Aplicaciones para Ambientes Distribuidos

Laboratorio # 2 – Prof. Regis Rivera

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

Sincronización en Java

Los programas multiproceso a menudo pueden llegar a una situación en la que varios subprocesos intentan acceder a los mismos recursos y finalmente producen resultados erróneos e imprevistos.

¿Por qué utilizar Java Synchronization?
La sincronización de Java se utiliza para asegurarse de que, mediante algún método de sincronización, solo un subproceso pueda acceder al recurso en un momento dado.

Bloques sincronizados de Java
Java proporciona una forma de crear subprocesos y sincronizar sus tareas mediante bloques sincronizados.

Un bloque sincronizado en Java se sincroniza en algún objeto. Todos los bloques sincronizados se sincronizan en el mismo objeto y solo pueden tener un subproceso ejecutado dentro de ellos a la vez. Todos los demás subprocesos que intentan entrar en el bloque sincronizado se bloquean hasta que el subproceso dentro del bloque sincronizado sale del bloque.

```
synchronized(sync_object)
{
  // Access shared variables and other
  // shared resources
}
```

Esta sincronización se implementa en Java con un concepto llamado monitores o bloqueos. Solo un subproceso puede poseer un monitor en un momento dado. Cuando un hilo adquiere un bloqueo, se dice que ha entrado en el monitor. Todos los demás subprocesos que intenten entrar en el monitor bloqueado se suspenderán hasta que el primer subproceso salga del monitor.

Tipos de sincronización
Hay dos sincronizaciones en Java que se mencionan a continuación:

1. Sincronización de procesos
    a. La sincronización de procesos es una técnica utilizada para coordinar la ejecución de múltiples procesos. Garantiza que los recursos compartidos estén seguros y en orden.
2. Sincronización de subprocesos

a. La sincronización de subprocesos se utiliza para coordinar y ordenar la ejecución de los subprocesos en un programa multihilo. A continuación se mencionan dos tipos de sincronización de subprocesos:
    i. Exclusivo mutuo
    ii. Cooperación (comunicación entre subprocesos en Java)

Exclusivo mutuo

Mutual Exclusive ayuda a evitar que los hilos interfieran entre sí mientras comparten datos. Hay tres tipos de Mutual Exclusive que se mencionan a continuación:

- Método sincronizado.
- Bloque sincronizado.
- Sincronización estática.

Laboratorio 2.1 Implementación de la sincronización de Java

Lab21.java

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

class SyncDemo {
    public static void main(String args[])
    {
        Sender send = new Sender();
        ThreadedSend S1 = new ThreadedSend(" Hola ", send);
        ThreadedSend S2 = new ThreadedSend(" Adios ", send);

        S1.start();
        S2.start();

        try {
            S1.join();
            S2.join();
        }
        catch (Exception e) {
            System.out.println("Interrumpido");
        }
    }
}

class ThreadedSend extends Thread {
    private String msg;
    Sender sender;

    ThreadedSend(String m, Sender obj)
    {
        msg = m;
        sender = obj;
    }

    public void run()
    {
        synchronized (sender)
        {
            sender.send(msg);
        }
    }
}
```

```
42
43  class Sender {
44      public void send(String msg)
45      {
46          System.out.println("Enviando\t" + msg);
47          try {
48              Thread.sleep(1000);
49          }
50          catch (Exception e) {
51              System.out.println("Hilo interrumpido");
52          }
53          System.out.println("\n" + msg + "Enviado");
54      }
55  }
56
```

La salida es la misma cada vez que ejecutamos el programa.

Explicación

En el ejemplo anterior, elegimos sincronizar el objeto Sender dentro del método run() de la clase ThreadedSend. Alternativamente, podríamos definir todo el bloque send() como sincronizado, produciendo el mismo resultado. Entonces no tenemos que sincronizar el objeto Message dentro del método run() en la clase ThreadedSend.

No siempre tenemos que sincronizar un método completo. A veces es preferible sincronizar solo una parte de un método. Los bloques sincronizados de Java dentro de los métodos hacen esto posible.

Laboratorio 2.2 Método sincronizado mediante una clase anónima

Lab22.java

```
1   import java.io.*;
2
3
4   public class GFG {
5
6       public static void main(String args[])
7       {
8
9           final Test obj = new Test();
10
11          Thread a = new Thread() {
12              public void run() { obj.test_function(15); }
13          };
14
15          Thread b = new Thread() {
16              public void run() { obj.test_function(30); }
17          };
18
19          a.start();
20          b.start();
21      }
22  }
23
24
25  class Test {
26      synchronized void test_function(int n)
27      {
28
29          for (int i = 1; i <= 3; i++) {
30              System.out.println(n + i);
31              try {
32                  Thread.sleep(500);
33              }
34              catch (Exception e) {
35                  System.out.println(e);
36              }
37          }
38      }
39  }
```

Importancia de la sincronización de subprocesos en Java

Nuestros sistemas funcionan en un entorno multihilo que se convierte en una parte importante para que el sistema operativo proporcione una mejor utilización de los recursos. El proceso de ejecutar dos o más partes del programa simultáneamente se conoce como Multithreading. Un programa es un conjunto de instrucciones en el que se ejecutan varios procesos y, dentro de un proceso, funcionan varios subprocesos. Los subprocesos no son más que procesos ligeros. Por ejemplo, en el ordenador estamos jugando a videojuegos al mismo tiempo que trabajamos con MS Word y escuchamos música. Entonces, estos son los procesos en los que estamos trabajando al mismo tiempo. En esto, cada aplicación tiene múltiples subprocesos, es decir, subprocesos. En el ejemplo anterior, escuchamos música en la que tenemos un reproductor de música como una aplicación que contiene múltiples subprocesos que se están ejecutando, como administrar listas de reproducción, acceder a Internet, etc. Por lo tanto, los hilos son la tarea a realizar y el multithreading son múltiples tareas/procesos que se ejecutan al mismo tiempo en el mismo programa.

Prioridades de subprocesos

En Java, a cada subproceso se le asigna una prioridad que determina cómo se deben tratar los subprocesos entre sí. La prioridad del subproceso se usa para decidir cuándo cambiar de un subproceso en ejecución al siguiente. Un subproceso de prioridad más alta puede adelantarse a un subproceso de prioridad más baja y puede tardar más tiempo de CPU. De una manera sencilla, un subproceso con mayor prioridad obtiene el recurso primero en comparación con el subproceso con menor prioridad. Pero, en caso de que dos subprocesos con la misma prioridad quieran el mismo recurso, la situación se complica. Por lo tanto, en un entorno multithreading, si los subprocesos con la misma prioridad están trabajando con el mismo recurso, dan resultados no deseados o código erróneo.

Pongamos un ejemplo. En una sala, tenemos varias computadoras que están conectadas a una sola impresora. En un momento dado, una computadora quiere imprimir un documento, por lo que usa una impresora. Al mismo tiempo, otro ordenador quiere que la impresora imprima su documento. Por lo tanto, dos computadoras exigen el mismo recurso, es decir, una impresora. Por lo tanto, si ambos procesos se ejecutan juntos, la impresora imprimirá el documento de una computadora y de otra. Esto producirá una salida no válida. Ahora, lo mismo sucede en el caso de los subprocesos si dos subprocesos con la misma prioridad o quieren el mismo recurso conducen a una salida inconsistente.

En Java, cuando dos o más hilos intentan acceder al mismo recurso simultáneamente, hace que el tiempo de ejecución de Java ejecute uno o más hilos lentamente, o incluso suspenda su ejecución. Para superar este problema, tenemos sincronización de hilos.

La sincronización significa coordinación entre múltiples procesos/subprocesos.

Tipos de sincronización:

Hay dos tipos de sincronización que son los siguientes:

- Sincronización de procesos
- Sincronización de subprocesos

Aquí nos centraremos principalmente en la sincronización de hilos.

La sincronización de subprocesos se refiere básicamente al concepto de un subproceso que se ejecuta a la vez y el resto de los subprocesos están en estado de espera. Este proceso se conoce como sincronización de subprocesos. Evita la interferencia de roscas y el problema de inconsistencia.

La sincronización se construye mediante bloqueos o monitor. En Java, un monitor es un objeto que se utiliza como un bloqueo mutuamente excluyente. Solo un subproceso a la vez tiene derecho a poseer un monitor. Cuando un subproceso obtiene un bloqueo, se suspenderán todos los demás subprocesos que intenten adquirir el monitor bloqueado. Por lo tanto, se dice que otros subprocesos están esperando el monitor, hasta que el primer subproceso sale del monitor. De una manera sencilla, cuando un subproceso solicita un recurso, ese recurso se bloquea para que ningún otro subproceso pueda funcionar o realizar ninguna modificación hasta que se libere el recurso.

La sincronización de subprocesos es de dos tipos:

1. Exclusivo mutuo
2. Comunicación entre subprocesos

A. Exclusividad mutua

Al compartir cualquier recurso, esto mantendrá el hilo interfiriendo entre sí, es decir, excluyéndose mutuamente. Podemos lograrlo a través de

- Método sincronizado
- Bloque sincronizado
- Sincronización estática

Método sincronizado

Podemos declarar un método como sincronizado usando la palabra clave "synchronized". Esto hará que el código escrito dentro del método sea seguro para subprocesos, de modo que ningún otro subproceso se ejecutará mientras se comparte el recurso.

Laboratorio 2.3 Propondremos impresiones de los dos hilos simultáneamente mostrando el comportamiento asíncrono sin sincronización de hilos.

Lab23.java

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

public class SynchroTest {

    public static void main(String[] args)
    {
        PrintTest p = new PrintTest();

        Thread1 t1 = new Thread1(p);
        Thread2 t2 = new Thread2(p);

        t1.start();
        t2.start();

    }
}
```

```
18
19  class PrintTest extends Thread {
20      public void printThread(int n)
21      {
22          for (int i = 1; i <= 10; i++) {
23              System.out.println("Thread " + n
24                                      + " esta trabajando...");
25              try {
26                  Thread.sleep(600);
27              }
28              catch (Exception ex) {
29                  System.out.println(ex.toString());
30              }
31          }
32          System.out.println("--------------------------");
33
34          try {
35              Thread.sleep(1000);
36          }
37
38          catch (Exception ex) {
39
40              System.out.println(ex.toString());
41          }
42      }
43  }
44
45
46  class Thread1 extends Thread {
47
48      PrintTest test;
49
50      Thread1(PrintTest p) { test = p; }
51
52      public void run()
53      {
54          test.printThread(1);
55      }
56  }
59  class Thread2 extends Thread {
60      PrintTest test;
61
62      Thread2(PrintTest p) { test = p; }
63      public void run() { test.printThread(2); }
64  }
```

Laboratorio 2.4 Veamos ahora otro ejemplo de acuerdo al tema

Lab24.java

```java
class SynchroTest {

    public static void main(String[] args)
    {

        PrintTest p = new PrintTest();

        Thread1 t1 = new Thread1(p);
        Thread2 t2 = new Thread2(p);

        t1.start();
        t2.start();
    }
}

class PrintTest extends Thread {

    synchronized public void printThread(int n)
    {
        for (int i = 1; i <= 10; i++) {
            System.out.println("Thread " + n
                               + " esta trabajando...");

            try {

                Thread.sleep(600);
            }
            catch (Exception ex) {
                System.out.println(ex.toString());
            }
        }
        System.out.println("--------------------------");
        try {

            Thread.sleep(1000);
        }
        catch (Exception ex) {
            System.out.println(ex.toString());
        }
    }
}
```

```
42
43  class Thread1 extends Thread {
44
45      PrintTest test;
46      Thread1(PrintTest p) { test = p; }
47
48      public void run()
49      {
50
51          test.printThread(1);
52      }
53  }
54
55  class Thread2 extends Thread {
56
57      PrintTest test;
58      Thread2(PrintTest p) { test = p; }
59      public void run()
60      {
61          test.printThread(2);
62      }
63  }
64
```

En el siguiente laboratorio veremos bloque sincronizado