      } catch (InterruptedException e) {
27
28        System.out.println("Error: en la espera del hilo");
29      }
30    }
31
32
33
34
35  }
36 }
```

barrera/Principal.java

2.6. Operaciones atómicas

Si el recurso crítico es una variable y la operación es muy simple se puede evitar un mecanismo de sincronización y usar una operación atómica. En este ejemplo en el programa Principal.java se define un objeto que permite realizar operaciones atómicas (línea 9). La operación atómica se utiliza en la clase SumaConcurrente.java (línea 20).

```
1  package mx.uam.pc.atomic;
2
3  import java.util.concurrent.atomic.AtomicInteger;
4
5  class SumaConcurrente implements Runnable{
6    static final int N=200000;
7    private String nombre;
8    private AtomicInteger atomic;
9    public SumaConcurrente(AtomicInteger atomic, String n)
10   {
11     this.atomic = atomic;
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          atomic.getAndIncrement();
21
22
23
24      }
25
26    }
27  }
```

atomic/SumaConcurrente.java

```
1  package mx.uam.pc.atomic;
2
3  import java.util.concurrent.atomic.AtomicInteger;
4
5  public class Principal{
6    static final int MAX_T = 4;
7    public static void main(String[] args)
8    {
9      AtomicInteger atomic = new AtomicInteger(0);
10
11     Thread[] t = new Thread[MAX_T];
12
13
14     for(int i=0;i<MAX_T;i++) {
15
16       t[i]=  new Thread(new SumaConcurrente(atomic, "T"+i));
17       t[i].start();
18     }
19
20     for(int i=0;i<MAX_T;i++){
21
22       try {
23         t[i].join();
24
25       } catch (InterruptedException e) {
26
27         System.out.println("Error: en la espera del hilo");
28       }
29     }
30
31     System.out.println("Resultado final con sincronizacion:"+atomic.get
       ());
32
33
34
35   }
36 }
```

atomic/Principal.java

3. Ejericicios

- Implemente el problema P productores - C consumidor
- Implemente el problema P productores - c consumidor con un almacén de tamaño N
- Implemente el barbero dormilón