COMPUTACIÓN CONCURRENTE

EXAMEN 2: Bloqueo, Semáforos y Barreras

Prof. Manuel Alcántara Juárez
manuelalcantara52@ciencias.unam.mx

Alejandro Tonatiuh Valderrama Silva
at.valderrama@ciencias.unam.mx

José de Jesús Barajas Figueroa
jebarfig21@ciencias.unam.mx

Luis Fernando Yang Fong Baeza
fernandofong@ciencias.unam.mx

Ricchy Alain Pérez Chevanier
alain.chevanier@ciencias.unam.mx

Fecha Límite de Entrega: 10 de Diciembre de 2020 a las 08:00:00 CT

1. Considera la siguiente implementación, la cual no sufre del problema del despertar perdido ni ficticio, y donde dos hilos invocan una sola vez y uno solo de los siguientes métodos:

```
Initialization:
Lock mutex = new ReentrantLock();
Condition waitForAJob = mutex.newCondition();
boolean hasSignalled = false;

-.  public void waitForWork() {
1.    mutex.lock();
2.    System.out.println("Inside WaitForWork Lock");
3.    while(!hasSignalled) {
4.      waitForJob.await();
5.      System.out.println("WaitForWork woke up");
-.    }
6.    mutex.unlock();
-.  }
-.  public void setDataReady() {
7.   mutex.lock();
8.   hasSignalled = true;
9.   waitForAJob.signal();
10.   mutex.unlock();
-.  }
```

   a) **[1.0pt]** ¿Cuál de las siguientes sucesiones de instrucciones representa una ejecución válida? Analízalo desde una perspectiva teórica.

      1) 1, 2, 7, 3, 4, 8, 9, 10, 1, 5, 3, 6. No se puede porque no hay forma de ejecutar la instruccion 1 dos veces
      2) 7, 1, 2, 3, 4, 8, 9, 1, 5, 6, 10. No se puede porque no hay forma de ejecutar la instruccion 1 dos veces
      3) 7, 1, 2, 3, 4, 8, 9, 5, 3, 6, 10. No porque de ser asi dos hilos estan en la secccion critica al mismo tiempo

4) 1, 2, 3, 7, 4, 8, 9, 10, 5, 3, 6. Me gusta, esta es la posible :D
5) 1, 7, 2, 3, 4, 8, 9, 5, 10, 3, 6. No porque de ser asi dos hilos estan en la secccion critica al mismo tiempo

b) **[1.0pt]** Modificando hasta 3 líneas de código, crea una implementación que no sufra del despertar perdido pero si del despertar ficticio. Muestra la ejecución.

```
Initialization:
Lock mutex = new ReentrantLock();
Condition waitForAJob = mutex.newCondition();
boolean hasSignalled = false;

public void waitForWork() {
  mutex.lock();
  System.out.println("Inside WaitForWork Lock");
  while(!hasSignalled) {
    waitForJob.await();
    System.out.println("WaitForWork woke up");
  }
  mutex.unlock();
}
public void setDataReady() {
    mutex.lock();
    waitForAJob.signal(); // esta es la unica linea que cambie
    hasSignalled = true;
    waitForAJob.signal();
    mutex.unlock();
}
```

Con esto despertamos al hilo causando un despertar ficticio pero nunca perdera el verdadero signal por lo tanto no hay despertar perdido.

c) **[0.5pt]** Modificando hasta 3 líneas de código, crea una implementación que sufra del despertar ficticio y del despertar perdido. Da la ejecución para este último caso.

```
Initialization:
Lock mutex = new ReentrantLock();
Condition waitForAJob = mutex.newCondition();
boolean hasSignalled = false;

public void waitForWork() {
  mutex.lock();
  System.out.println("Inside WaitForWork Lock");
  while(!hasSignalled) {
    waitForJob.await();
    System.out.println("WaitForWork woke up");
  }
  mutex.unlock();
}
public void setDataReady() {
    mutex.lock();
    waitForAJob.signal(); // esta es la unica linea que cambie
    hasSignalled = true;
    mutex.unlock();
}
```

Con esto despertamos al hilo causando un despertar ficticio, el hilo de waitForWork se despierta pero cuando se despierta la condicion no se ha cumplido, pero es mas, nunca se va a despertar otra vez, por lo tanto aunque la condicion se cumplio el hilo nunca va a despertar, por lo tanto tenemos un despertar perdido.

2. Una empresa te acaba de contratar porque un virus (proceso) maligno invadió su sistema y esta provocando inconsistencia de información. Tu jefe te comentó que si se garantiza la M exclusión mutua ya no habrá problema. Sabiendo eso, te pusiste a buscar en el código y encontraste la siguiente clase:

```
1.  public class MExclusion implements Lock {
2.    private Semaphore mutex;
3.    // @param n: total number of threads
4.    // @param m: allowed threads in C.S.
5.    public MExclusion(int n, int m) {
6.      mutex = new Semaphore(m);
7.    }
8.    public void lock() {
9.      mutex.acquire();
10.   }
11.   public void unlock() {
12.     mutex.release();
13.   }
14. }
```

a) **[1.0pt]** Sabiendo que el virus es uno de los procesos, argumenta como podría violar la propiedad de la M exclusión. Muestra la ejecución.

```
final var m = new MExclusion(4, 2);
final var t1 = new Thread(() -> {
    m.lock();
    ...
    m.unlock();
});

final var t2 = new Thread(() -> {
    m.lock();
    ...
    m.unlock();
});

final var t3 = new Thread(() -> {
    m.lock();
    ...
    m.unlock();
});

final var t4 = new Thread(() -> { // the bad one
    m.unlock();
});

t1.start();
t2.start();
t3.start();
t4.start();
```

Aqui lo que podria hacer el virus si supiera como esta implementada la clase MExclusion es que podria llamar a unlock sin antes haber hecho lock, imaginemos que primero se ejecuto t1, y entro, luego t2 y entro, luego t3 y por como se diseño se queda t3 atorados porque solo pueden estar 2 hilos en la seccion critica, pero luego como t4 hizo un unlock ahora t3 se mueve y tenemos 3 hilos en la seccion critica con un lock que solo se planeo para dos accesos simultaneos.

b) **[1.5pt]** Agregando únicamente código y sin usar spins, modifica la clase para prevenir un comportamiento inadecuado. HINT: Puedes lanzar una excepción si se detecta un problema. Argumenta porque funciona tu solución.

```java
public class MExclusion implements Lock {
  private Semaphore mutex;
  private ThreadLocal<Boolean> isOnCS = new
  ↪  ThreadLocal<Boolean>(false);

  // @param n: total number of threads
  // @param m: allowed threads in C.S.
  public MExclusion(int n, int m) {
    mutex = new Semaphore(m);
  }
  public void lock() {
    final var isOnCSNow = isOnCS.get();
    if (!isOnCSNow) {
        mutex.acquire();
        isOnCS.set(true);
    }
  }
  public void unlock() {
    final var isOnCSNow = isOnCS.get();
    if (!isOnCSNow) {
        throw new IllegalMonitorStateException();
    }
    isOnCS.set(false);
    mutex.release();
  }
}
```

El unico problema que tenias es que no habia una restriccion de que teniamos que podiamos llamar a lock muchas veces o peor aun llamar a unlock sin antes tener el lock, por lo tanto usando una variable booleana podemos evitar que haya problema si se llama a lock muchas veces y mejor aun, lanzar una excepción si alguien quiere llamar a unlock sin tener el ock.

3. Considera el problema del Productor/Consumidor para dos hilos, en donde el productor tiene espacio ilimitado, y ambos hilos invocan el método `solve`, ya sea para producir o consumir.

   a) **[1.0pt]** Da una implementación para el método `void solve` que solucione el problema utilizando un solo semáforo.

```
public class ProducerConsumer {
    private Semaphore mutex;

    public ProducerConsumer() {
      mutex = new Semaphore(1);
    }

    public void solve(final boolean isConsumer) {
        mutex.adquire();

        if (isConsumer) {
            ... // consume code
        }
        else {
            ... // producer code
        }

        mutex.release();
    }
}
```

b) **[1.0pt]** ¿Si el semáforo es débil o fuerte, podría ocurrir que el productor o consumidor se muera de inanición tratando de ejecutar el método `solve`? Argumenta tu respuesta.

Claro, si es un semaforo fuerte, pues todo bien, cuando acabe un agente el otro agente esta garantizado (si es que estaba esperando) a entrar despues, pero si tenemos un semaforo debil nada impide por ejemplo a consumidor de seguir entrando y entrando a un lugar donde ya no hay nada que consumir sin dejar pasar al productor.

c) **[1.0pt]** Supón ahora que los hilos invocan una sola vez el método `solve`. Bajo esta nueva suposición, ¿Se podría utilizar un mutex en lugar del semáforo para resolver el problema? Argumenta tu respuesta.

Muy buena pregunta, bueno, si solo tenemos un productor que va a producir una vez o un consumidor que va a consumir una vez, claro que se podria, de hecho se puede reescribir el codigo usando un ReentrantLock. Este solo permite acceder a un de los dos hilos a la SC, y deja fuera al otro hasta que el anterior salga, solucionando perfectamente el problema.

4. **[1.0pt]:** Resuelve uno solo de los siguientes ejercicios:

a) Da una implementación con a lo más 10 lineas de código, en la que si se utilizan semáforos débiles se tenga hambruna, pero si se usan semáforos fuertes no. No debe de haber deadlocks.

b) Muestra una ejecución en donde los hilos logren pasar la barrera del código descrito en la siguiente figura.

c) Muestra una ejecución en donde los hilos se queden en deadlock utilizando el código descrito en siguiente figura.

```
// 4 hilos continuamente invocan el método wait.
Initialization:
int hasPassed = 0;
Semaphore open = new Semaphore(0);
final int n;

1. public void wait() {
2.  synchronized(this) { hasPassed++; }
3.  if (hasPassed == n) open.release();
4.  open.acquire();
5.  open.release();
6.  synchronized(this) { hasPassed--; }
7.  if (hasPassed == 0) open.acquire();
8. }
```

Si lanzas al hilo 1 se va a quedar atorado en 4, los primeros 3 hilos hacen eso, pero luego el 4to libera un lugar para que el primer hilo baje y este a su vez da un release mas, permitiendo que bajen y asi, total, al final tiene a los 4 hilos en la linea 6, los 6 minizan la variable hasPassed hasta que vale 0. entonces los 4 hilos van a ejecutar uno por uno la linea 7 con esto 3 de ellos quedan atorados, pero uno pasa o entra, ahora mismo hasPassed vale 0, por lo tanto cuando el hilo que paso vuelve a llamar al wait se queda atorado en la linea 4, y con eso tiene 4 hilos atorados.

5. Haz sido contrado por Greenpeace para intentar incrementar la población de las ballenas que han tenido problemas de sincronización para encontrar pareja. El truco es que para tener hijos, se necesitan tres ballenas, un macho, una hembra y otra para jugar al casamentero, literalmente, para juntar las otras dos ballenas y avisarles cuando el acto ha concluido. Cada ballena se representa por un proceso independiente y es necesario implementar los siguientes métodos `Macho()`, `Hembra()`, `Casamentero()`. Una ballena macho invoca `Male()` y espera hasta que haya una hembra y un casamentero; del mismo modo, una ballena hembra debe esperar hasta que estén presentes una ballena macho y un casamentero. Una vez que los tres están presentes, los tres finalizan la invocación del método.

   a) **Opción A [2.0pt]:** Da una implementación para los 3 métodos utilizando únicamente semáforos. Argumenta porque es correcta y muestra un ejemplo.

```
class Solver {
    int num_threads_until_here = 0;
    Semaphore lock = new Semaphore(1);
    Semaphore open = new Semaphore(0);

    wait() {
        lock.adquire(); // lock num_threads_until_here
            num_threads_until_here++;
            if (num_threads_until_here == 3) open.release();
        lock.release(); // unlock num_threads_until_here

        open.adquire(); // stop until all
        open.release();
    }

    Macho() {
        wait();
    }

    Hembra() {
        wait();
    }

    Casamentero() {
        wait();
    }
}
```

Ya quedo, la idea es usar un semaforo que inicia con 1 lugar disponible como lock, para hacer operaciones sobre un contador de manera atomica, luego, si no estan todos paramos a los threads en stop until all cuando llegue el ultimo hacemos un release extra para hacer una cadenita e ir liberando a todos.

b) **Opción B [1.0pt]:** Da una implementación para los 3 métodos utilizando únicamente candados. Argumenta porque es correcta y muestra un ejemplo.

```
class Solver {
        int num_threads_until_here = 0;
        Lock lock = new ReentrantLock();
        Lock master = new ReentrantLock();
        Condition waiting = master.newCondition();

        wait() {
            lock.lock();
                num_threads_until_here++;
            lock.unlock();

            if (num_threads_until_here == 3) {
                master.lock();
                waiting.signal();
                master.unlock();
            } else {
                master.lock();
                waiting.await();
                waiting.signal();
                master.unlock();
            }
        }

        Macho() {
            wait();
        }

        Hembra() {
            wait();
        }

        Casamentero() {
            wait();
        }
    }
```

La idea es sencilla, usamos un lock para poder aumentar un contador de manera atomica, luego los primeros dos hilos se van a quedar en el await esperando a que alguien los despierte, el 3 hilo es el que despierta a uno de ellos, ese hilo 1 o 2 despierto va a hacer otro signal para el otro hilo y con eso estamos libres tanto el hilo 3, como el 1 o 2 y finalmente el 2 o 1.