Aplicaciones para Ambientes Distribuidos

Laboratorio # 3 – Prof. Regis Rivera

Objetivo: Crear aplicaciones que apliquen sincronización en Java, con el fin de poner en practica programación concurrente (parte II)
Nota: Los nombres de archivos de cada aplicación están dados en el icono

Bloque sincronizado

Si declaramos un bloque como sincronizado, solo el código que está escrito dentro de ese bloque se ejecuta secuencialmente, no el código completo. Esto se usa cuando queremos acceso secuencial a alguna parte del código o para sincronizar alguna parte del código.

```
synchronized (object reference)
{
    // Insert code here
}
```

Laboratorio 3.1 Implementación de un bloque sincronizado en Java

Lab31.java

```
1  class SynchroTest {
2      public static void main(String[] args)
3      {
4
5          PrintTest p = new PrintTest();
6
7          Thread1 t1 = new Thread1(p);
8          Thread2 t2 = new Thread2(p);
9
10         t1.start();
11         t2.start();
12     }
13 }
14
15 class PrintTest extends Thread {
16
17     public void printThread(int n)
18     {
19
20         synchronized (this)
21         {
22
23             for (int i = 1; i <= 10; i++) {
24
25
26                 System.out.println("Thread " + n
27                                    + " esta trabajando...");
28
29                 try {
30
31                     Thread.sleep(600);
32                 }
33
34                 catch (Exception ex) {
35
36                     System.out.println(ex.toString());
37                 }
38             }
39         }
```

```
40
41          System.out.println("--------------------------");
42
43          try {
44              Thread.sleep(1000);
45          }
46
47          catch (Exception ex) {
48
49              System.out.println(ex.toString());
50          }
51      }
52  }
53
54  class Thread1 extends Thread {
55
56      PrintTest test;
57      Thread1(PrintTest p) { test = p; }
58
59      public void run() { test.printThread(1); }
60  }
61
62  class Thread2 extends Thread {
63
64      PrintTest test;
65      Thread2(PrintTest p) { test = p; }
66
67      public void run() { test.printThread(2); }
68  }
```

Sincronización estática

En esto, el método sincronizado se declara como "estático", lo que significa que el bloqueo o monitor se aplica en la clase, no en el objeto, de modo que solo un subproceso accederá a la clase a la vez.

Lab32.java

Laboratorio 3.2 Implementación de sincronización estática en Java

```
1   class SynchroTest {
2       public static void main(String[] args)
3       {
4           Thread1 t1 = new Thread1();
5           Thread2 t2 = new Thread2();
6
7           t1.start();
8           t2.start();
9       }
10  }
11
12  class PrintTest extends Thread {
13
14      synchronized public static void printThread(int n)
15      {
16          for (int i = 1; i <= 10; i++) {
17
18              System.out.println("Thread " + n
19                                 + " esta trabajando...");
20
21              try {
22                  Thread.sleep(600);
23              }
24
25              catch (Exception ex) {
26
27                  System.out.println(ex.toString());
28              }
29          }
30
31          System.out.println("--------------------------");
```

```
32
33          try {
34              Thread.sleep(1000);
35          }
36
37          catch (Exception ex) {
38              System.out.println(ex.toString());
39          }
40      }
41  }
42
43  class Thread1 extends Thread {
44
45      public void run()
46      {
47
48          PrintTest.printThread(1);
49      }
50  }
51
52  class Thread2 extends Thread {
53
54      public void run()
55      {
56
57          PrintTest.printThread(2);
58      }
59  }
```

Sincronización de métodos y bloques en Java

Los subprocesos se comunican principalmente compartiendo el acceso a los campos y a los objetos a los que hacen referencia los campos. Esta forma de comunicación es extremadamente eficiente, pero hace posibles dos tipos de errores: interferencia de subprocesos y errores de consistencia de memoria. Se necesitan algunas construcciones de sincronización para evitar estos errores. El siguiente ejemplo muestra una situación en la que necesitamos sincronización.

Laboratorio 3.3 Ilustrar la necesidad de sincronización (en este caso, en Java)

Lab33.java

```
1   import java.io.*;
2
3   class GfG
4   {
5       public static void main (String[] args)
6       {
7           Multithread t = new Multithread();
8           t.increment();
9           System.out.println(t.getValue());
10      }
11  }
12
13  class Multithread
14  {
15      private int i = 0;
16      public void increment()
17      {
18          i++;
19      }
20
21      public int getValue()
22      {
23          return i;
24      }
25  }
```

En el laboratorio 3.3 se realizan tres operaciones:

1. Obtenga el valor de la variable i.
2. Incremente el valor recuperado.
3. Y almacene el valor aumentado de i en su ubicación.

Aquí

- El primer hilo obtiene el valor de i. (Actualmente el valor i es 0) y lo aumenta en uno, por lo que el valor de la variable i se convierte en 1.
- Ahora el 2º hilo accede al valor de i que sería 0 ya que el 1º hilo no lo almacenó de nuevo en su ubicación.
- Y el 2º hilo también lo incrementa y guárdalo de nuevo en su ubicación. Y 1º también lo almacena.
- Finalmente, el valor de la variable i es 1. Pero debería ser 2 por el efecto de ambos hilos. Es por eso que necesitamos sincronizar el acceso a la variable compartida i.

Java es un lenguaje multihilo en el que varios hilos se ejecutan en paralelo para completar su ejecución. Necesitamos sincronizar los recursos compartidos para asegurarnos de que a la vez solo un subproceso pueda acceder al recurso compartido.

Si un objeto es compartido por varios subprocesos, entonces es necesario sincronizarlo para evitar que el estado del objeto se corrompa. La sincronización es necesaria cuando Object es mutable. Si el objeto compartido es inmutable o todos los subprocesos que comparten el mismo objeto solo leen el estado del objeto, no se modifican, entonces no es necesario sincronizarlo.

El lenguaje de programación Java proporciona dos modismos de sincronización:

- Sincronización de métodos
- Sincronización de sentencias (sincronización de bloques)

Sincronización de métodos

Los métodos sincronizados permiten una estrategia sencilla para evitar la interferencia de subprocesos y los errores de coherencia de memoria. Si un objeto es visible para más de un subproceso, todas las lecturas o escrituras en los campos de ese objeto se realizan a través del método sincronizado.

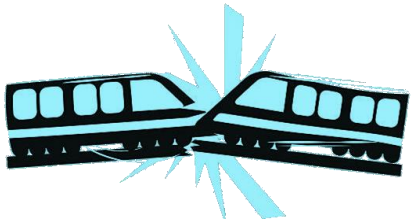
No es posible que se intercalen dos invocaciones para métodos sincronizados. Si un subproceso está ejecutando el método sincronizado, todos los demás subprocesos que invoquen el método sincronizado en el mismo objeto tendrán que esperar hasta que el primer subproceso termine con el objeto.

Laboratorio 3.4 Mostrar si más de un subproceso accede al método getLine() sin sincronización.

Lab34.java

```java
import java.io.*;

class GFG
{
    public static void main(String[] args)
    {
        Line obj = new Line();

        Train train1 = new Train(obj);
        Train train2 = new Train(obj);

        train1.start();
        train2.start();
    }
}

class Line
{
    public void getLine()
    {
        for (int i = 0; i < 3; i++)
        {
            System.out.println(i);
            try
            {
                Thread.sleep(400);
            }
            catch (Exception e)
            {
                System.out.println(e);
            }
        }
    }
}

class Train extends Thread
{
    Line line;

    Train(Line line)
    {
        this.line = line;
    }

    @Override
    public void run()
    {
        line.getLine();
    }
}
```

Puede haber dos trenes (más de dos) que deban circular al mismo tiempo, por lo que existe la posibilidad de colisión. Por lo tanto, para evitar colisiones, necesitamos sincronizar la línea en la que quieren correr varios elementos.



Laboratorio 3.5 Acceso sincronizado al método getLine() en el mismo objeto

Lab35.java

```java
class GFG
{
    public static void main(String[] args)
    {
        Line obj = new Line();

        Train train1 = new Train(obj);
        Train train2 = new Train(obj);

        train1.start();
        train2.start();
    }
}

class Line
{

    synchronized public void getLine()
    {
        for (int i = 0; i < 3; i++)
        {
            System.out.println(i);
            try
            {
                Thread.sleep(400);
            }
            catch (Exception e)
            {
                System.out.println(e);
            }
        }
    }
}

class Train extends Thread
{
    Line line;

    Train(Line line)
    {
        this.line = line;
    }

    @Override
    public void run()
    {
        line.getLine();
    }
}
```

Sincronización de bloques

Si solo necesitamos ejecutar algunas líneas de código subsecuentes, no todas las líneas (instrucciones) de código dentro de un método, entonces debemos sincronizar solo el bloque del código dentro del cual existen las instrucciones requeridas.

Por ejemplo, supongamos que hay un método que contiene 100 líneas de código, pero solo hay 10 líneas (una tras otra) de código que contienen una sección crítica de código, es decir, estas líneas pueden modificar (cambiar) el estado del objeto. Por lo tanto, solo necesitamos sincronizar estas

10 líneas de método de código para evitar cualquier modificación en el estado del objeto y para asegurarnos de que otros subprocesos puedan ejecutar el resto de las líneas dentro del mismo método sin ninguna interrupción.

Laboratorio 3.6 Sincronización por bloques en Java

Lab36.java

```java
import java.io.*;
import java.util.*;

class GFG
{
    public static void main (String[] args)
    {
        Geek gk = new Geek();
        List<String> list = new ArrayList<String>();
        gk.geekName("Texto Geek", list);
        System.out.println(gk.name);
    }
}

class Geek
{
    String name = "";
    public int count = 0;

    public void geekName(String geek, List<String> list)
    {
        synchronized(this)
        {
            name = geek;
            count++;
        }

        list.add(geek);
    }
}
```

Puntos importantes:

- Cuando un subproceso entra en un método o bloque sincronizado, adquiere el bloqueo y, una vez que completa su tarea y sale del método sincronizado, libera el bloqueo.
- Cuando el subproceso entra en el método o bloque de instancia sincronizado, adquiere el bloqueo de nivel de objeto y cuando entra en el método o bloque estático sincronizado, adquiere el bloqueo de nivel de clase.
- La sincronización de Java producirá una excepción de puntero nulo si el objeto utilizado en el bloque sincronizado es nulo. Por ejemplo, si en synchronized(instance) , instance es null, lanzará una excepción de puntero nulo.
- En Java, wait(), notify() y notifyAll() son los métodos importantes que se utilizan en la sincronización.
- No se puede aplicar la palabra clave java synchronized con las variables.
- No sincronice en el campo no final en el bloque sincronizado porque la referencia al campo no final puede cambiar en cualquier momento y, a continuación, es posible que diferentes subprocesos se sincronicen en diferentes objetos, es decir, que no haya sincronización en absoluto.

Ventajas

- Subprocesos múltiples: Dado que Java es un lenguaje multiproceso, la sincronización es una buena manera de lograr la exclusión mutua en recursos compartidos.
- Métodos estáticos y de instancia: Tanto los métodos de instancia sincronizados como los métodos estáticos sincronizados se pueden ejecutar simultáneamente porque se utilizan para bloquear diferentes objetos.

Limitaciones

- Limitaciones de simultaneidad: La sincronización de Java no permite lecturas simultáneas.
- Disminuye la eficiencia: El método sincronizado de Java se ejecuta muy lentamente y puede degradar el rendimiento, por lo que debe sincronizar el método cuando sea absolutamente necesario, de lo contrario no y sincronizar el bloque solo para la sección crítica del código.