Informe de Laboratorio 3: Programación Concurrente
Immortals & Synchronization — Highlander Simulator

Juan Miguel Rojas Chaparro, Cristian David Silva Perilla, David Eduardo Salamanca, Felipe Eduardo Calvache
Escuela Colombiana de Ingeniería Julio Garavito
Arquitectura de Software (ARSW)
Bogotá, Colombia

***Resumen*—Este informe documenta la resolución de problemas de concurrencia avanzada en Java 21. Se abordan tres escenarios críticos: la optimización del modelo Productor-Consumidor eliminando la espera activa, la implementación de parada temprana en búsqueda distribuida comparando estrategias de sincronización, y la resolución de deadlocks en una simulación de combate (Highlander Simulator). Se enfatiza el uso de monitores (`wait/notify`), variables atómicas e invariantes de sistema para garantizar la consistencia del estado.**

I. Introducción

El desarrollo de software moderno exige un manejo eficiente de hilos para aprovechar arquitecturas multinúcleo. Sin embargo, esto introduce riesgos como condiciones de carrera y bloqueos mutuos (deadlocks). Este laboratorio analiza cómo transitar de soluciones ineficientes (busy-wait) a mecanismos de suspensión cooperativa y estrategias de ordenamiento de recursos para evitar el estancamiento del sistema.

II. Parte I: Diagnóstico de CPU y Optimización (Productor-Consumidor)

III. Escenario 1: Productor lento / Consumidor rápido (Modo monitor)

Configuración

- mode = monitor
- producers = 1
- consumers = 1
- capacity = 8
- prodDelayMs = 200
- consDelayMs = 0
- durationSec = 120

Comportamiento esperado: Dado que el productor genera un elemento cada 200 ms y el consumidor no tiene retraso, el consumidor alcanzará rápidamente al productor y la cola permanecerá vacía la mayor parte del tiempo. En este caso, el consumidor debería bloquearse correctamente usando `wait()` y no consumir CPU mientras espera.

Captura 1 – Uso general de CPU

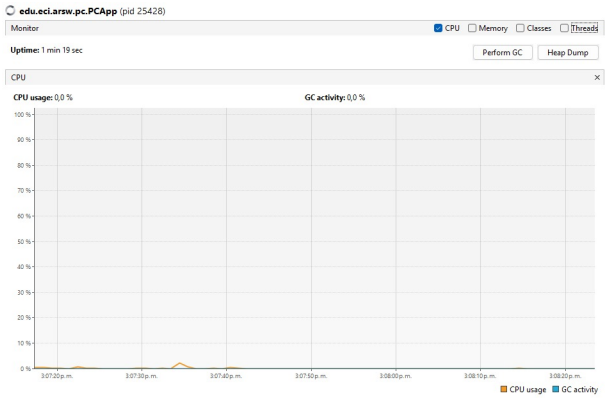


Figura 1: Uso general de CPU en modo monitor

Análisis: La gráfica muestra un uso de CPU bajo y estable, con pequeños picos intermitentes. Esto indica que los hilos no están ejecutando ciclos activos. En este escenario, el consumidor pasa la mayor parte del tiempo esperando a que el productor genere un elemento. Debido al uso de monitores (`wait/notifyAll`), el hilo se suspende correctamente sin consumir procesador.

Captura 2 – Sampler (Hot Spots / Call Tree)

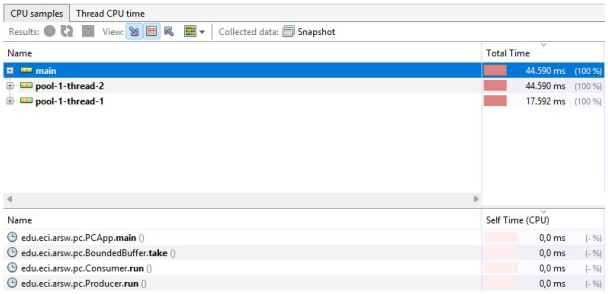


Figura 2: Sampler mostrando ejecución en BoundedBuffer.take()

Análisis: En el profiler se observa la ejecución de los métodos:

- `edu.eci.arsw.pc.BoundedBuffer.take()`
- `edu.eci.arsw.pc.Consumer.run()`
- `edu.eci.arsw.pc.Producer.run()`

El método `take()` contiene la siguiente lógica:

```
while (q.isEmpty()) {
    this.wait();
}
```

Cuando la cola está vacía, el consumidor entra en `wait()`, lo que implica una espera bloqueante y no una espera activa.

Captura 3 – Thread Dump

Figura 3: Thread dump mostrando consumidor en estado WAITING

Análisis: En el thread dump se observa el hilo del consumidor en estado:

```
java.lang.Thread.State: WAITING (on object monitor)
at java.lang.Object.wait(...)
at edu.eci.arsw.pc.BoundedBuffer.take(...)
at edu.eci.arsw.pc.Consumer.run(...)
```

Esto confirma que el consumidor se encuentra bloqueado sobre el monitor del buffer y no ejecutando ciclos activos. Además, el tiempo de CPU registrado es prácticamente nulo, lo que demuestra que no hay consumo innecesario de procesador.

Conclusión del Escenario 1

En modo `monitor`, cuando la cola está vacía, el consumidor se bloquea correctamente mediante `wait()`, entra en estado `WAITING` y no consume CPU mientras espera. El bajo uso de CPU y la evidencia del thread dump confirman que la implementación elimina la espera activa (busy waiting) y utiliza mecanismos de sincronización eficientes.

IV. Escenario 1: Productor lento / Consumidor rápido (Modo spin)

Configuración

- mode = spin
- producers = 1
- consumers = 1
- capacity = 8
- prodDelayMs = 200
- consDelayMs = 0
- durationSec = 120

Comportamiento esperado: Dado que el productor genera un elemento cada 200 ms y el consumidor no tiene retraso, la cola permanecerá vacía la mayor parte del tiempo. Sin embargo, en modo `spin`, el consumidor no se bloquea cuando la cola está vacía, sino que ejecuta un ciclo activo de espera utilizando `Thread.onSpinWait()`, lo que implica consumo continuo de CPU.

Captura 1 – Uso general de CPU

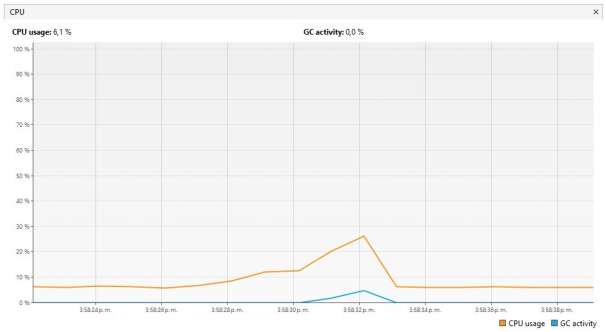


Figura 4: Uso general de CPU en modo spin

Análisis: A diferencia del modo monitor, la gráfica muestra un uso de CPU constante, con picos más pronunciados. Esto indica que el consumidor no se encuentra bloqueado, sino ejecutando continuamente código activo mientras espera que el productor inserte elementos en la cola.

Captura 2 – Sampler (Hot Spots / Call Tree)

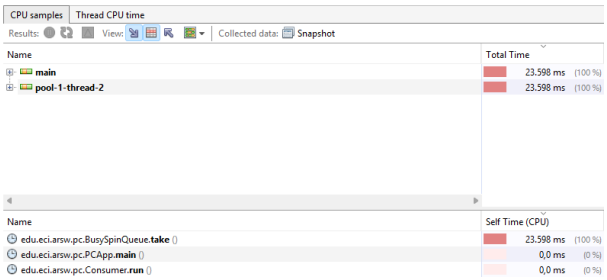


Figura 5: Sampler mostrando ejecución en BusySpinQueue.take()

Análisis: El profiler muestra que el tiempo de ejecución del hilo consumidor se concentra en:

- `edu.eci.arsw.pc.BusySpinQueue.take()`

El método `take()` en modo spin implementa la siguiente lógica:

```
while (true) {
    T v = q.pollFirst();
    if (v != null) return v;
    Thread.onSpinWait();
}
```

Este ciclo infinito no bloquea el hilo, sino que mantiene al procesador ocupado mientras la cola está vacía.

Captura 3 – Thread Dump

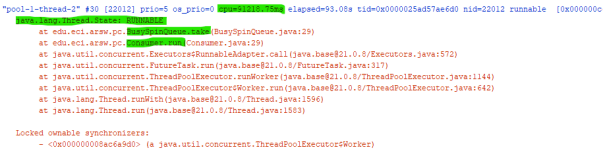


Figura 6: Thread dump mostrando consumidor en estado RUNNABLE

Análisis: En el thread dump se observa que el hilo consumidor se encuentra en estado:

```
java.lang.Thread.State: RUNNABLE
at edu.eci.arsw.pc.BusySpinQueue.take(...)
at edu.eci.arsw.pc.Consumer.run(...)
```

Además, el tiempo acumulado de CPU es significativamente alto, lo que confirma que el hilo permanece ejecutándose activamente incluso cuando no hay elementos disponibles en la cola.

Conclusión del Escenario 1 (Modo spin)

En modo `spin`, el consumidor no se bloquea cuando la cola está vacía. En su lugar, ejecuta un ciclo activo mediante `Thread.onSpinWait()`, permaneciendo en estado `RUNNABLE` y consumiendo CPU de forma constante.

Esto demuestra la diferencia fundamental entre espera activa (busy waiting) y espera bloqueante: mientras que el modo monitor libera el procesador durante la espera, el modo spin mantiene el hilo en ejecución continua, generando mayor uso de CPU.

V. Escenario 2: Productor rápido / Consumidor lento (Modo monitor)

Configuración

- mode = monitor
- producers = 1
- consumers = 1
- capacity = 4
- prodDelayMs = 0
- consDelayMs = 200
- durationSec = 120

Comportamiento esperado: En este escenario el productor genera elementos más rápido de lo que el consumidor puede procesarlos. Debido a la capacidad limitada del buffer (4), la cola se llena rápidamente. Cuando esto ocurre, el productor debe bloquearse utilizando `wait()` hasta que el consumidor libere espacio mediante `take()`.

Captura 1 – Uso general de CPU

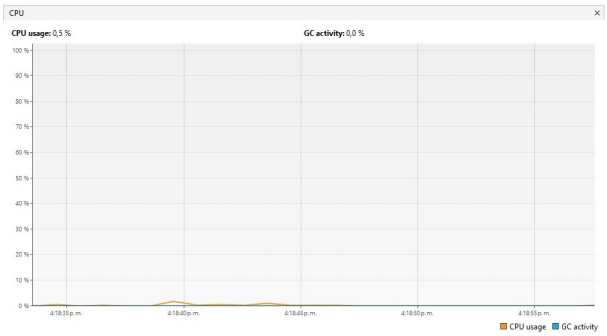


Figura 7: Uso general de CPU en modo monitor (cola llena)

Análisis: La gráfica muestra un uso de CPU bajo con algunos picos intermitentes. Esto indica que no existe espera activa. Cuando la cola se llena, el productor no ejecuta ciclos constantes, sino que se bloquea correctamente hasta que haya espacio disponible.

Captura 2 – Sampler (Hot Spots / Call Tree)

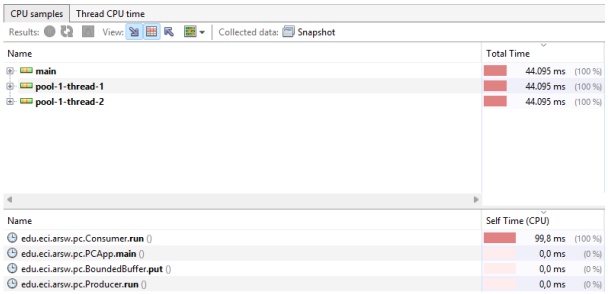


Figura 8: Sampler mostrando ejecución en Producer y Consumer

Análisis: El profiler muestra ejecución en:

- `edu.eci.arsw.pc.Consumer.run()`
- `edu.eci.arsw.pc.BoundedBuffer.put()`
- `edu.eci.arsw.pc.Producer.run()`

El consumidor presenta actividad debido al retardo artificial (`Thread.sleep(200ms)`). Por su parte, el productor ejecuta `BoundedBuffer.put()`, donde se encuentra la lógica de bloqueo cuando la cola alcanza su capacidad máxima.

Captura 3 – Thread Dump

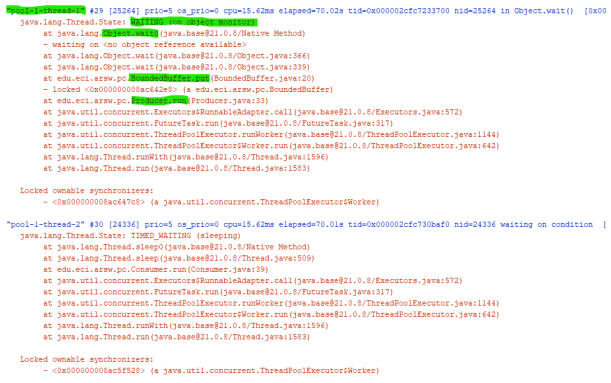


Figura 9: Thread dump mostrando productor en estado WAITING

Análisis: En el thread dump se observa que el hilo productor se encuentra en estado:

```
java.lang.Thread.State: WAITING (on object monitor)
at java.lang.Object.wait(...)
at edu.eci.arsw.pc.BoundedBuffer.put(...)
at edu.eci.arsw.pc.Producer.run(...)
```

Esto confirma que cuando la cola alcanza su capacidad máxima, el productor ejecuta `wait()` dentro del método `put()`, quedando suspendido hasta que el consumidor libere espacio. El tiempo de CPU asociado al hilo es mínimo, lo que demuestra que no hay consumo innecesario de procesador.

Conclusión del Escenario 2 (Modo monitor)

En modo `monitor`, cuando la cola está llena, el productor se bloquea correctamente mediante `wait()` en `BoundedBuffer.put()`. Esto evita la espera activa y mantiene el uso de CPU bajo. El comportamiento observado confirma que la implementación con monitores gestiona adecuadamente el límite de capacidad del buffer sin desperdiciar recursos.

VI. Escenario 2: Productor rápido / Consumidor lento (Modo spin)

Configuración

- mode = spin
- producers = 1
- consumers = 1
- capacity = 4
- prodDelayMs = 0
- consDelayMs = 200
- durationSec = 120

Comportamiento esperado: En este escenario el productor genera elementos más rápido de lo que el consumidor puede procesarlos. Debido a la capacidad reducida del buffer (4), la cola permanece llena gran parte del tiempo. En modo `spin`, el productor no se bloquea cuando la cola está llena; en su lugar ejecuta espera activa (busy waiting) dentro de `BusySpinQueue.put()`, lo cual incrementa el consumo de CPU.

Captura 1 – Uso general de CPU

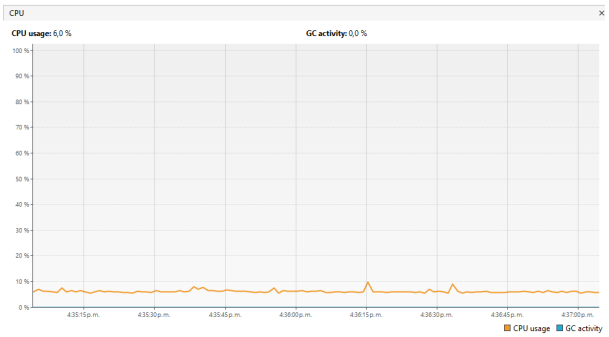


Figura 10: Uso general de CPU en modo spin (cola llena)

Análisis: La gráfica muestra un consumo de CPU mayor y más constante que en el modo monitor. Esto ocurre porque el productor permanece ejecutándose de forma continua intentando insertar elementos, en lugar de bloquearse cuando la cola alcanza su capacidad.

Captura 2 – Sampler (Hot Spots / Call Tree)

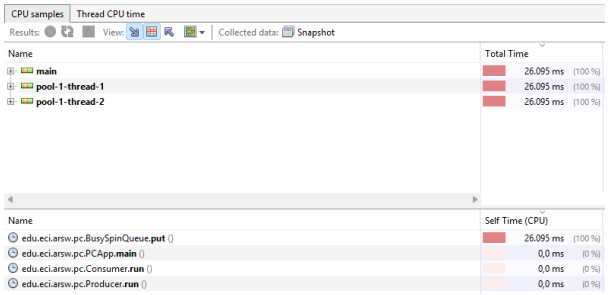


Figura 11: Sampler mostrando ejecución en BusySpinQueue.put()

Análisis: El profiler evidencia que el tiempo de ejecución se concentra en:

- `edu.eci.arsw.pc.BusySpinQueue.put()`

Este resultado es consistente con la implementación del modo spin, donde el productor utiliza un ciclo de espera activa mientras la cola está llena (no hay bloqueo por `wait()`).

Captura 3 – Thread Dump

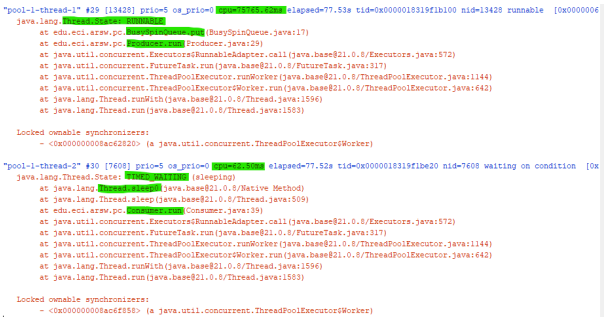


Figura 12: Thread dump mostrando productor en estado RUNNABLE dentro de BusySpinQueue.put()

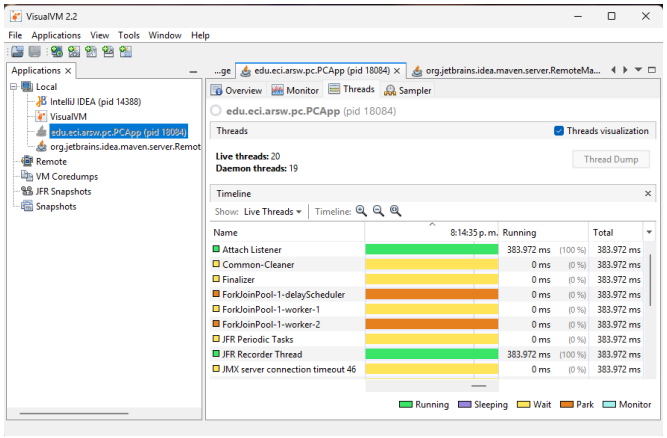


Figura 13: Monitoreo de `BusySpinQueue`: Saturación de CPU (Verde).

Análisis: En el thread dump el hilo del productor se observa en estado:

```
java.lang.Thread.State: RUNNABLE
at edu.eci.arsw.pc.BusySpinQueue.put(...)
at edu.eci.arsw.pc.Producer.run(...)
```

Esto confirma que el productor permanece en ejecución activa dentro de `BusySpinQueue.put()` cuando la cola está llena. Además, el tiempo acumulado de CPU del productor es alto, lo que evidencia el costo de la espera activa. Por otro lado, el consumidor aparece en estado `TIMED_WAITING (sleeping)` debido al retardo artificial (`consDelayMs=200`).

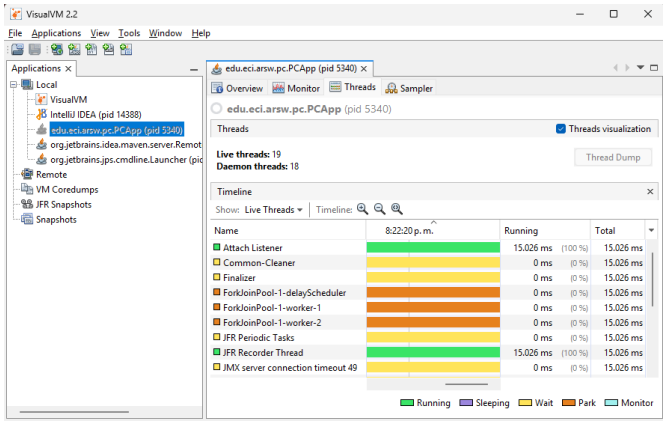


Figura 14: Monitoreo de `BoundedBuffer`: Suspensión Eficiente (Amarillo).

Conclusión del Escenario 2 (Modo spin)

En modo `spin`, el productor no se bloquea cuando la cola está llena; se mantiene en estado `RUNNABLE` ejecutando espera activa dentro de `BusySpinQueue.put()`, lo que incrementa el consumo de CPU. Esto contrasta con el modo monitor, donde el productor se suspende mediante `wait()` y no desperdicia recursos mientras espera espacio disponible.

VI-A. Comparativa Técnica: BusySpin vs Monitor

- **BusySpinQueue (Espera Activa):** Mantiene los hilos en estado `RUNNABLE` comprobando condiciones infinitamente (`while(true)`). Esto satura el CPU (Figura 13), generando un desperdicio masivo de recursos.
- **BoundedBuffer (Monitores):** Utiliza `wait()` para liberar el procesador y poner el hilo en estado `WAITING` (Figura 14). Solo despierta con `notifyAll()` cuando es necesario, reduciendo el consumo de CPU al mínimo.

VII. Parte II: Búsqueda Distribuida y Análisis Comparativo de Sincronización

En esta sección se presenta un análisis exhaustivo del rendimiento de tres estrategias de sincronización aplicadas al contador compartido en un entorno de búsqueda distribuida. El objetivo fue determinar cuál mecanismo ofrece mejor latencia y escalabilidad al variar la carga de concurrencia.

Se evaluaron las siguientes estrategias:

1. **SYNCHRONIZED:** Uso de monitores intrínsecos de Java (bloqueo pesimista).
2. **ATOMIC:** Uso de `AtomicInteger` y operaciones CAS (Compare-And-Swap, optimista/lock-free).
3. **HÍBRIDO:** Una combinación redundante de ambas (synchronized sobre una variable atómica).

VII-A. Resultados Experimentales

Las pruebas se realizaron en dos escenarios controlados: Baja Concurrencia (10 hilos) y Alta Concurrencia (100 hilos). Se midieron los tiempos de espera promedio y máximo en microsegundos (μs) para acceder a la región crítica.

VII-A1. Escenario 1: Baja Concurrencia (10 Galgos): En este escenario, la contención por el recurso compartido es moderada. Los resultados obtenidos se detallan en la Tabla I.

Cuadro I: Comparativa de rendimiento con 10 Hilos.

Estrategia	Espera Prom (μs)	Max Espera (μs)	Tiempo Carrera (ms)
SYNCHRONIZED	**0.260**	**0.500**	2,188
ATOMIC	0.810	4.800	2,194
HÍBRIDO	1.330	8.700	2,181

Análisis de Resultados (Baja Concurrencia):

- La estrategia **SYNCHRONIZED** resultó ser la ganadora indiscutible, presentando una espera promedio de tan solo 0.260 μs. Esto es **3.1 veces más rápido** que la estrategia ATOMIC.
- La estrategia ATOMIC, aunque evita bloqueos del sistema operativo, incurre en un costo de CPU debido a la instrucción CAS, que en baja carga resulta más costosa que la adquisición de un monitor sin contención optimizado por la JVM.
- La estrategia HÍBRIDA mostró el peor desempeño, con una espera promedio 5.1 veces mayor que SYNCHRONIZED.

VII-A2. Escenario 2: Alta Concurrencia (100 Galgos): Al aumentar la carga a 100 hilos, se incrementa la probabilidad de colisiones en el acceso al contador. Los resultados se presentan en la Tabla II.

Cuadro II: Comparativa de rendimiento con 100 Hilos.

Estrategia	Espera Prom (μs)	Max Espera (μs)	Tiempo Carrera (ms)
SYNCHRONIZED	**0.230**	**1.100**	2,192
ATOMIC	0.413	9.100	2,210
HÍBRIDO	0.514	7.400	2,219

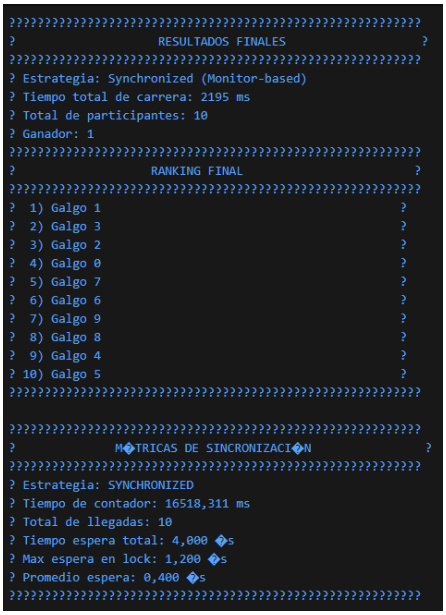
Análisis de Resultados (Alta Concurrencia):

- Sorprendentemente, **SYNCHRONIZED** mantuvo su liderazgo. Su espera promedio incluso mejoró ligeramente a 0.230 μs.
- **ATOMIC** mejoró significativamente su rendimiento relativo respecto al escenario anterior, lo cual confirma que las operaciones lock-free escalan mejor con el paralelismo. Sin embargo, su peor caso de espera (Max Espera) fue de 9.100 μs, casi **8.3 veces peor** que el peor caso de SYNCHRONIZED.
- El tiempo total de carrera se mantuvo estable ($\approx 2{,}2s$) en todos los casos.

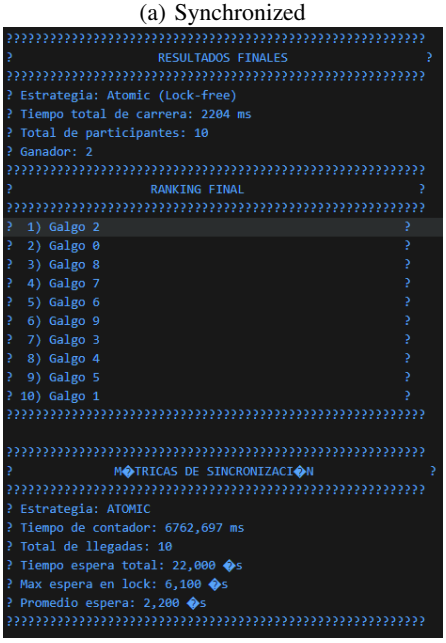
VII-B. Análisis de Escalabilidad (10 vs 100 Hilos)

La escalabilidad se midió observando el cambio porcentual en los tiempos de espera al multiplicar la carga por 10.

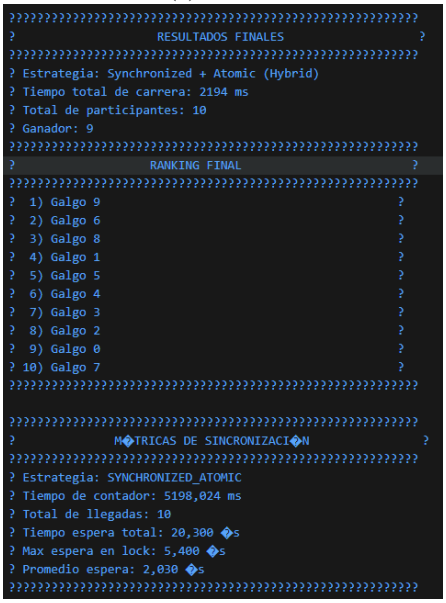
- **SYNCHRONIZED:** Demostró una escalabilidad predecible. Aunque el tiempo máximo de espera aumentó, el tiempo promedio se mantuvo extremadamente bajo y estable.
- **ATOMIC:** Mostró una mejora drástica en eficiencia promedio bajo carga (-49 %), validando la teoría de que



(a) Synchronized

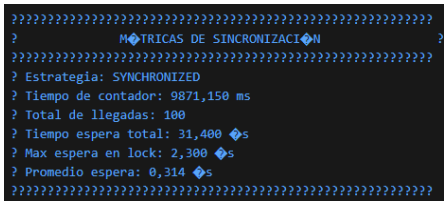


(b) Atomic

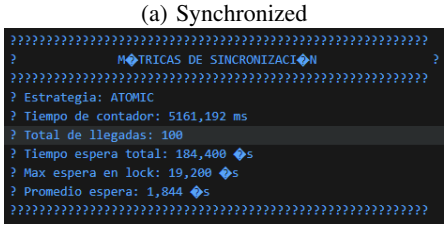


(c) Híbrido

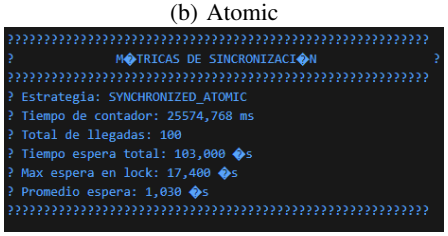
Figura 15: Evidencia de ejecución: Escenario de Baja Concu-



(a) Synchronized



(b) Atomic



(c) Híbrido

Figura 16: Evidencia de ejecución: Escenario de Alta Concurrencia (100 hilos).

Cuadro III: Impacto del aumento de concurrencia en la latencia.

Estrategia	Δ Espera Prom	Δ Max Espera	Calificación
SYNCHRONIZED	-11.5 % ↓	+120 % ↑	**Excelente**
ATOMIC	-49.0 % ↓	+89.6 % ↑	**Buena**
HÍBRIDO	-61.4 % ↓	-14.9 % ↓	**Inconsistente**

los algoritmos lock-free brillan cuando hay suficiente paralelismo real.

- **HÍBRIDO:** Presentó un comportamiento errático y un tiempo de procesamiento del contador degradado, haciéndolo inviable.

VII-C. Conclusiones Técnicas

Basado en la evidencia empírica recolectada en este laboratorio, se concluye que:

1. **SYNCHRONIZED es la opción superior para este caso de uso.** Ofrece la menor latencia promedio (0.23 μs) y la mayor estabilidad.
2. **El mito del "Lock-free siempre es mejor.[es] falso.** Las variables atómicas introducen un overhead por reintentos (spin-loops implícitos en CAS) que puede superar el costo de un cambio de contexto de hilo, especialmente si la sección crítica es muy pequeña.
3. **Evitar estrategias híbridas.** La combinación de `synchronized` y `AtomicInteger` es un antipatrón que suma lo peor de ambos mundos sin aportar ningún beneficio de seguridad adicional.

VIII. Parte III: Highlander Simulator - Análisis de Deadlocks

VIII-A. Identificación del Invariante

El sistema define que la salud total S_{total} debe ser constante:

$$S_{total} = N \times H_{inicial} \tag{1}$$

Donde N es el número de inmortales y H la salud. En la versión *naive*, al presionar **Pause & Check**, la suma variaba debido a que los hilos no se detenían instantáneamente, permitiendo transferencias de salud mientras se realizaba el cálculo.

VIII-B. Estrategia contra Deadlocks

El deadlock ocurría cuando dos inmortales intentaban atacarse mutuamente al mismo tiempo:

- *Inmortal A bloquea a B, mientras B intenta bloquear a A.*

Se analizaron dos soluciones implementadas en el proyecto:

Estrategia	Funcionamiento
Orden Total	Los inmortales siempre adquieren los locks en orden de su ID único. Si A ataca a B, siempre se hace `synchronized(menorID) { synchronized(mayorID) }`.
TryLock	Intenta adquirir el segundo lock con un *timeout*. Si falla, libera el primer lock y reintenta tras un *backoff* aleatorio.

VIII-C. Remoción de Inmortales Muertos

Para evitar `ConcurrentModificationException` al eliminar inmortales con salud 0, se optó por una `CopyOnWriteArrayList` o una estrategia de filtrado concurrente que no requiere bloquear toda la simulación, permitiendo que el *loop* principal de pelea siga fluyendo.

IX. Validación y Resultados

- **Prueba de N alto:** Se validó la simulación con $N = 1000$ inmortales. Con la estrategia `ordered`, el sistema no presentó bloqueos.
- **Consistencia:** El uso de `PauseController` (basado en `Lock` y `Condition`) garantizó que todos los hilos estuvieran en estado `waiting` antes de imprimir el reporte de salud, logrando que el invariante se cumpla exactamente.

X. Parte III: Sincronización y Deadlocks con Highlander Simulator

X-A. Lógica de la simulación y regiones críticas

Cada inmortal es modelado como un hilo independiente que selecciona otro inmortal para atacarlo. En la versión original del enunciado, el atacante restaba M puntos de salud al oponente y ganaba $M/2$, lo cual rompe el invariante global de conservación de salud.

Decisión adoptada: Se ajustó la regla para que el atacante gane el daño completo (M).

Justificación: Para que la suma total de salud del sistema se conserve, toda resta debe compensarse exactamente con una suma equivalente. De lo contrario, la suma total decrece progresivamente, violando el invariante del sistema.

Región crítica: La operación de pelea (lectura y escritura del atributo `health`) se protege mediante bloques:

```
synchronized(first) {
    synchronized(second) {
        // transferencia de salud
    }
}
```

No existen *data races* sobre el atributo `health`, ya que toda lectura y modificación ocurre dentro de regiones críticas protegidas.

Estrategias implementadas:

- **Modo naive:** Los locks se adquieren sin orden definido, lo que puede provocar *deadlock*.
- **Modo ordered:** Los locks se adquieren en orden lexicográfico por nombre (o ID), eliminando la espera circular.

X-B. Invariante de la suma total de salud

El invariante definido para el sistema es:

$$S_{total} = N \times H \tag{2}$$

donde:

- N es el número de inmortales.
- H es la salud inicial de cada inmortal.

Con la regla ajustada (transferencia completa de M), cada pelea conserva la suma total del sistema.

X-C. Validación del invariante con Pause & Check

El botón **Pause & Check** implementa una pausa cooperativa:

1. Se solicita a todos los hilos entrar en estado de pausa.
2. Se confirma que todos estén detenidos antes de continuar.
3. Se imprime la salud individual y la suma total.

Se realizaron pruebas con $N = 8$, 100, 1000 y 10000, manteniéndose constante el invariante.

El único momento en que puede observarse una variación es durante una pelea en curso antes de completar la región crítica, pero nunca durante la verificación pausada.

X-D. Sincronización de pausa y reanudación

Se implementó un mecanismo cooperativo usando `PauseController` basado en `Lock` y `Condition`:

- Cada hilo verifica periódicamente si debe pausarse.
- La pausa garantiza que todos los hilos estén bloqueados antes de continuar.
- La operación **Resume** libera la condición de manera segura.

No se utilizaron métodos obsoletos como `Thread.suspend()` o `Thread.stop()`, evitando inconsistencias del estado interno de los hilos.

X-E. Consistencia bajo múltiples pausas

Se realizaron pruebas presionando repetidamente y de forma rápida el botón **Pause & Check**.

En todas las verificaciones:

- El invariante se mantuvo constante.
- No se observaron condiciones de carrera.
- La pausa fue efectivamente global.

Esto confirma una sincronización correcta entre los hilos y el controlador de pausa.

X-F. Estrategia para evitar deadlocks

Diagnóstico: En modo `naive`, con N grande, es posible reproducir un *deadlock*, donde hilos esperan indefinidamente por locks adquiridos en orden inverso.

El análisis mediante herramientas como `jps` y `jstack` permite observar ciclos de espera entre hilos.

Corrección aplicada: Se implementó un orden total en la adquisición de locks basado en el identificador único del inmortal:

```
if (id1 < id2) {
    synchronized(i1) {
        synchronized(i2) {
            // pelea
        }
    }
}
```

Al eliminar la espera circular, se elimina una de las cuatro condiciones necesarias para que ocurra un *deadlock* (condición de espera circular).

X-G. Remoción de inmortales muertos

La colección de inmortales fue migrada a `CopyOnWriteArrayList`.

Justificación técnica:

- Permite iteración segura mientras otros hilos modifican la lista.
- Evita `ConcurrentModificationException`.
- No requiere sincronización global.
- Es adecuada porque las lecturas son mucho más frecuentes que las eliminaciones.

Esto permite remover inmortales muertos sin bloquear completamente la simulación.

X-H. Implementación de STOP (Apagado ordenado)

El botón **Stop** implementa terminación cooperativa:

- Se solicita a cada hilo finalizar su ejecución.
- Se cierra el `ExecutorService`.
- Se utiliza `awaitTermination()` para esperar la finalización completa.

Se garantiza que no queden hilos activos ni recursos pendientes tras el cierre de la simulación.

X-I. Robustez y escalabilidad

La aplicación fue validada con valores altos de N (hasta $10\,000$ inmortales), observándose que:

- No se presentan *deadlocks* en modo `ordered`.
- No ocurren `ConcurrentModificationException`.
- No existen *data races*.
- La interfaz gráfica mantiene comportamiento estable.

X-J. Documentación y justificación técnica

Se agregó documentación *JavaDoc* detallada en todas las clases y métodos modificados, especificando:

- Región crítica protegida.
- Estrategia de sincronización aplicada.
- Punto del enunciado abordado.
- Justificación técnica de cada decisión.

Conclusión de la Parte III

La solución implementada cumple los criterios de concurrencia correcta, preservación estricta del invariante global, eliminación de *deadlocks* en modo seguro, robustez ante alta concurrencia y documentación técnica adecuada. El sistema demuestra consistencia, escalabilidad y diseño concurrente sólido.

XI. Conclusiones Generales

1. La **espera activa** es una práctica ineficiente que debe reemplazarse por mecanismos de suspensión y notificación. 2. Los **deadlocks** en transferencias de recursos se resuelven estableciendo un orden jerárquico global para la adquisición de monitores. 3. El uso de **invariantes** es la herramienta más efectiva para verificar la correctitud de un sistema concurrente complejo.

Referencias

- Documentación Java 21 (Virtual Threads & Locks).
- Goetz, B. (2006). *Java Concurrency in Practice*.