Programación Concurrente
Programación Concurrente Multihilos - Java
Parte II

Sincronización

Cuando dos o mas hilos necesitan utilizar el mismo recurso se da la posibilidad de que las operaciones se entrelacen provocando inconsistencias en el recurso compartido. Los hilos compitan por el acceso a ese recurso. En este caso hablamos de **"sincronizacion por competencia"**. Si la sincronización se hace de forma correcta, los hilos no realizarán acciones que puedan generar inconsistencias en los datos. Pero, en caso contrario podria darse una "condición de carrera" entre los hilos que intentan acceder al recurso, si este no está "cuidado" de forma apropiada, y generarse las inconsistencias.

¿A que nos referimos con "inconsistencias en los datos"? ¿Cómo puede darse una condición de carrera?

Supongamos el caso de varios hilos que utilizan una estructura de datos de forma compartida, por ejemplo una pila, en la que los hilos pueden apilar elementos, consultar el tope de la pila y desapilar elementos. Especificamente, consideremos una pila de cadenas de caracteres, a la que llamamos PO, y una inversión de la pila, a la que llamamos PI que debe ser llenada por los hilos. Los elementos de PI deben ser cadenas que son una "traslación" 1 lugar hacia adelante (es decir se genera una cadena con los caracteres siguientes alfabeticamente a los de la cadena original) de la cadena que esta en el tope de la pila PO que se desea invertir, en ese momento. Es decir que si en el tope de PO esta la cadena *"hola"*, el hilo que este ejecutándo su run, debera apilar en PI la cadena *"ipmb"*, y si en el tope de la pila PO está la cadena *"abcd"* deberá apilar en PI la cadena *"bcde"* . En este caso una inconsistencia sería que haya 2 cadenas iguales proximas en la pila, o falte alguna cadena. Y esto podria suceder si 2 hilos acceden a consultar el tope de la pila concurrentemente, de forma que los 2 leen el mismo tope

Ejemplo de PO y PI con inconsistencia sería:

PO	PI correcta	PI incorrecta
krod hola fedg abcd	bcde gfed ipmb lspe	bcde ipmb lspe

¿Cómo se llega a esa situación de PI incorrecta? ¿Qué sucedió con la cadena "*gfed*" correspondiente a la traslación de "*fedg*"?

Entonces … recordar que

Los recursos que pueden ser accedidos por varios hilos o procesos que pueden modificar el estado del recurso, son ***recursos críticos,*** y poseen una ***sección crítica*** que es necesario que los hilos traten con ***exclusividad***.

En una siuación de "sincronización por competencia", es fundamental que los hilos que acceden a una ***sección crítica*** lo hagan en ***exclusión mutua***.

y ...

¿Cómo diseñar la sincronización?
Diseñar la sincronización apropiada es difícil. Podriamos pensar en una relación cliente-servidor donde el servidor es el objeto compartido, y los clientes los que utilizan ese recurso compartido. En este escenario se puede dar la responsabilidad de la sincronización al cliente o al servidor.
Si le damos la responsabilidad al cliente, es necesario que TODOS los clientes que utilicen ese recurso compartido se ocupen de considerar las sentencias necesarias para lograr una buenea sincronización, y para que funcione debemos confiar en que todos los clientes de ese recurso se comportarán de forma correcta.
Si le damos la responsabilidad al servidor, o sea al recurso compartido, logramos que el recurso se proteja a si mismo de inconsistencias. Lo puede hacer con métodos o bloques sincronizados o algun otro mecanismo de sincronización. Esto impide que algun cliente utilice el objeto sin sincronización, es decir independiza al cliente de la responsabilidad de contemplar la sincronización.

Mecanismos para lograr la exclusión mutua

Para lograr la exclusión mutua para trabajar sobre la sección crítica, existen diversas posibilidades.

- trabajar con un mecanismo de sincronización básico provisto por el lenguaje
- utilizar semáforos binarios
- otras opciones

Métodos y Bloques sincronizados *(synchronized)*

El mecanismo de sincronización básico provisto por Java son los métodos y bloques sincronizados. Para ello se utiliza la palabra reservada "synchronized" y la característica de que cada objeto tiene una llave de acceso o lock propio.

En este constexto cuando un hilo invoca un método sobre el recurso compartido, que está

sincronizado, lo primero que debe suceder es que obtenga el lock del objeto compartido. Si logra obtener el lock, significa que ningun otro hilo tiene el lock en su poder, es decir que ningun otro hilo esta ejecutando ese método ni ningun otro método sincronizado sobre el mismo objeto. Y además que ningun hilo esta ejecutando un bloque de código que este dentro de un bloque sincronizado sobre el mismo objeto.

Ejemplo: un sistema de reservas para un concierto

Una computadora central conectada a terminales remotas via enlaces de comunicación es usada para hacer reservas de asientos para un concierto. Cada terminal se encuentra en un punto de venta.

Para reservar un asiento en uno de los puntos de venta, un cliente elige un asiento vacío y el empleado entra el numero del asiento elegido en la terminal y genera un ticket, si esta libre.

Como las terminales comparten la tabla de asientos puede ocurrir que mientras el empleado intenta registrar la reserva del asiento solicitado y generar el ticket, el asiento se ocupe.

Se requiere un sistema que evite doble reserva del mismo asiento mientras que permite que los clientes elijan entre los asientos disponibles. Se requiere un sistema que sea seguro, que no genere inconsistencias. Para ello se debe identificar las secciones críticas y accederlas en exclusión mutua.

Podemos considerar el recurso compartido “salaDeConcierto”, y los hilos que representan los empleados que utilizan las terminales para registrar las reservas. No es necesario simular los clientes, el numero de asiento se generará aleatoriamente entre los disponibles.

La sección crítica de salaDeConcierto será el trozo de código en el que se verifique la disponibilidad del asiento y se realice la actualización del mismo.

Primer intento con métodos sincronizados

```
Class SalaDeConcierto {

      int[]  asientosLibresyOcupados;
      //true significa que esta libre.

      public SalaDeConcierto( int tamañoSala) {
            asientosLibresyOcupados = new int[tamañoSala];
            this.iniciarAsientos ();
      }

      private iniciarAsientos () {
            // los inicializa a todos en true
            …
      }

      public boolean hayAsientosLibres() {
            …
      }

      public synchronized boolean verificarAsientoLibre
```

```
                                                (int nroAsiento {
            return (asientosLibresyOcupados [nroAsiento]);
      }

      public synchronized void ocuparAsiento (int nroAsiento){
            asientosLibresyOcupados [nroAsiento] = false;
      }

      …
}
```

y en el hilo empleadoTerminal el método run …

```
Class EmpleadoTerminal implements Runnable(){
      // constructor y métodos auxiliares
      …

      private boolean enHorario() {
            …
      ]

      public void run(){
            int asiento;
            while (this.enHorario() ){
                  asiento = … // se genera aleatoriamente un
                              //nro entre 1 y tamaño de la sala
                  if (salaConc.verificarAsientoLibre (asiento)){
                        sala.OcuparAsiento (asiento)
                  }

      ...
}
```

El problema con esta solución es que aún cuando los métodos están sincronizados y por lo tanto aseguran la exclusión mutua en el acceso al recurso (la colección de asientos, en este caso), pueden ocurrir inconsistencias en los datos, por condiciones de carrera.

Considere que hay varios hilos EmpleadoTerminal (empTer[1], empTer[2], … empTer[K]) accediendo al recurso para registrar reservas. Suponga la siguiente situación:

- empTer[1] verificaAsientoLibre(5) ---> true, //significa que el asiento nro 5 esta libre aún
- epmTer[1] pierde la CPU, pero sabe que el asiento 5 esta libre
- empTer[2] comienza a ejecutarse, verificaAsientoLibre(5) ---> true, (dado que el asiento sigue estando libre), y lo reserva.
- empTer[2] pierde la CPU, pero alcanzo a terminar la reserva del asiento 5

- empTer[1] continua su ejecución con el valor “true” obtenido en el paso previo en su ejecucion, y hace la reserva del asiento 5 (que ahora ya está ocupado).

En este caso se produce una doble reserva, que es lo que se deberia evitar.

Segundo intento con métodos sincronizados

¿Cómo se puede resolver entonces, para controlar esa situación?
La propuesta es hacer la verificación y actualización como algo atómico, es decir considerando que la sección crítica contempla los 2 momentos: la verificación y la actualización en caso de ser posible.

```
Class SalaDeConcierto {


      public synchronized boolean reservar (int nroAsiento) {
            boolean éxito = false;

            if (this.verificarAsiento (nroAsiento)) {
                  this.ocuparAsiento(nroAsiento);
                  éxito = true;
            }

            return (éxito)
      }

….
}
```

y el run del EmpleadoTerminal será como sigue

```
public void run(){
      int asiento;

      while (this.enHorario() ){

// se genera aleatoriamente un nro entre 1 y tamaño de la sala
            asiento = …
            if (salaConc.reservar (asiento)) {
                  System.out.println (“ … ”  )     :
            }
      }
}
```

Objetos sincronizadores (Retomamos el ejemplo del Apunte parte I)

Existen situaciones en las que trabajar con bloques sincronizados permite mejorar el diseño y el desempeño al aumentar el nivel de concurrencia.

Por ejemplo, si una clase tiene varios métodos que requieren exclusión mutua pero actuan sobre distinto grupo de variables, utilizar métodos sincronizados no sería la mejor opcion.

```
Public Class Muestra {
      …  uno;
      … dos;
      … tres;
      … cuatro;

      // actua sobre uno y dos
      public  synchronized void unoDosPrimero ()
            { … }

      // actua sobre uno y dos
      public  synchronized void unoDosSegundo ()
            { … }

      // actua sobre tres y cuatro
      public  synchronized void tresCuatroPrimero ()
            { … }

      // actua sobre tres y cuatro
      public  synchronized void tresCuatroSegundo ()
            { … }

      // acta sobre uno y dos
      public  synchronized void unoDosTercero ()
            { … }
}
```

Si una instancia de la clase Muestra es compartida por varios hilos:

```
...

Muestra  mCompartida = new Muestra(...);
Thread[] hilos = new Thread[6];

for (int i=0; i<6; i++) {
      hilos[i] = new Thread (new RunMuestra(mCompartida, …))
}

// lanzamiento de los hilos, con el rec. comprar. mCompartida

      ...
```

cada vez que un hilo ejecute alguno de estos métodos, tendrá la llave de *mCompartida* en su poder hasta que termine esa ejecución, y mientras tanto los otros hilos no podrán acceder a ninguno de los métodos, ya que TODOS están sincronizados y por ello necesitan la llave. En esto caso, considerando que hay 3 métodos que actuan sobre las variables uno y dos solamente, y los otros 2 métodos actuan sobre las variables tres y cuatro solamente, se podría mejorar la sincronización trabajando con bloques sincronizados. Esos bloques se sincronizarian sobre objetos especialmente creados para lograr esa sincronización.
Una mejora puede lograrse como sigue:

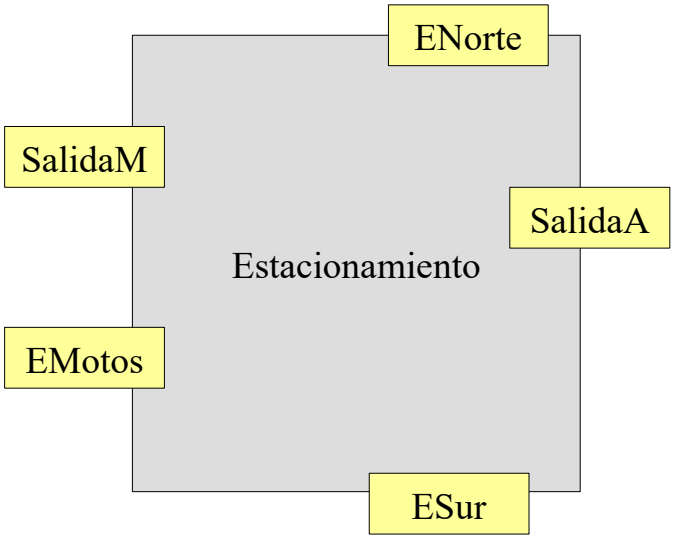
```
Public Class Muestra {
      …  uno;
      … dos;
      … tres;
      … cuatro;
      private Object unoDos = new Object();
      private Objecto tresCuatro = new Object();

      public void unoDosPrimero () {    // actua sobre uno y dos
            synchronized  (unoDos) {... }
      }

      public void unoDosSegundo () {   // actua sobre uno y dos
            synchronized  (unoDos) {... }
      }

      public void tresCuatroPrimero () {   // actua sobre tres y cuatro
            synchronized  (tresCuatro) {... }
      }

      public void tresCuatroSegundo () {  // actua sobre tres y cuatro
            synchronized  (tresCuatro) {... }
      }

      public void unoDosTercero () {  // acta sobre uno y dos
            synchronized  (unoDos) {... }
      }
}
```

De esta forma se utilizan los objetos *unoDos* y *tresCuatro* como **"objetos sincronizadores"**, es decir objetos que se utilizan solo para lograr la exclusión mutua para trabajar sobre el trozo de codigo encerrado en el bloque sincronizado. Asi, mientras un hilo este ejecutando, por ejemplo el método *unoDosPrimero(),* mantendra el lock del objeto *unoDos* solamente, lo que evita que otros hilos puedan ejecutar la secciones críticas de los métodos que utilizan ese mismo lock (que son *unoDosSegundo* y *unoDosTercero),* pero permite que otro hilo pueda ejecutar cualquiera de los métodos que requieren el lock del objeto *tresCuatro.*

Consideremos el caso del estacionamiento de un supermercado:
Un supermercado tiene un estacionamiento con capacidad para autos y motos. Tiene 2 entradas Sur y Norte y una salida dedicadas al ingreso y salida de autos, y 1 entrada y 1 salida dedicadas al ingreso y salida de motos. Tiene capacidad máximoAutos para los autos y maximoMotos para las motos.

Y las acciones posibles son:

- ingreso de autos por la Entrada Norte o la Entrada Sur,
- ingreso de motos por la Entrada para Motos,
- salida de motos por la Salida para motos,
- salida de autos por la Salida para Autos.



Vamos a considerar que en cada punto de ingreso hay un hilo controlIngreso que se encarga de genear la llegada de autos/motos al estacionamiento, y en cada punto de salida hay un hilo controlSalida que se encarga de generar la salida de un auto/moto del estacionamiento. Como el estacionamiento tiene una capacidad máxima para los autos y una capacidad máxima para las motos, los hilos encargados del ingreso deben controlar esa situación. Cuando se genera la llegada de un auto/moto y la capacidad correspondiente del estacionamiento esta completa, la llegada quedara sin efecto (es decir el auto/moto se ira sin ingresar).

Entonces necesitamos:

- 2 hilos para el controlIngreso de autos, uno para la entrada Norte y otro para la entrada Sur
- 1 hilo para el controlSalida de autos
- 1 hilo para el controlIngreso de motos
- 1 hilo para el controlSalida de motos

Todos estos hilos compartiran el recurso Estacionamiento.

El recurso Estacionamiento mantendrá

- la información del maximoAutos, máximoMotos,
- la información de cantAutosActual, y cantMotosActual.
- método ingresarAuto()
- método ingresarMoto()
- método salirAuto()
- método salirMoto()

En una **primera aproximación a la solución** podríamos considerar todos los métodos sincronizados, de forma de asegurarnos la exclusión mutua al trabajar en las zonas críticas (aquellas relacionadas con el acceso y modificación de las variables)

```
public class Estacionamiento {
      private int maxAutos;
      private int maxMotos;
      private int cnatAutosActual= 0;
      private int cnatMotosActual= 0;

      public synchronized boolean ingresarAuto(){
            …
      }

      public synchronized boolean ingresarMoto(){
            …
      }

      public synchronized boolean salirAuto(){
            …
      }

      public synchronized boolean salirMoto(){
            …
      }
}
```

Como el estacionamiento es el recurso compartido por TODOS los hilos, cuando por ejemplo, el hilo *controlIngresoNorte* este ejecutando el método *ingresarAuto*(), va a tener la llave del objeto estacionamiento, y ningun otro de los hilos podrá ejecutar un método sobre estacionamiento. Si bien es correcto que los hilos *controlIngresoSur* y *controlSalidaAutos* esten imposibilitados, no ocurre lo mismo con los hilos *controlEntradaMotos* y *controlSalidaMotos,* ya que estos últimos deben actuar sobre las variables relacionadas a las motos, que no se ven afectadas por el ingreso y salida de autos.

Este es un ejemplo en el que la sincronización de todos los métodos no es la mejor solución, ya que traba la concurrencia.
¿Qué podemos hacer para mejorarlo?

Dado que se pueden identificar claramente 2 partes: el **estacionamiento para motos** y **el estacionamiento para autos**, se puede:

- Trabajar con bloques sincronizados sobre objetos sincronizadores, considerando un objeto para sincronizar el trabajo sobre las motos y otro objeto para sincronizar el trabajo sobre los autos. .
- Particionar el Estacionamiento, para considerar estacionamientoAutos y estacionamientoMotos.
- Considera un objeto sincronizador para, por ejemplo la sincronización sobre las motos y sincronizar los metodos que trabajan sobre el ingreso y la salida de autos, es decir utilizar el lock del objeto compartido estacionamiento.

Con estas soluciones se logra permitir el trabajo sobre los autos de forma concurrente e independiente con el trabajo sobre las motos, dado que el ingreso/salida de motos no afecta a la capacidad disponible para los autos, y el ingreso/salida de autos no afecta a la capacidad disponible para las motos.

```
public class Estacionamiento {
    private int cantAutos = 0;
      //cantidad actual de autos en el estacionamiento

    private int cantMotos = 0;
      //cantidad actual de motos en el estacionamiento

   private int maxAutos = 0;
     // espacio máximo para los autos

   private int maxMotos = 0;
     // espacio máximo para las motos

   private Object objMoto = new Object();
 // objeto que se utilizara para lograr la sincronizacion al
 // considerar el ingreso y salida de motos



// la sincronización sobre el ingreso y la salida de autos se
// trabajara con métodos synchronizados

    public Estacionamiento (int autos, int motos){
        maxAutos = autos;
        maxMotos = motos;
    }
```

```
    public synchronized boolean ingresarAuto (){
        boolean puede = false;
        if (cantAutos < maxAutos) {
           cantAutos ++;
           puede = true;
        }
        return (puede);
    }


    public synchronized void salirAuto (){
        if (cantAutos>0) {
           cantAutos--;
           System.out.println("----------------------
                                se va un auto ");
        }  else System.out.println("-----------------
                                    no hay auto");
    }


    public boolean ingresarMoto (){
        boolean puede = false;
        synchronized (objMoto){
           if (cantMotos < maxMotos) {
                cantMotos ++;
                puede = true;
           }
           return (puede);
        }
    }



    public void salirMoto (){
        synchronized(objMoto) {
            if (cantMotos>0) {
                cantMotos--;
                System.out.println("-----------------
                                    se va una moto ");
            } else System.out.println("no hay moto");
        }
    }
}
```

Nota IMPORTANTE : En el ejemplo, para todos los hilos puede suceder que ejecuten el método cuando no deberían y en ese caso se escribe un mensaje. Esto puede suceder reiteradamente. Es decir, por ejemplo el hilo “controlSalidaMotos” podria intentar registrar la salida de una moto

varias veces sin éxito porque no hay motos en el estacionamiento y entonces escribirá el mensaje cada vez..

Esta situación NO deberia darse, es decir debemos encontrar una solución al problema de que un hilo no haga ese trabajo de espera, mientras ocupa la CPU.
Esto es UN NUEVO PROBLEMA … al que llamamos "***espera activa***"

… Continuará …

Bibliografia

- **Concurrent Programming in JAVA: design principles and patterns** - Doug Lea
- **Thinking in JAVA** - Bruce Eckel - President, MindView Inc. - 2008
- **Seven Concurrency Models in Seven Weeks**. When Threads Unravel - Paul Butcher - The Pragmatic Programmers Inc. - 2014
- **Java Concurrency in Practice** - Brian Goetz - et all - Addison Wesley 2006.
- **Documentacion Oracle**: http://docs.oracle.com/javase/tutorial/essential/concurrency/ y el paquete java.util.concurrent.
- **Orientación a Objetos con Java y UML** - Carlos Fontella - nueva libreria nl 2004