

Computación Concurrente - Tarea 3

Damián Rivera González
Alexis Hernandez Castro

October 15, 2019

1. Acabas de entrar a trabajar en una empresa, en la que al parecer no tienen idea de como funciona el cómputo concurrente. Y en la cual tienen el siguiente código que incrementa en dos unidades una variable compartida utilizando dos hilos. Describe detalladamente al menos 4 problemas (condición de carrera, datos, despertar perdido, etc.) que tiene el código. Posteriormente, muestra una solución en la que para toda ejecución siempre imprima 2. NOTA: No consideres errores de compilación, solo enfocate en aquellos de semántica.

```
public static void main(String[] args) {  
    int count = 0;  
    Thread t1, t2 = new Thread(new Runnable() {  
        public void run() {  
            count++;  
        }  
    });  
    t1.start();  
    t2.start();  
    System.out.println(count);  
    for (int i = 0; i < 2; i++) {  
        condition.await();  
    }  
}
```

Veamos a la instrucción de la línea 7, *count* ++, como:

```
1  int temp = counter;  
2  counter = temp + 1;
```

- Condición de Carrera:
Por lo cual la instrucción genera una condición de carrera ya el resultado del programa será diferente dependiendo del orden de la ejecución por los dos hilos.
- Condición por los datos:
De igual manera la instrucción de la línea 7 genera una condición por los datos, ya que ambos hilos acceden al valor de la variable *count* lo leen y existe una escritura cuando se reasigna el valor de la misma.

- Otro:
 - Otro:
2. Utilizando únicamente semáforos crea una estructura de datos concurrente que pueda guardar N objetos, y que implemente la siguiente interfaz:

Algoritmo Productor/Consumidor

```
public interface Buffer<T> {
    // Agrega un elemento al buffer; se bloquea si esta lleno.
    public void put(T item);
    // Elimina un elemento del buffer; se bloquea si esta vacio.
    public T get();
    //Obtiene el número de elementos en el buffer.
    public int count();
}
```

Creamos la estructura de la siguiente manera:

```
1 public class Estructura implements Buffer<T>{
2
3     BoundeQueue<T> e = new BoundeQueue<T>(N);
4     Semaphore usarE = new Semaphore(1);
5     Semaphore emptyE = new Semaphore(n); // Lugares libres
6     Semaphore fullE = new Semaphore(0); // Lugares ocupados
7
8     public void put(T item) {
9         emptyE.acquire(); // Vemos si podemos tomar
10        usarE.acquire();   // Para poder usar SC
11        e.put(item);
12        fullE.release();   // Hay un lugar mas ocupado
13        usarE.release();   // Ya no usamos SC
14    }
15
16    public T get() {
17        T item;
18        fullE.acquire();   // Vemos si hay almenos un elemento
19        usarE.acquire();   // Para poder usar SC
20        item = e.get();
21        emptyE.release();  // Mostramos que quitamos un elemento
22        usarE.release      // Ya no usamos SC
23        return item;
24    }
25
26    public int count(){
27        return e.size();
28    }
29 }
```

3. Utilizando la misma interfaz del ejercicio 1, da una implementación pero ahora utilizando únicamente candados y condiciones. ¿Tu implementación sufre del problema del despertar perdido? Argumenta porque no, o muestra un escenario en donde suceda.

```
1 public class Estructura implements Buffer<T>{
```

```

2     BoundeQueue<T> e = new BoundeQueue<T>(N);
3     ReentrantLock usarE = new ReentrantLock();
4     Condition fullE = usarE.newCondition();
5     Condition emptyE = usarE.newCondition();
6
7     public void put(T item) {
8         usarE.lock();
9         while (e.isFull()) { // Si E esta llena esperamos
10             fullE.await();
11         }
12         e.put(item);
13         emptyE.signal(); // Al agregar un elemento avisamos que ya
no es vacia
14         usarE.unlock();
15     }
16
17     public T get() {
18         T item;
19         usarE.lock();
20         while(e.isEmpty()){ // Si E esta vacia esperamos
21             emptyE.await();
22         }
23         item = e.get();
24         fullE.signal(); // Al quitar un elemento avisamos que ya no
esta llena
25         usarE.unlock();
26         return item;
27     }
28
29     public int count(){
30         return e.size();
31     }
32 }

```

Esta implementación no sufre el problema del despertar perdido, ya que en cada adición de un elemento en la estructura justo la siguiente instrucción es mandar un signal para indicar que ya hay al menos un elemento más (en caso de que alguien quiera tomar uno). De igual manera si quitamos un elemento la siguiente instrucción es avisar que al menos hay un elemento menos (en caso de que alguien quiera agregar otro). Por lo que hacemos tantos **Signal** como elementos agregamos o quitamos por lo que avisamos en todo momento a los hilos que esten esperando en una u otra cola de condición.

4. Demuestra que en el candado **ReadWriteLock** visto en clase, cuando un lector ejecuta **signalAll** solo puede haber hilos escritores bloqueados.

Tenemos los siguiente:

```

1 // Readers
2
3 public void lock(){
4     mutex.lock();
5     while(writer){
6         condition.await();
7     }
8     readAcquires++;
9     mutex.unlock();
10 }
11
12 public void unlock(){
13     mutex.lock();
14     readers--;
15     if(readers == 0){
16         condition.signalAll();
17     }
18     mutex.unlock();
19 }

```

```

1 // Writers
2
3 public void lock(){
4     mutex.lock();
5     while(writer){
6         condition.await();
7     }
8     writer = true;
9     while(readAcquires !=
10         readReleases){
11         condition.await();
12     }
13     mutex.unlock();
14 }
15 public void unlock(){
16     mutex.lock();
17     writer = false;
18     condition.signalAll();
19     mutex.unlock();
20 }

```

5. [2 pts] En clase vimos que la implementación de la estructura SimpleBoard le da prioridad a los lectores. Posteriormente, discutimos una implementación que utiliza spin-locks para darle prioridad a los escritores. Modifica la estructura SimpleBoard para que ahora únicamente use semáforos. Argumenta porque no puede haber un escritor y lector dentro de la S.C. Hint: Utiliza 4 semáforos binarios, dos de ellos para proteger las escrituras a las variables y los otros 2 para controlar la entrada de los lectores y escritores.
6. Escribe un código genérico utilizando semáforos (de preferencia muy corto) para un hilo tk de manera que se pueda producir inanición (suponiendo que todos los hilos ejecutan el mismo código) si los semáforos son débiles, pero no podría ocurrir si los semáforos son fuertes. Tu código no debe tener deadlocks, independientemente del tipo de semáforo que se utilice.

```

1 public class SimpleBoard<T>(){
2
3     private int readers = 0;
4     private semaphore emptySC = new Semaphore(1);
5     private T message;
6     private Semaphore permiteLector = new Semaphore(1);
7
8
9     public void write (T msg){
10         emptySC.acquire();
11         message = msg;
12         emptySC.release();
13     }
14 }
15
16 public void read(){
17     permiteLector.acquire();
18     if(readers == 0) emptySC.acquire();
19     readers ++;
20     permiteLector.release();

```

21
22 }
23
24 }

Para este caso un hilo t_k muere de inanición si un semáforo es débil ya que los semáforos débiles tienen la característica de mandar cualquier hilo o sea que no cumplen la característica de justicia, en los semáforos fuertes van entrando de una manera ordenada en este caso el hilo t_k no muere de inanición ya que en el turno k pasara.

7. Considera el siguiente código:

Algoritmo Productor/Consumidor

```
class Mysterious{
    private int x = 1;
    private ReentrantLock mutex = new ReentrantLock();
    private Condition c = mutex.newCondition();
    private Condition d = mutex.newCondition();

    public void foo1() {
        mutex.lock();
        if (x == 2) { c.await(); }
        x = x + 1;
        d.signalAll();
        mutex.unlock();
    }

    public void foo2() {
        mutex.lock();
        if (x == 0) { c.signalAll(); d.await(); }
        x = x - 1;
        mutex.unlock();
    }
}
```

- a) Demuestra que no tiene deadlocks bajo el supuesto de que los hilos invocan infinitamente los métodos `foo1()` y `foo2()`.

Supongamos que existe un deadlock entonces los hilos se encolan en `c` y `d` para esto la variable `x` tiene que tener el valor de 2 y de 0 al mismo tiempo notemos que esto no es posible ya que el candado asegura la exclusión mutua para la variable compartida `x`.

- b) Supón ahora que el sistema incluye únicamente los hilos `t`, `u`, `v`. Describe un escenario en donde el hilo `t` en algún punto muere de inanición. Ningún hilo falla, por lo que el calendarizador puede ejecutarlos en cualquier momento.

En esta ejecución consiste en atorar a `t` en la cola `c` y mantener con los hilos `u` y `v` el valor de $x > 0$ para que el hilo `t` nunca despierte.

hilo t	hilo c	hilo v	memoria
			x = 1
foo1, x = x+1			x = 2
foo1 , read(x),c.wait()			x = 2
	foo2,x = x-1		x = 1
		foo1,x =x+1	x = 2
	foo2, x = x-1		x = 1
		foo1,x =x+1	x = 2
	foo2, x = x-1		x = 1

8. Otra manera de resolver el problema de la barrera reusable para N hilos es la siguiente: Un hilo especial llamado manager espera por todos los hilos participantes a que lleguen a la barrera. Cuando todos llegan les devuelve la señal a todos. Muestra un algoritmo que implemente esta misma idea utilizando semáforos y demuestra que ningún hilo puede traspasar la barrera dos o más veces seguidas.

```
1
2 public void enter(){
3     synchronized(this){
4         thread.manager.wakeup();
5         if(thread.name == Manager){
6             while(haveEnter != N){
7                 thread.sleep();
8             }
9             gate.release();
10            notifyAll();
11        }
12    }
13 }else{
14     if(semaphores[thread.ID].getValue() < 1){
15         semaphores[thread.ID].release();
16         haveenter++;
17     }else{
18         wait();
19     }
20 }
21 }
```

```
1
2 public void leave(){
3     synchronized(this){
4         if(thread.name == manager){
5             while(haveEneter != 0){
6                 thread.sleep();
7             }
8             gate.acquire();
9         }else{
10            if(semaphores[thread.ID].getValue() > 0){
11                semaphores[thread.ID].acquire();
12                haveEneter--;
13            }
14        }
15    }
16 }
```

Para esta solución utilizamos un arreglo de semaforos donde cada hilo tiene un semaforo para entrar a la barrera el hilo manager verifica el numero de hilos que han dado permiso a su propio semaforo una vez que todos los semaforos estan prendidos entra a la seccion critica, el metodo leave verifica que todos los hilos abandonen la barrera y el hilo manager cierra la puerta para volverla reusable.