Laboratorio 02 ARSW: Programación Concurrente - Carrera de Galgos

María Paula Rodríguez Muñoz
Juan Andrés Suárez
Juan Pablo Nieto
Tomás Felipe Ramírez

1. Objetivo

El objetivo de este laboratorio es analizar, corregir y diseñar una solución concurrente para una simulación de carrera de galgos, identificando problemas de sincronización y regiones críticas.

Los objetivos específicos incluyen:

- Corregir la aplicación para que los resultados se muestren únicamente cuando todos los hilos hayan finalizado.
- Identificar las regiones críticas que generan inconsistencias en el ranking.
- Implementar funcionalidades de pausa (Stop) y continuación (Continue) usando `wait()` y `notifyAll()`.
- Aplicar mecanismos de sincronización mediante un monitor común.

2. Contexto del Problema

La simulación presenta las siguientes características:

- Cada **galgo** corre de manera concurrente (un hilo por galgo).
- Todos los galgos comparten un **registro de llegada**.
- El sistema permite **iniciar**, **detener** y **reanudar** la carrera.
- Al finalizar, se debe mostrar el **orden de llegada** de forma consistente.

3. Problemas Iniciales Identificados

1. **Condiciones de carrera:** Los 17 galgos accedían a `nextPosition` y `winner` al mismo tiempo, causando posiciones duplicadas.
2. **Resultados prematuros:** Se mostraba el ganador antes de que todos los galgos terminaran (faltaba `join()`).
3. **Sin control de pausa:** No había forma de pausar/reanudar la carrera (faltaba `wait()`/`notifyAll()`).

4. Regiones Críticas

4.1. ArrivalRegistry (Registro de llegada)

Problema: Varios galgos leen `nextPosition` antes de incrementarlo → misma posición para varios.

Solución: `synchronized` para que solo un hilo ejecute a la vez:

```
public synchronized ArrivalSnapshot registerArrival(String dogName) {
    final int position = nextPosition++;
    if (position == 1) { winner = dogName; }
    return new ArrivalSnapshot(position, winner);
}
```

4.2. RaceControl (Control de pausa)

Problema: El UI cambia `paused` mientras los galgos lo leen → no respetan la pausa.

Solución: Monitor común con `wait()`/`notifyAll()`:

```
public void awaitIfPaused() throws InterruptedException {
    synchronized (monitor) {
        while (paused) { monitor.wait(); }  // Galgo espera
    }
}

public void resume() {
    synchronized (monitor) {
        paused = false;
        monitor.notifyAll();  // Despierta a TODOS
```

```
    }
}
```

5. Evidencias de Ejecución

5.1. Ejecución de la Carrera

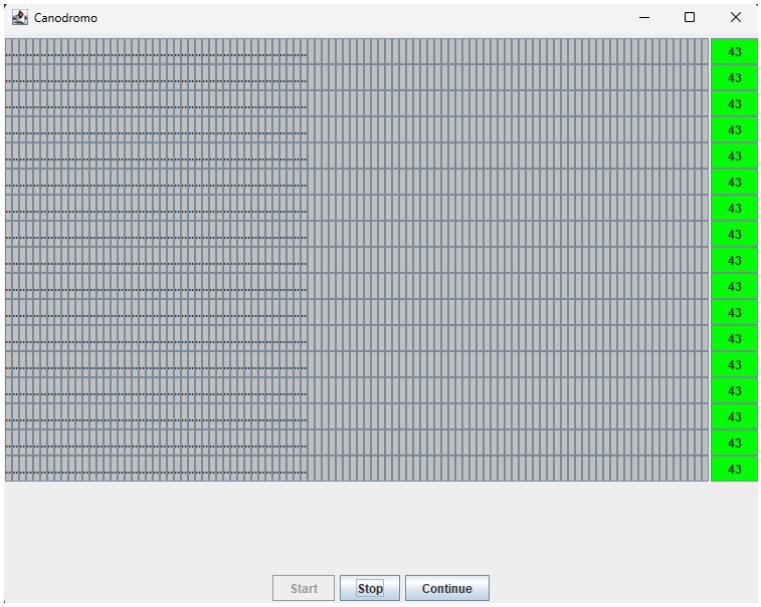

Figura 1: Interfaz gráfica durante la ejecución de la carrera

5.2. Resultado Final

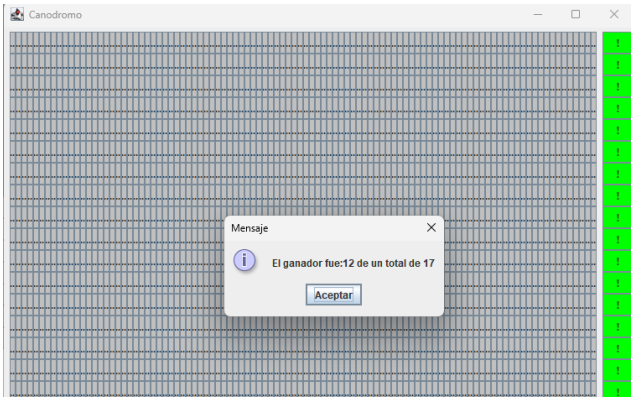

Figura 2: Diálogo mostrando el ganador después de que todos los hilos finalizaron

5.3. Logs de Consola

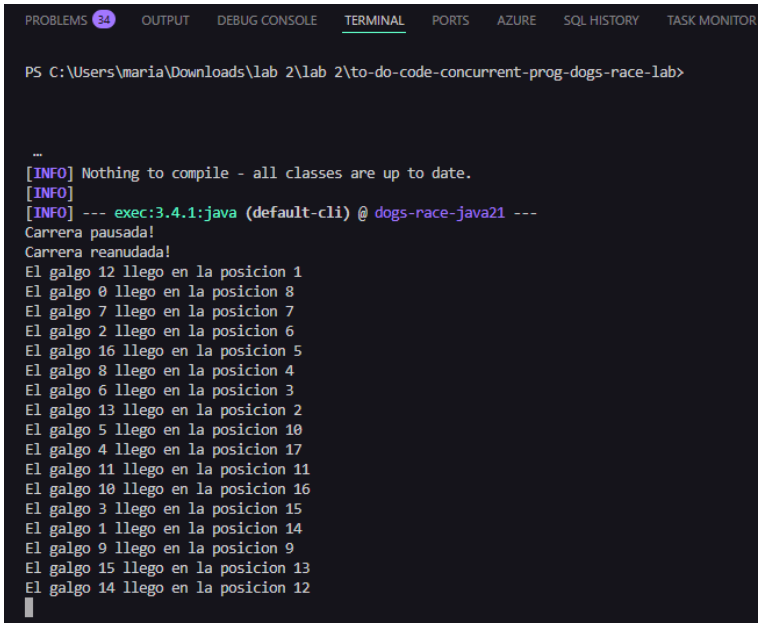

Figura 3: Salida de la consola mostrando los logs de ejecución

6. Conclusiones

1. La programación concurrente requiere identificar cuidadosamente las regiones críticas donde se accede a recursos compartidos.

2. El mecanismo `synchronized` de Java proporciona exclusión mutua efectiva para proteger regiones críticas.

3. El patrón de monitor con `wait()` y `notifyAll()` permite coordinar la pausa y reanudación de múltiples hilos de manera eficiente.

4. El método `join()` es esencial para sincronizar la finalización de hilos antes de mostrar resultados.

5. Es importante sincronizar solo las regiones críticas necesarias para evitar degradar el rendimiento con sincronización excesiva.