COMPUTACIÓN CONCURRENTE

PRÁCTICA 4

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

Karla Vargas Godoy
karla.vargas@ciencias.unam.mx

Omar Arroyo Munguía
omar.am@ciencias.unam.mx

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

Fecha de Entrega: 12 de Abril de 2019 a las 23:59:59pm.

1. Objetivo

El objetivo fundamental de esta práctica, es reafirmar los conocimientos adquiridos en clase para sincronizar hilos mediante semáforos y barreras. Para ello implementarás una barrera, asi como también utilizarás una barrera para sincronizar hilos en un caso de uso.

2. Introducción

En esta práctica trabajarás con una base de código construida con Java 8 y Maven 3.

En el código que recibirás junto con este documento, prodrás encontrar la clase `App` que tiene un método *main* que puedes ejecutar como cualquier programa escrito en *Java*. Para eso primero tienes que empaquetar la aplicación y después ejecutar el *jar* generado. Utiliza comandos como los siguientes:

```
$ mvn package
...
...
$ java -jar target/practica04-1.0.jar
```

Recuerda que para ejecutar las pruebas unitarias de la misma es necesario ejecutar el siguiente comando:

```
$ mvn test
```

3. Actividades

Resuleve los siguientes ejercicios:

3.1. Barrera de Árbol Estático

Implementa la clase `StaticTreeBarrier`, que es similar a las barreras vistas en clase pero tiene dos mejoras importantes con respecto a otras barreras, no sufre de contención de memoria y no tiene que recorrer grandes estructuras para funcionar. Es importante mencionar que el árbol que generarás no necesariamente es completo. Para verificar que tu implementación es correcta tendrás que pasar totas las pruebas unitarias contenidas en la clase `BarrierTest`. La descripción de esta barrera la podrás encontrar en el libro *The art of multiprocesor programming* en la página 402.

3.2. El Mundo de las Pelotas

Encontrarás 3 clases:

1. `Ball`, una instancia de esta clase representará una pelota que vive en el `BallWorld`. La clase implementa la interfaz `Runnable` y ejecuta un *ciclo* infinito en el método *run* donde la pelota repetidamente realiza un movimiento que actualiza su posición. Cada invocación fuerza a repintar el mundo y esperar por un periodo de tiempo de 40ms. La pelota también tiene la habilidad de pintarse a sí misma dado el contexto gráfico. Po lo tanto, una vez que se crea, cada pelota se registra a sí misma en el mundo en donde vive invocando al método *addBall*.

2. `BallWorld`, una instancia de esta clase representa el mundo en donde viven las pelotas y puede contener muchas de ellas en un `List`. El mundo es una subclase de un `JPanel`. Por consecuencia, el mundo se dibuja a sí mismo, utilizando el método *paintComponent* que a su vez le indica a cada pelota que debe de dibujarse. Los detalles para pintar un componente no son importantes en este ejercicio.

3. `App`, esta clase contiene el método principal, el cual crea un mundo, genera y añade algunas pelotas a él y finalmente arranca la ejecución de la simulación.

Hay que notar que le flujo de control de este ejercicio es un poco sutil. El movimiento de las pelotas y el repintado se inicia de forma independiente por cada una de las pelotas, ya que cada una de ellas se ejecuta en un hilo independiente. Debido a esto distintas pelotas pueden ejecutar el método *makeMovement* y *world.repaint()*, de manera concurrente. Invocar al método *world.repaint* de manera concurrente no crea problemas, porque lo garantiza la documentación de `Swing`. Por otro lado invocar al método *repaint()* también desencadena invocar al método *draw* de cada una de las pelotas registradas en el mundo. Sin embargo esto también es correcto, ya que ambos métodos *draw* y *makeMovement* son sincronizados, que significa que no se ejecutarán de manera concurrente en un objeto dado. Como se puede ver, este programa sencillo tiene una concurrencia complicada. Es aceptable tener un diseño como este para programas sencillos, pero para cosas más grandes el comportamiento concurrente debe ser diseñado de una manera estructurada, o de lo contrario será muy complicado.

3.2.1. Matando las pelotas

Tu primera tarea es agregar un nuevo hilo a tu programa que se encarge de matar a las pelotas tanto en tiempos y orden aleatorios. Matar una pelota significa que su método *run* debe de terminar. Además, la pelota debe de ser retirada del mundo (de una manera segura para los subprocesos). Después de realizar esto, el recolector de basura en el sistema eventualmente recuperará el espacio asignado para el objeto pelota.

Modifíca la clase `App` para que el hilo que mata a los demás solamente se active cuando la aplicación recibe la opción *enableKiller*, con un comando como el que sigue:

```
$ java -jar target/practica04-1.0.jar --enableKiller
```

Para resolver el problema, puedes hacer uso de un semáforo que las bolas deben de tratar de adquirir con el fin de *obtener el permiso para morir*. El semáforo se inicializa en cero y luego se libera un número de veces en el método principal main. Ten en cuenta que para tratar de adquirir el semáforo el método *tryAcquire* es muy útil. Implementa y prueba tu programa. Asegúrate de que entiendes por qué el orden de matar es impredecible. Si lo deseas, puedes cambiar el comportamiento del programa, para que una pelota renazca después de un tiempo aleatorio, para que puedas apreciar mejor el comportamiento sin necesidad de reiniciar el programa.

3.2.2. Congelando las pelotas

Ahora debes modificar el programa para que tenga el siguiente comportamiento: Si una pelota que rebota después de algunos movimientos se encuentra en la zona diagonal del mundo, es decir, la pelota está a una distancia menor a su radio de la recta $y = x$, entonces se *congele*, es decir, deja de moverse . Ten en cuenta que una pelota puede saltar por encima de la zona en diagonal en un solo movimiento; lo no causará que se congele. Cuando todas las bolas se han congelado en la diagonal, todas se despiertan y siguen rebotando hasta que se congelan de nuevo en la diagonal. Este ciclo de rebotar y congelarse continúa para siempre. Para resolver este ejercicio deberás de utilizar la barrera que implementaste en ***2.1***.