Universidad Nacional Autónoma de México

Facultad de Ciencias

Computación Concurrente

Tarea 08

Condiciones

Profesor:

Salvador González Arellano

Integrantes:

Contreras Ibarra Francisco

Marín Parra José Guadalupe de Jesús

Ortega Gónzalez José Ethan

Ramírez López Alvaro

1. Teoría:

1. Supongamos que tenemos 2 funciones A y B. Ambas funciones son ejecutadas en forma paralela por n y m hilos respectivamente. La función A se ejecuta primero y a partir de ahí, los hilos que ejecutan la función B, no pueden iniciar hasta que todos los hilos hayan ejecutado la función A y viceversa. Propón una implementación dando un esbozo.

   Solución: Podemos entender la problemática como una carrera de relevos en la cual la siguiente función, es decir, la B, no puede iniciar hasta que llegue la función A. Análogamente se hace con el inverso. Veamos el siguiente pseudocódigo que resuelve el problema.

Algorithm 1 funciones(A,B)

```
1: thread n;
2: thread m;
3: boolean terminado = false;
4: boolean recorrido = false;
5: while recorrido == false do
6:     n.iniciar;
7:     if n.terminado then
8:         m.iniciar();
9:         n.terminado = true;
10:        if m.terminado then
11:            m.terminado = true;
12:        end if
13:    else
14:        m.esperar();
15:        terminado = false;
16:    end if
17:    if n.terminado && m.terminado then
18:        recorrido = true;
19:    end if
20: end while
21: print n,m
```

2. Otra manera de resolver el problema de la barrera reutilizable para N es la siguiente:

   a) Un hilo especial llamado "Encargado" o "Manager" espera por todos los hilos participantes a que lleguen a la barrera. Cuando todos llegan les devuelve la señal a todos. Propón un algoritmo o un esbozo de implementación que resuelva esta idea, puedes hacerlo usando semáforos o condiciones o ambas.

   Solución:

3. ¿Qué hace el siguiente código?, si tiene errores escríbelos y también el cómo solucionarlos.

```
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();
  }
}
```

Solución: Este código implementa un contador, la variable `x` puede verse como un contador donde los métodos `foo1` y `foo2` van a incrementarlo o decrementarlo respectivamente. En estos métodos hacemos usos de condiciones, en `foo1` cuando llegamos que `x = 2` esperamos, y de igual forma en `foo2` cuando `x = 0`. Sin embargo, el código presenta varios problemas:

- Ni `foo1`, ni `foo2` hacen uso de los bloques try/finaly, por lo que si ocurre una excepción en los métodos el código no liberaría el mutex, por lo que ocurre un error.
- Ni `foo1`, ni `foo2` lanzan la excepción `InterruptedException` que es necesaría al usar los métodos `signalAll`.
- Las llamadas a las condiciones están mal, algunas no están en donde deberían, como la condición en el if de `foo2`.

Una mejor implementación se muestra a continuación:

```java
private int counter = 1;
private static final int LIMIT = 8;

private ReentrantLock mutex = new ReentrantLock();
private Condition counterZero = mutex.newCondition();
private Condition counterLimit = mutex.newCondition();

public void increment() throws InterruptedException {
    try {
        mutex.lock();
        if (counter == LIMIT) {
            System.out.println(Thread.currentThread().getName() + " wait on counter limit");
            counterLimit.await();
        }
        System.out.println("Incrementing the counter " + counter);
        counter++;
        counterZero.signalAll();
    } finally {
        mutex.unlock();
    }
}

public void decrement() throws InterruptedException {
    try {
        mutex.lock();
        if (counter == 0) {
            System.out.println(Thread.currentThread().getName() + "wait on counter 0 value");
            counterZero.await();
        }
        System.out.println("Decrementing the counter " + counter);
        counter--;
    } finally {
        counterLimit.signalAll();
        mutex.unlock();
    }
}
```

Para el método `foo1`, en la condición del if, ahora se usa una constante para mayor flexibilidad.

2. Extra:

Por si sientes que no pasarás el Zera examen, este te ha dejado un encargo, se trata de investigar los siguientes conceptos y explicar dando un pequeño ejemplo.

- ¿Qué es un semáforo fuerte en Java?

Solución: Entendemos por semáforo a una señal que puede detener procesos en alguna posición determinada. Los semáforos fuertes son aquellos cuya cola de procesos sigue una política FIFO estricta.

- ¿Qué es un semáforo débil en Java?

Solución: Son aquellos cuyos procesos se seleccionan de forma aleatoria.

- ¿Cómo usamos los threadPools en java?

Solución: Permiten crear un conjunto de hilos que se van procesando dentro de una cola conforme se van completando los anteriores. Para usarlos hay que tomar en cuenta lo siguiente.

Tenemos 3 clases core, *ThreadPool*, *WorkerThread* y *Done*. Implementamos la clase Worker que implemente la interfaz Runnable para después desde cualquier otra clase, crear el ThreadPool con un número de hilos para asignárselos a los Workers, de esta forma estamos usando ThreadPools.

- Investiga para que sirve el Future en Java (va relacionado con hilos).

Solución: Es un objeto que se construye para albergar un valor en un futuro. El uso principal es su método *get()* el cual obtiene el valor real del futuro, en caso de que el futuro todavía no se haya completado, al llamar a este método nos quedaremos bloqueados hasta que se complete.