



Parte III — (Avance) Sincronización y *Deadlocks* con *Highlander Simulator*

1. Revisa la simulación: N inmortales; cada uno **ataca** a otro. El que ataca **resta M** al contrincante y **suma M/2** a su propia vida.
2. **Invariante:** con N y salud inicial H, la suma total debería permanecer constante (salvo durante un update). Calcula ese valor y úsalo para validar.

Respuesta:

Invariante

$$In0 = N * H$$

Cuando ocurre un ataque:

Cuando un inmortal A ataca a otro B:

- B pierde M
- A gana M/2

Entonces el cambio total en la suma de vida es:

$$\Delta In = (+M/2) + (-M) = -M/2$$

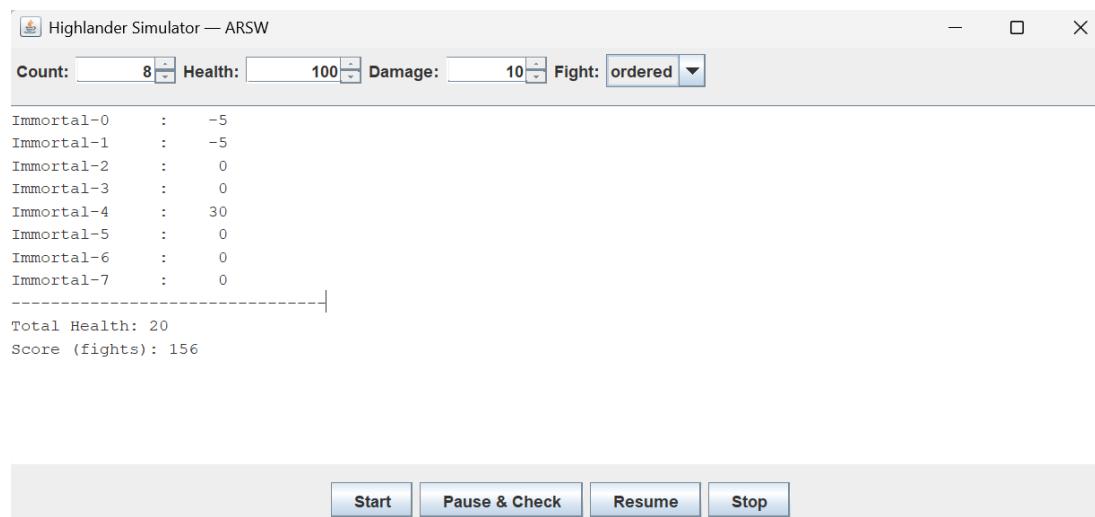
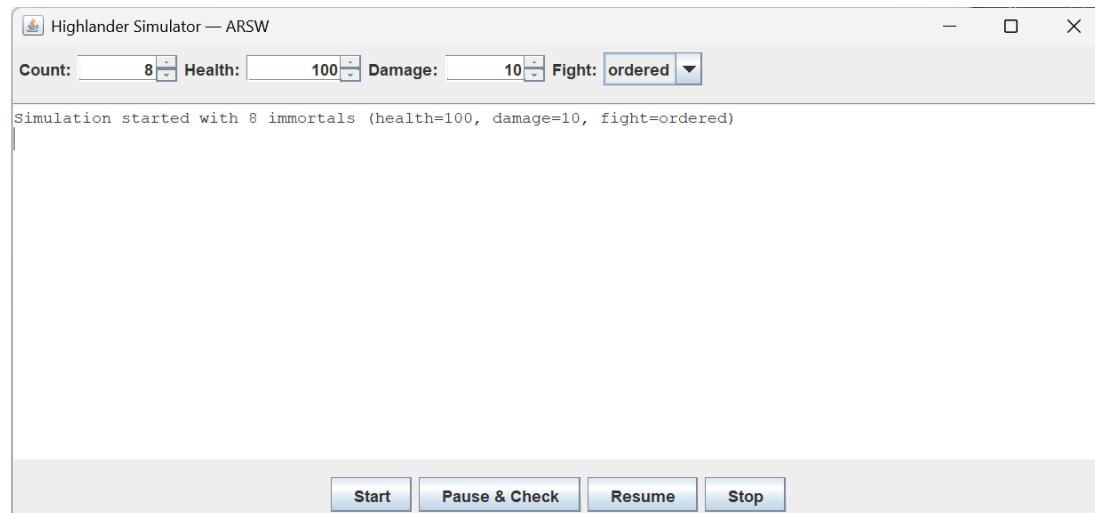
Esto significa que cada ataque completo reduce la vida total del sistema en M/2.

Fuera del update, el sistema debe cumplir esta relación:

$$VT = N * H - (\text{atques completos}) * \frac{M}{2}$$

3. Ejecuta la UI y prueba “Pause & Check”. ¿Se cumple el invariante? Explica.

Respuesta:



Datos:

- N = 8

- H = 100

- M = 10

$$VT = 800 - 156 * 5$$

$$VT = 20$$

Así que la invariante global se cumpliría matemáticamente.

Como se está en ordered:

- El sistema sincroniza el acceso.
- Pero no válida correctamente que:
 - un inmortal muerto no ataque.
 - un inmortal no quede con vida negativa.
- Los hilos siguen ejecutándose incluso cuando ya no deberían

4. **Pausa correcta:** asegura que **todos** los hilos queden pausados **antes** de leer/imprimir la salud; implementa **Resume** (ya disponible).

Respuesta: Sí, la pausa es correcta; todos los hilos quedan detenidos antes de leer/imprimir la salud.

```

private final Condition unpause = lock.newCondition();
private final Condition allPaused = lock.newCondition();

private volatile boolean paused = false;
private int pausedThreads = 0;
private int totalThreads = 0;

public void pause() { lock.lock(); try { paused = true; } finally { lock.unlock(); } }
public void resume() { lock.lock(); try { paused = false; pausedThreads=0; unpause.signalAll(); } finally { lock.unlock(); } }
public void setTotalThreads(int totalThreads) {
    lock.lock();
    try { this.totalThreads = totalThreads; }
    finally { lock.unlock(); }
}
public void awaitIfPaused() throws InterruptedException {
    lock.lock();
    try {
        while (paused){
            pausedThreads++;
            if (pausedThreads == totalThreads) {
                unpause.signalAll();
            }
            unpause.await();
            pausedThreads--;
        }
    }finally { lock.unlock(); }
}
public void awaitAllPaused() throws InterruptedException {
    lock.lock();
    try {
        while (pausedThreads < totalThreads) {
            allPaused.await();
        }
    } finally {
        lock.unlock();
    }
}

```

5. Haz *click* repetido y valida consistencia. ¿Se mantiene el invariante?

Respuesta: Al hacer múltiples clicks en *Pause & Check* sin reanudar la simulación, los valores permanecen constantes. Esto indica que la pausa es correcta y que el invariante global se mantiene.

6. **Regiones críticas:** identifica y sincroniza las secciones de pelea para evitar carreras; si usas múltiples *locks*, anida con **orden consistente**:

Respuesta:

```
private void fightOrdered(Immortal other) {
    Immortal first = this.name.compareTo(other.name) < 0 ? this : other;
    Immortal second = this.name.compareTo(other.name) < 0 ? other : this;
    synchronized (first) {
        synchronized (second) {
            if (this.health <= 0 || other.health <= 0) return;
            other.health -= this.damage;
            this.health += this.damage / 2;
            scoreBoard.recordFight();
        }
    }
}
```

El uso de locks es necesario para proteger las regiones críticas donde se modifica el estado compartido de los inmortales. Los locks garantizan exclusión mutua, evitan condiciones de carrera y mantienen la consistencia del sistema. Al adquirir los locks en un orden consistente, se previenen deadlocks sin sacrificar concurrencia.

7. Si la app se **detiene** (posible *deadlock*), usa **jps** y **jstack** para diagnosticar.

Respuesta: Si la aplicación se queda congelada, podemos revisar si hay un deadlock usando jps para encontrar el número de proceso de la app y jstack para ver qué están haciendo los hilos. El informe de jstack muestra claramente si hay hilos que están esperando locks unos de otros, es decir, un deadlock. Esto suele pasar en el modo *naive*, donde los locks se toman sin un orden consistente. En el modo *ordered*, como los locks se adquieren siempre siguiendo un mismo orden, se evita que los hilos se bloqueen entre sí.

8. Aplica una **estrategia** para corregir el *deadlock* (p. ej., **orden total** por nombre/id, o **tryLock(timeout)** con reintentos y *backoff*).

Respuesta:

```
private void fightOrdered(Immortal other) throws InterruptedException {
    Immortal first = this.name.compareTo(other.name) < 0 ? this : other;
    Immortal second = this.name.compareTo(other.name) < 0 ? other : this;

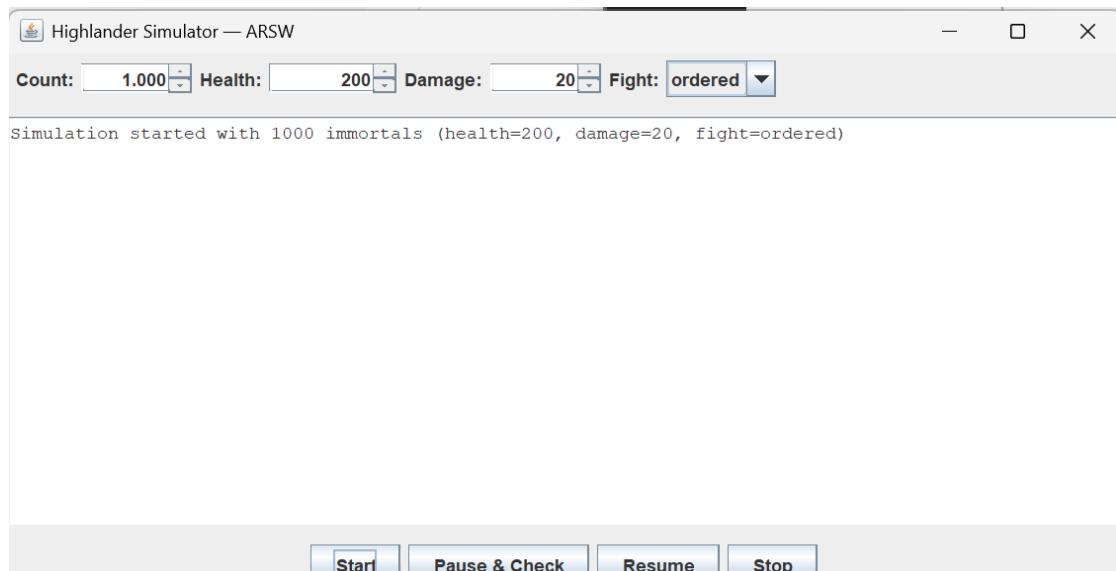
    boolean success = false;
    while (!success && running) {
        if (first.lock.tryLock(timeout: 100, TimeUnit.MILLISECONDS)) {
            try {
                if (second.lock.tryLock(timeout: 100, TimeUnit.MILLISECONDS)) {
                    try {
                        if (this.health <= 0 || other.health <= 0) return;
                        other.health -= this.damage;
                        this.health += this.damage / 2;
                        scoreBoard.recordFight();
                        success = true;
                    } finally {
                        second.lock.unlock();
                    }
                }
            } finally {
                first.lock.unlock();
            }
        }
        if (!success) Thread.sleep(milliseconds: 5);
    }
}
```

Así que al utilizar **tryLock**:

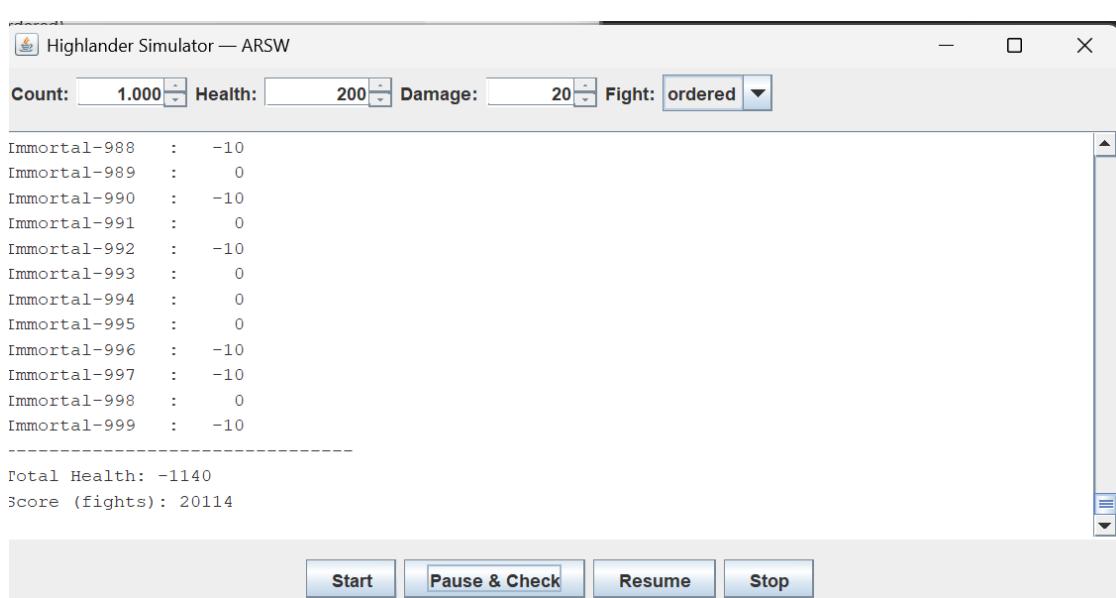
- Cada hilo intenta tomar los locks con un **timeout**.
- Si no puede, libera los que ya tiene y espera un poquito antes de volver a intentar.
- Esto evitaría posibles deadlocks dinámicamente, incluso si los hilos intentan tomar locks en distinto orden.

9. Valida con **N=100, 1000 o 10000** inmortales. Si falla el invariante, revisa la pausa y las regiones críticas.

Respuesta: Ahora procedemos a probar con N = 1000 y verificamos que se siga cumpliendo la invariante.



The screenshot shows the 'Highlander Simulator — ARSW' window. At the top, there are input fields for 'Count' (set to 1.000), 'Health' (set to 200), 'Damage' (set to 20), and 'Fight' (set to 'ordered'). Below the inputs, a message states: 'Simulation started with 1000 immortals (health=200, damage=20, fight=ordered)'. At the bottom, there are four buttons: 'Start', 'Pause & Check', 'Resume', and 'Stop'.



The screenshot shows the 'Highlander Simulator — ARSW' window with the same parameter settings as the first screenshot. The log area displays the health status of the last 12 immortals: Immortal-988, -989, -990, -991, -992, -993, -994, -995, -996, -997, -998, and -999. Each immortal has a health value of -10, except for Immortal-989 which has 0 health. Below the list, it shows 'Total Health: -1140' and 'Score (fights): 20114'. At the bottom, there are four buttons: 'Start', 'Pause & Check', 'Resume', and 'Stop'.

Datos:

- N = 1000

- $H = 200$

- $M = 20$

$$VT = 200.000 - 201.140$$

$$VT = -1.140$$

Cumple la invariante.

10. **Remover inmortales muertos** sin bloquear la simulación: analiza si crea una **condición de carrera** con muchos hilos y corrige **sin sincronización global** (colección concurrente o enfoque *lock-free*).

Respuesta: Se aplicaron los siguientes cambios:

- `population` es thread-safe, se puede iterar y remover sin sincronización global.
- Los inmortales muertos se eliminan durante el cálculo de `totalHealth` sin bloquear la simulación.
- Se usa `ConcurrentLinkedQueue` porque en tu simulación hay muchos hilos accediendo y modificando la lista de inmortales al mismo tiempo.

```
54
55     @Override
56     public void run() {
57         try {
58             while (running) {
59
60                 controller.awaitIfPaused();
61                 if (!running) break;
62
63                 Immortal opponent = pickOpponent();
64                 if (opponent == null || !opponent.isAlive()) continue;
65
66                 fightOrdered(opponent);
67
68                 if (getHealth() <= 0) {
69                     running = false;
70                     population.remove(this);
71                     break;
72                 }
73
74                 Thread.sleep(millis: 2);
75             }
76         } catch (InterruptedException e) {
77             Thread.currentThread().interrupt();
78         }
79     }
```

```

    public final class ImmortalManager implements AutoCloseable {
        private final List<Future<?>> futures = new ArrayList<()>();
        private final PauseController controller = new PauseController();
        private final ScoreBoard scoreBoard = new ScoreBoard();
        private final ExecutorService exec;

        private final ConcurrentLinkedQueue<Immortal> population = new ConcurrentLinkedQueue<()>();

        private final String fightMode;
        private final int initialHealth;
    }

```

11. Implementa completamente STOP (apagado ordenado).

Respuesta:

Características del STOP implementado:

- Marca todos los inmortales como detenidos (running = false).
- Llama a shutdown() del executor y espera con awaitTermination.
- Si algún hilo no termina en el tiempo límite, fuerza shutdownNow().
- Limpia la lista de futures para liberar.

```

J Immortal.java 3 ● J ImmortalManager.java 6. M ●
src > main > java > edu > eci > arsw > immortals > J ImmortalManager.java > ImmortalManager > stop()
14     public final class ImmortalManager implements AutoCloseable {
30         public ImmortalManager(int n, String fightMode, int initialHealth, int damage) {
37             ...
38         }
39         public synchronized void start() {
40             if (exec != null) stop();
41             exec = Executors.newVirtualThreadPerTaskExecutor();
42             for (Immortal im : population) {
43                 futures.add(exec.submit(im));
44             }
45         }
46         public void pause() { controller.pause(); }
47         public void resume() { controller.resume(); }
48         public void stop() {
49             for (Immortal im : population) im.stop();
50             if (exec != null) {
51                 exec.shutdown();
52                 try {
53                     if (!exec.awaitTermination(5, TimeUnit.SECONDS)) {
54                         exec.shutdownNow();
55                     }
56                 } catch (InterruptedException ie) {
57                     exec.shutdownNow();
58                     Thread.currentThread().interrupt();
59                 }
60             }
61             exec = null;
62         }
63         public int aliveCount() {
64             ...
65             futures.clear();
66         }
67     }

```