Computación Concurrente - Tarea 3

Damián Rivera González
Alexis Hernandez Castro

October 9, 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 Estrucutura 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 Estrucutura 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 sigueinte:

```
// Readers

public void lock(){
  mutex.lock();
  while(writer){
    condition.await();
  }
  readAcquires++;
  mutex.unlock();
}

public void unlock(){
  mutex.lock();
  readers--;
  if(readers == 0){
    condition.signalAll();
  }
  mutex.unlock();
}
```

```
// Writers

public void lock(){
  mutex.lock();
  while(writer){
    condition.await();
  }
  writer = true;
  while(readAcquires !=
   readReleases){
    condition.await();
  }
  mutex.unlock();
}

public void unlock(){
  mutex.lock();
  writer = false;
  condition.signalAll();
  mutex.unlock();
}
```

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.

7. Considera el siguiete 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().

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.

8. Otra manera de resolver el problema de la barrera reutilizable 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.