Miguel Ángel Ramírez Pérez mramper0604@g.educaand.es

Explicación - Ejercicio - Contador Concurrente Sincronizado

1. Explicación de Contador v1

En esta primera versión, tenemos dos métodos para incrementar un contador: ***incrementarSinSincronizar()*** y ***incrementar().*** La diferencia principal entre ellos es la seguridad en concurrencia (***thread-safe***)

El problema en *incrementarSinSincronizar()*:
Este método es el ejemplo perfecto donde hay una condición de carrera, ya que es inseguro para ser usado por varios hilos.

Conceptos:

- **Condición de Carrera:** Ocurre cuando dos o más hilos acceden a un recurso (en este caso es la variable ***valor***) y al menos uno de ellos modifica el recurso. El resultado depende del orden aleatorio en el que los hilos se planifican por el sistema operativo.
- **No Atomicidad:** El problema es que la operación ***valor++*** no es atómica. Aunque parece solo una línea de código, desde dentro se divide en tres pasos:
  - **LEER:** El hilo carga el valor actual en la memoria
  - **MODIFICAR:** El hilo suma 1 al valor que se ha leído
  - **ESCRIBIR:** El hilo guarda el nuevo valor de nuevo en la memoria.

¿Qué sucede en el código?

El ***valor*** es 100 y dos hilos (Hilo A y Hilo B) intentan incrementar al mismo tiempo:

1. Hilo A Lee ***valor*** (obtiene 100).
2. El planificador de hilos pausa al Hilo A y da paso al Hilo B.
3. Hilo B Lee ***valor*** (sigue siendo 100, porque A no ha escrito nada).
4. Hilo B Modifica (100 + 1 = 101).
5. Hilo B Escribe 101 en ***valor*** .
6. El planificador vuelve a dar paso al Hilo A.
7. Hilo A Modifica (100 + 1 = 101, basado en el valor obsoleto que leyó).
8. Hilo A Escribe 101 en ***valor***.

Resultado: Se han realizado dos incrementos, pero el contador solo ha subido en 1. Se ha perdido un incremento. Esto es lo que se observa en ***Main01***, donde el resultado final nunca es 100.000.

La Solución: incrementar()

Conceptos:

1. **Exclusión Mutua:** ***synchronized*** implementa un mecanismo de exclusión mutua. Cada objeto en Java posee un "candado" interno, conocido como monitor lock.
2. **Sección Crítica:** Al declarar un método como ***synchronized***, todo el método se convierte en una sección crítica. Esto significa que solo un hilo puede ejecutar ese método a la vez.

¿Qué sucede en el código?

1. El Hilo A quiere ejecutar ***incrementar()*** y adquiere el "candado" del objeto ***Contador01.***
2. El Hilo A entra en la sección crítica y realiza los tres pasos (Leer, Modificar y Escribir) sin interrupción.
3. Mientras el Hilo A está dentro, el Hilo B también intenta ejecutar ***incrementar()***
4. El Hilo B no puede adquirir el candado (porque lo tiene el Hilo A) y pasa a estado **BLOCKED** (Bloqueado)
5. El Hilo A finaliza el método y **libera el candado**
6. El Hilo B se "despierta" (pasa a estado **RUNNABLE**) e intenta adquirir el candado
7. El Hilo B lo consigue, ejecuta la operación completa y libera el candado

De esta forma, se asegura que la operación ***valor++*** sea atómica en la práctica, y el resultado de ***Main02*** es siempre 100000

2. **Comparación: synchronized (v1) vs. AtomicInteger (v2)**
Ambos métodos consiguen el resultado correcto, pero sus acciones y rendimiento son diferentes.

Diferencias:

Características	**synchronized (Contador v1.0)**	**AtomicInteger (Contador v2.0)**
Analogía	Es como un baño con un candado. Solo una persona (hilo) puede entrar a la vez. Los demás deben esperar fuera haciendo cola.	Es como una pescadería. Varios hilos intentan coger el siguiente número a la vez, pero el sistema garantiza que solo uno lo consigue. Si fallas, lo intentas al momento.
¿Qué hace el Hilo?	Pide permiso (al candado). Si está ocupado, el Sistema Operativo lo "duerme" (estado BLOCKED) hasta que el candado se libera.	Intenta la operación directamente. Si falla (porque otro hilo se adelantó), no se duerme, simplemente lo vuelve a intentar muy rápido.
Coste / Impacto	"Pesado". Pone un hilo a "dormir" y "despertarlo" consume tiempo y recursos del Sistema Operativo.	"Ligero". Reintentar la operación es muy rápido y no implica al Sistema Operativo.

Mejor Uso	Ideal para proteger "secciones grandes" (bloques de código largos o que modifican varias variables a la vez).	Ideal para proteger "valores únicos" (un simple número, un contador).
Rendimiento	Más lento si muchos hilos compiten por el mismo candado. Se forma un "cuello de botella" con hilos durmiendo.	Mucho más rápido y eficiente, especialmente cuando muchos hilos compiten (alta contención).

¿Cuál prefiero y por qué?

Para este ejercicio, prefiero sin dudarlo la versión **Contador v2** con ***AtomicInteger***. La razón principal es el rendimiento ya que ***AtomicInteger*** utiliza operaciones de hardware que son mucho más rápidas y eficientes que el mecanismo de bloqueo de ***synchronized***, el cual introduce un alto coste al suspender y reactivar hilos. Gracias a esto, ***AtomicInteger*** escala mucho mejor. Además, su uso ***(incrementAndGet())*** expresa de forma más clara el realizar un incremento.