COMPUTACIÓN CONCURRENTE
PRÁCTICA 2, PRIMITIVAS DE SINCRONIZACION

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: 12 de Abril de 2021 a las 23:59:59pm.

1. Objetivo

NOTA: REPLANTEAR OBJETIVO El objetivo de esta práctica es implementar algunos algoritmos de sincronización vistos en clase y una aplicación del uso de las primitivas de sincronización que construiste mediante estos algoritmos.

2. Introducción

NOTA: ¿qué conocimientos teóricos y de java tiene que conocer el estudiante para poder hacer esta práctica?

3. Desarrollo

En esta práctica trabajarás con una base de código construida con Java 9[1] y Maven 3, también proveemos pruebas unitarias escritas con la biblioteca `Junit 5.5.1` que te darán retrospectiva inmediata sobre el correcto funcionamiento de tu implementación[2] .

[1] De nuevo puedes utilizar cualquier versión de java que sea mayor o igual a Java 8 simplemente ajustando el archivo pom.xml

[2] Bajo los casos que vamos a evaluar, mas estas no aseguran que tu es implementación es correcta con rigor formal

Para ejecutar las pruebas unitarias necesitas ejecutar el siguiente comando:

```
$ mvn test
```

Para ejecutar las pruebas unitarias contenidas en una única clase de pruebas, utiliza un comando como el siguiente:

```
$ mvn -Dtest=PetersonLockTest test
```

En el código que recibirás la clase `App` tiene un método *main* que puedes ejecutar como cualquier programa escrito en *Java*. Para eso primero tienes que empaquetar la aplicación y finalmente ejecutar el jar generado. Utiliza un comando como el que sigue:

```
$ mvn package
...
...
$ java -jar target/practica02.jar
```

4. Evaluación

Además de que tu código pase las pruebas unitarias sin que estas hayan sido modificadas, también debes de escribir una justificación breve de por qué tu solución cumple con las propiedades requeridas en la descripción del problema, añade estas justificaciones al inicio de cada clase o como pequeños comentarios intermedio dentro de tu implementación.

5. Problemas

5.1. Algoritmo de Peterson

Para esta actividad tienes que implementar el algoritmo de Peterson visto en clase para generar un *candado* que solucione el problema de la exclusión mutua para dos hilos. En el código fuente que acompaña a este documento encontrarás la clase

`PetersonLock` que implementa la interfaz `Lock`, tu tarea es completar la implementación de esta clase. Para validar que tu implementación es correcta tienes que pasar todas las pruebas unitarias que se encuentran en la clase `PetersonLockTest`.

5.2. Problema de los ladrones

En una isla ubicada en la villa de Concurrentelandia, tenemos a un policía custodiando un botín que tiene una palabra clave secreta y dicho botín solo puede ser tomado con dicha palabra clave, sabiendo que este botín está formado de los impuestos de la gente, un grupo de n ladrones quiere obtener este botín para regresarlo al pueblo. Al mismo tiempo el policía se enteró del plan de los ladrones y este decidió llamar refuerzos, sin embargo no llegarán antes de un tiempo x, de esta manera el policía decide estar dando rondines para asergurarse de que los ladrones no logren robarse el botín hasta que lleguen los refuerzos. Como eres el mejor programador de computación concurrente de la isla, te han pedido que programes una simulación de esta escena para saber en qué casos los ladrones logran el objetivo y para saber en qué casos el botín se mantiene a salvo.

Como los ayudantes de tu curso de programación concurrente son tan buena onda, ya te dieron el esqueleto para la simulación, simplemente enfócate en la parte del algoritmo y su funcionamiento correcto, en este PDF se explica todo lo relacionado con el esqueleto y el funcionamiento de la práctica.

La única restricción respecto a la solución de la práctica es que no importa qué pase primero, ya sea que un ladrón encuentre la solución o que el policía termine sus iteraciones, esta debe de interrumpir a todos los demás hilos con la instrucción `Thread.interrupt()` de la clase `Thread` del API de Java, en ambos casos, ya sea que los ladrones encuentren la llave o que el policía los atrape.

Tus ayudantes te han dado las siguientes interfaces, tu trabajo es implementarlas en los archivos correspondientes para pasar las pruebas unitarias, la primer interfaz se llama `Vault` descrita como sigue.

```
public interface Vault {
    void setPassword();
    boolean isCorrectPassword(int possiblePassword);
    boolean wasPasswordFound();
}

public interface Thief {
```

```
        boolean findPassword(Vault vault, int lowerBound,
                             int upperBound);
    }
```

Y por último, se tiene la siguiente clase que representa al policía que es una clase concurrente normal que extiende de la clase `Thread`

```
    public class Policeman extends Thread {

        Vault vault;

        Policeman(Vault vault) {
            //Code.
        }

        @Override
        public void run() {
            //Write here your solution
        }
    }
```

La solución debe de garantizar que no existe ningún problema de condición de carrera ni carrera por los datos entre los ladrones y el policía. **Hint:** Piensa por qué se dividen en rangos los ladrones la búsqueda de la clave.