1. Introducción

Aunque aquí se presenta un breve resumen teórico **Es totalmente recomendable la lectura completa de los "Apuntes sobre concurrencia en Java" que se encuentran en campus virtual.**

En esta práctica veremos, como implementaríamos en java mecanismos de sincronización manuales (espera ocupada) como complemento al tema 2 de la asignatura.

Pero, como sabes la espera activa no es razonablemente utilizable por su consumo innecesario de recursos, por eso se presenta sólo a título de ejercicio

Desde la versión 2 Java incluyó los primeros mecanismos de sincronización del lenguaje (sin espera activa) que empezaremos a utilizar para comprobar cómo resolveríamos los mismos problemas con ambos instrumentos:

En Java todos los objetos tienen un bloqueo asociado, lock o cerrojo, que puede ser adquirido y liberado mediante el uso de métodos y sentencias synchronized lo que nos permite el acceso en exclusión mutua a un método o una parte de código

INSTRUMENTOS BÁSICOS PARA GESTIÓN DE CONCURRENCIA EN JAVA

Mecanismos de bloqueo (regiones críticas y exclusión mutua):

- **Métodos synchronized.**
- **Bloques synchronized .**

Sincronización de atributos modificados por varios threads:

- **Uso de volatile**

Mecanismos de sincronización y comunicación de los threads :

- **wait(), notify(),notifyAll()...**

http://www.herongyang.com/Java/Synchronization-Support-in-Java-synchronized.html
http://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html

CERROJOS

Todos los objetos tienen asociado un cerrojo, que nos permite bloquear el acceso al objeto, Cuando un hilo trata de obtener un cerrojo que está bloqueado, éste quedará suspendido hasta que el cerrojo sea liberado. Cada cerrojo tiene, por lo tanto, una lista de hilos suspendidos a la espera de que se libere. Cuando esto ocurre, cualquiera de ellos es despertado, y se le cede el cerrojo

EXCLUSIÓN MUTUA

Los cerrojos pueden utilizarse para garantizar exclusión mutua, es decir para que los hilos accedan a los recursos de forma controlada, de modo que solo un hilo sea dueño de un recurso en un determinado momento

El lenguaje Java permite al programador indicar la ejecución en exclusión mutua de partes del código, gracias a esto, se permite un control sincronizado de acceso a recursos (por ejemplo variables), Java proporciona dos modos principales de sincronización para conseguir la exclusión mutua. La primera son los métodos sincronizados y la segunda los bloques sincronizados.

En la implementación de una clase, pueden especificarse algunos (o todos) los métodos como ***synchronized*** para conseguir el acceso en exclusión mutua a los mismos. . Cuando un hilo realiza una llamada a un método synchronized , antes de que se comience a ejecutar el código del método, el hilo debe conseguir adquirir el cerrojo asociado con el objeto this que se está utilizando. Gracias a eso, solo un hilo puede estar ejecutando el código de ese método. Más aún, solo un hilo puede estar ejecutando alguno de todos **los métodos sincronizados** de un objeto, pues todos, antes de comenzar a ejecutarse, deben obtener el mismo cerrojo. Hay que destacar que esto es a nivel de *objetos*, pues cada uno tendrá un cerrojo. Dos hilos podrán estar ejecutando el mismo método sincronizado al mismo tiempo si son de objetos diferentes (aunque de la misma clase).

Synchronized también puede utilizarse con una clase estatica (static).

SEÑALIZACIÓN

Los objetos en Java también disponen de unos mecanismos simples para la comunicación entre hilos.Para ello, todos los objetos implementan los métodos wait() y notify(). A grandes rasgos:

Un hilo que llama al método wait() de un cierto objeto queda suspendido hasta que otro hilo llame al método notify() del mismo objeto. Por lo tanto, todos los objetos tienen una lista de hilos que están suspendidos a la espera de la llamada a notify() en ese objeto. El método notify() solo despierta a uno de las hilos suspendidos, y, como con los cerrojos, la máquina virtual no obliga a la existencia de una planificación, por lo que cada implementación tiene libertad para decidir a cual despertar. Si se llama a notify() y no hay ningun hilo suspendido esperando, la llamada no hace nada.

Todas las llamadas al método wait() de un objeto deben estar ejecutadas por una hebra que posea el cerrojo del objeto (ya sea desde dentro de un método o un bloque sincronizado). Además, la ejecución de dicho método se compone de la realización de dos operaciones *de forma atómica*: la liberación del cerrojo del objeto, y la suspensión de la hebra. Ambas deben realizarse seguidas, de forma inseparable. Del mismo modo, todas las hebras que llamen al método notify() deberán también estar en posesión del cerrojo del objeto. La llamada al

método despierta una de las hebras en espera. Ésta no podrá comenzar a ejecutarse directamente, pues necesitará volver a bloquear el cerrojo que había liberado anteriormente. Por lo tanto, la hebra pasará de estar suspendida a la espera de un notify()a estar suspendida a la espera de conseguir el cerrojo del objeto, que tendrá bloqueado en ese momento la hebra que ha llamado al notify().

2. Ejercicios

1 Sincronización, pequeño Productor/Consumidor.

Tenemos una Clase **`Cola`** que implementa una *"cola"* (realmente es una cola sin buffer que sólo almacena un elemento entero en una variable compartida **`n`** pero que pretendemos nos sirva para sincronizar dos procesos Productor y Consumidor. La clase tiene dos métodos: **`get()`** { obtiene un elemento} y **`put()`** {pone un elemento, que genera en una secuencia creciente}. Esta cola es el recurso cuyo acceso tratamos de sincronizar.

Realmente los métodos sólo nos mostraran por pantalla un mensaje simulando que ponen u obtienen un elemento para el hilo que los ejecute. Para hacer más apreciable su funcionamiento los métodos simulan que tardan un determinado tiempo en realizar el proceso durmiendo el hilo un tiempo aleatorio.

Importante: La sincronización nunca debe depender de estos retardos, el productor y el consumidor deben estar sincronizados independientemente de su velocidad.

Como hemos dicho tenemos también otras dos clases **`Productor y Consumidor`**, clases que implementan Runnable. .**`Productor`**: llama repetidamente a **`put`** para que genere elementos.. **`Consumidor`**: análogo al productor pero llama a **`get`** para que obtenga los elementos. Ambos nos mostraran por pantalla información del elemento que ponen/cogen.

Tareas a realizar:

1.- Crea la clase **`main`** donde creamos la cola así como un hilo productor y otro consumidor

Comprueba su funcionamiento …. Tenemos un problema de sincronización:

- El productor y el consumidor no están sincronizados.

Modifica el programa creando estas tres versiones:

A. Sincroniza el productor consumidor mediante una variable lógica y una espera activa.

- ¿Es necesario el uso de volatile o synchronized?

B. Realiza la sincronización utilizando los métodos de sincronización de Java **wait, notify, o notifyAll**.

- Si lo consideras necesario Utiliza ***syncronized*** en los métodos get() y put(), Comprueba el funcionamiento y explica qué ha cambiado y cuál es la función de syncronized.
- http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#notify%28%29
-

C. Modifica el programa B para que exista un productor y dos hilos consumidores y el método get() nos indique para qué hilo está obteniendo el dato.

```
class Cola {
    private int n=0;
      int get(){
        try {
            Thread.sleep((long) (Math.random() *500));
        } catch (InterruptedException ex) {
            Logger.getLogger(Cola.class.getName()).log(Level.SEVERE, null, ex);
        }
        System.out.println("Obtengo: " + n);
        return n;

    }
     void put (){
        System.out.print("Generando... ");
        try {
            Thread.sleep((long) (Math.random() *500));
        } catch (InterruptedException ex) {
            Logger.getLogger(Cola.class.getName()).log(Level.SEVERE, null, ex);
        }
        System.out.println("Pongo: "+ ++n);
    }
}
```

```
class Productor implements Runnable{
    Cola q;
    Productor(Cola q){
        this.q=q;
    }
    public void run(){
        int i=0;
        while(true){ //bucle infinito
           q.put();
        }
    }
}

   class Consumidor implements Runnable{
       Cola q;
       Consumidor(Cola q){
           this.q=q;
       }

       public void run(){
           while (true){ //bucle infinito
              q.get();
           }
       }
   }
```

2. El jardín ornamental

En Moodle puedes descargar el código de un pequeño Applet que nos implementa una simulación en entorno gráfico del problema del Jardín Ornamental. No es necesario que te preocupes de los métodos gráficos. Sólo necesitamos centrarnos en las clases Contador y Torno.

Hemos visto que el algoritmo de Dekker nos permite implementar una exclusión mutua mediante espera activa, pero no es el único, hay otros como el de Peterson , Lamport o el de Eisemberg & Mcguire

A. Crea una versión utilizando el Algoritmo de Peterson para proteger la sección crítica y que el contador funcione correctamente

B. Utiliza la herramienta de Java **syncronized**, para una versión sin espera activa.

3. Contesta a las siguientes preguntas:

1.- ¿Qué finalidad tiene el modificador volatile?

2.-¿Para qué sirve synchronized?

3.- ¿Cómo funcionan wait(), notify() y notify?