Sincronización de procesos

José Luis Quiroz Fabián
Adriana Pérez Espinosa

Marzo 2017

1 Introducción

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*.

Un SpinLock es un candado no bloqueante, por lo que si un hilo (thread) encuentra el candado abierto, lo toma y continua su ejecución. Por el contrario, si el hilo encuentra el candado cerrado, se queda ejecutando un bucle hasta que es liberado. A pesar de que esto representan una "espera ocupada", generalmente los spinlocks son convenientes ya que muchos recursos son bloqueados por sólo una fracción de milisegundos y consumiría más tiempo que un proceso esperando por uno de esos recursos ceda la CPU y tenga que conseguirla más tarde. Con el uso de spinlocks se evitaría la sobrecarga que implica la replanificación de tareas del sistema operativo.

Por esta razón, los núcleos de los sistemas operativos emplean con frecuencia los spinlocks en circunstancias donde es más probable que sean eficientes. Si el bloqueo se mantiene durante un período elevado de tiempo los spinlocks son muy costosos [*https* : *//es.wikipedia.org/wiki/Semáforo_(informática)*] [*https* : *//es.wikipedia.org/wiki/Spinlock*].

2 Ejemplo: Semáforos y barreras

```
import java.util.Arrays;

public class Tablero {

    static final int TAM = 14;

    private int[] tablero = new int[TAM];
    private int renglon;

    public Tablero() {

    }
    public Tablero(int renglon) {
        this.renglon = renglon;
    }
    public Tablero(int[] tablero, int renglon) {
        this.tablero = tablero;
        this.renglon = renglon;
    }
    public int[] getTablero() {
        return tablero;
    }

    public void setTablero(int[] tablero) {
        this.tablero = tablero;
    }

    public int getRenglon() {
        return renglon;
    }

    public void setRenglon(int renglon) {
        this.renglon = renglon;
    }
```

```
    public String toString(){

        return Arrays.toString(tablero);

    }
}
```

```
import java.util.LinkedList;
import java.util.concurrent.Callable;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.Semaphore;

/*
Considere que en su principal debe declarar:
Semaphore mutex = new Semaphore(1);
CyclicBarrier barrera = new CyclicBarrier(NHILOS);
donde NHILO es la cantidad de hilos que se ejecutaran.
*/

public class ReinasSemaforos implements Callable<Integer>{

    static LinkedList<Tablero> L = new LinkedList<Tablero>();

    static boolean[] vacios = new boolean[Principal.NHILOS];

    private Semaphore mutex;
    private CyclicBarrier barrera;
    private int id;

    ReinasSemaforos(Semaphore mutex, CyclicBarrier barrera, int id){

        this.mutex = mutex;
        this.barrera = barrera;
        this.id=id;

    }

    boolean esComida(int col, Tablero tablero){

        boolean comida=true;
        int i,j;
        int reng = tablero.getRenglon();

        i=reng-1;
        while ( (i>=0) && ((tablero.getTablero())[i]!=col)) /* verifica columna*/
            i--;
        if(i < 0){
            i=reng-1;
            j=col-1;
            while ( (i>=0) && (j>=0) && ((tablero.getTablero())[i]!=j))
            { i--; j--; }
            if( (i<0) || (j < 0))
            {
                i=reng-1;
                j=col+1;
                while ( (i>=0) && (j<=Tablero.TAM) && ((tablero.getTablero())[i]!=j) )
                { i--; j++; }
                if( (i<0) || (j > Tablero.TAM))
                    comida = false;
            }
        }
        return (comida);

    }

    Tablero obtenerDato(){

        int contadorVacios=0;
        Tablero aux;

        do{
            try {
```

```
            mutex.acquire();

        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        if(L.size()>0){

            aux = L.remove();
            mutex.release();
            vacios[id]=false;
            return aux;
        }else{

            if(!vacios[id]){
                vacios[id]=true;
            }
            for(boolean vi:vacios){

                if(vi){
                    contadorVacios++;
                }
            }
            mutex.release();

            if(contadorVacios==Principal.NHILOS){

                return null;
            }else{
                contadorVacios=0;
            }
        }
    }while(true);
}
Integer calcularReinas(){

    int num_sol=0,col;
    Tablero elem;
    Tablero nuevo;

    num_sol=0;


    while ((elem=obtenerDato())!=null){
        for(col=0;col < Tablero.TAM; col++ ){
            if (elem.getRenglon() < (Tablero.TAM-1)){
                if (!esComida(col,elem) ){
                    (elem.getTablero())[(elem.getRenglon())] = col;

                    nuevo = new Tablero(elem.getTablero().clone(),elem.getRenglon()+1);

                    try {
                        mutex.acquire();

                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }

                    L.addFirst(nuevo);

                    mutex.release();

                }

            }
            else
                if (!esComida(col,elem) ){
                    (elem.getTablero())[(elem.getRenglon())] = col;
                    num_sol+=1;
                }
        }
    }


    return new Integer(num_sol);
```

```
    }


    @Override
    public Integer call() throws Exception {

        Tablero init;
        Integer soluciones = null;

        if(id==0){
            init = new Tablero(0);
            L.addFirst(init);

        }

        barrera.await();

        soluciones = calcularReinas();
        return soluciones;
    }


}
```

Para ejecutar el ejemplo anterior, desarrolle su clase Principal. Observe que se esta usando hilos *Callable*.

3 Ejercicios

1.- Implementar los SpinLock haciendo uso de la interface *Lock* de Java. Los métodos a implemenetar son:

```
public interface Lock {

    /**
     * Acquires the lock only if it is free at the time of invocation.
     *
     * <p>Acquires the lock if it is available and returns immediately
     * with the value {@code true}.
     * If the lock is not available then this method will return
     * immediately with the value {@code false}.
     *
     * <p>A typical usage idiom for this method would be:
     *  <pre> {@code
     * Lock lock = ...;
     * if (lock.tryLock()) {
     *   try {
     *     // manipulate protected state
     *   } finally {
     *     lock.unlock();
     *   }
     * } else {
     *   // perform alternative actions
     * }}</pre>
     *
     * This usage ensures that the lock is unlocked if it was acquired, and
     * doesn't try to unlock if the lock was not acquired.
     *
     * @return {@code true} if the lock was acquired and
     *         {@code false} otherwise
     */
    boolean tryLock();
    /**
     * Releases the lock.
     *
     * <p><b>Implementation Considerations</b>
     *
     * <p>A {@code Lock} implementation will usually impose
     * restrictions on which thread can release a lock (typically only the
     * holder of the lock can release it) and may throw
     * an (unchecked) exception if the restriction is violated.
     * Any restrictions and the exception
     * type must be documented by that {@code Lock} implementation.
     */
    void unlock();
```

```
}
```

Recomendación: Dentro de su clase SpinLock hacer uso de un atributo de tipo *AtomicInteger* mediante el cual se pueda hacer uso del método *compareAndSet*. Usando el método anterior (y su Id: $Thread.currentThread().getId()$ que es único) un hilo pude verificar si el candado está libre o no.

```
Constructor and Description

AtomicInteger()
Creates a new AtomicInteger with initial value 0.
AtomicInteger(int initialValue)
Creates a new AtomicInteger with the given initial value.

Method Summary

public final boolean compareAndSet(int expect,
                  int update)

Atomically sets the value to the given updated value if the current value == the expected value.

Parameters:
    expect - the expected value
    update - the new value
Returns:
    true if successful. False return indicates that the actual value was not equal to the expected value.
```

```
import java.util.concurrent.atomic.AtomicInteger;

public class EjemploAtomicInteger {

    private static AtomicInteger at = new AtomicInteger(0);

       static class MiHiloRunnable implements Runnable {

           private int miContador;
           private int miContadorPrevio;
           private int miContadorMasCinco;
           private boolean esNueve;

           public void run() {
               miContador = at.incrementAndGet();
               System.out.println("Thread " + Thread.currentThread().getId() + " / Contador : " + miContador);
               miContadorPrevio = at.getAndIncrement();
               System.out.println("Thread " + Thread.currentThread().getId() + " / Valor Previo : " +
                    miContadorPrevio);
               miContadorMasCinco = at.addAndGet(5);
               System.out.println("Thread " + Thread.currentThread().getId() + " / + Cinco : " +
                    miContadorMasCinco);
               esNueve = at.compareAndSet(9, 3);
               if (esNueve) {
                   System.out.println("Thread " + Thread.currentThread().getId()
                   + " / El valor fue Nueve por lo que se actualizo" + at.intValue());
               }
            }
       }

       public static void main(String[] args) {
           Thread t1 = new Thread(new MiHiloRunnable());
           Thread t2 = new Thread(new MiHiloRunnable());
           t1.start();
           t2.start();
       }
}
```

2.- En el ejemplo de las reinas cambie los semáforos por candados Spinlock y calcule el tiempo de ejecución para un tablero con 14 y 15 reinas.