Laboratorio Sincronización de hilos y DeadLocks 03

Sebastián Cardona, Laura Gil, Zayra Gutiérrez

Ingeniero de Sistemas

Javier Toquica Barrera

Universidad Escuela Colombiana de ingeniería Julio Garavito

2025-1

Contenido

Introducción

En el desarrollo de aplicaciones concurrentes, la correcta sincronización de hilos es fundamental para garantizar la eficiencia y evitar problemas como condiciones de carrera y bloqueos mutuos (deadlocks). Este laboratorio tiene como objetivo explorar y solucionar estos desafíos a través de ejercicios prácticos en Java.

En la primera parte del laboratorio, se trabajará con el patrón Productor-Consumidor, implementando mecanismos de sincronización con wait() y notifyAll() para optimizar el uso de la CPU y evitar sobrecargas de memoria. A través del análisis con herramientas como JVisualVM, se evaluará el impacto del uso ineficiente de los recursos y se realizarán los ajustes necesarios para mejorar el rendimiento.

La segunda parte del laboratorio se centrará en la simulación del juego "Highlander Simulator", donde múltiples hilos representan inmortales en batalla. Se analizarán problemas comunes en la concurrencia, como la inconsistencia en el invariante de salud, y se implementarán estrategias para su corrección. Además, se explorarán técnicas para evitar deadlocks mediante un ordenamiento seguro de adquisición de bloqueos y se trabajará en la eliminación controlada de hilos inactivos sin afectar el rendimiento del sistema.

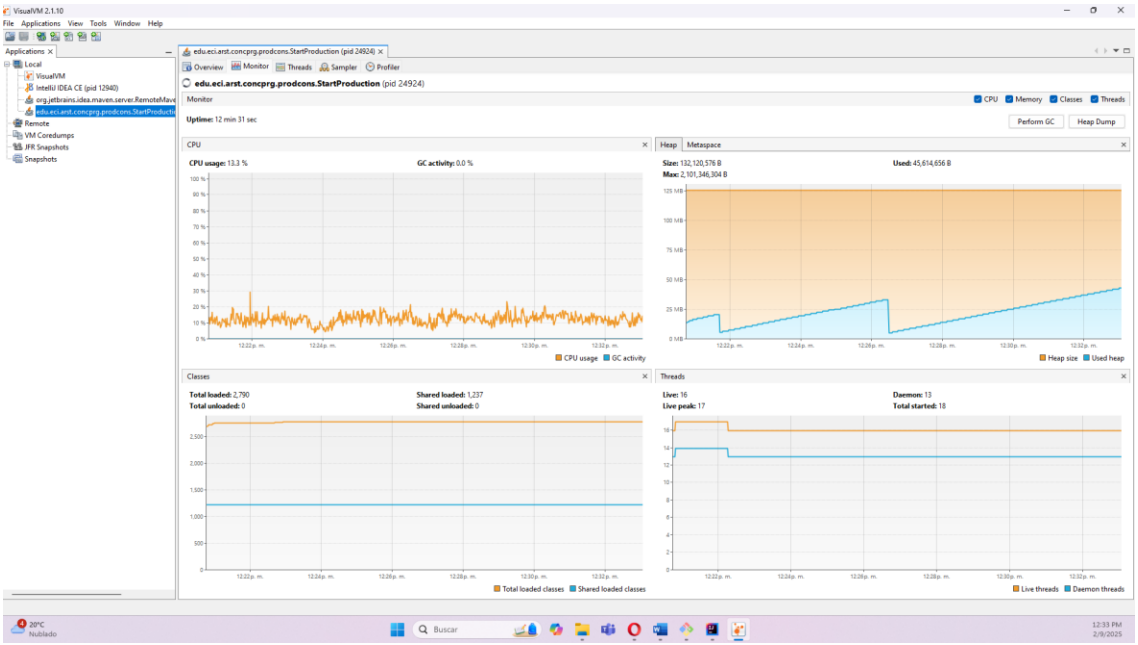
Toda la parte práctica de este laboratorio, incluyendo el código y las implementaciones realizadas, puede encontrarse en el siguiente repositorio de GitHub:
Repositorio del laboratorio: https://github.com/ZayraGS1403/ARSW-Lab03

Este laboratorio permitirá afianzar conocimientos clave en concurrencia, mejorar la capacidad de diagnóstico y solución de problemas en aplicaciones multihilo y comprender la importancia de una correcta gestión de los recursos compartidos.

Desarrollo del Laboratorio

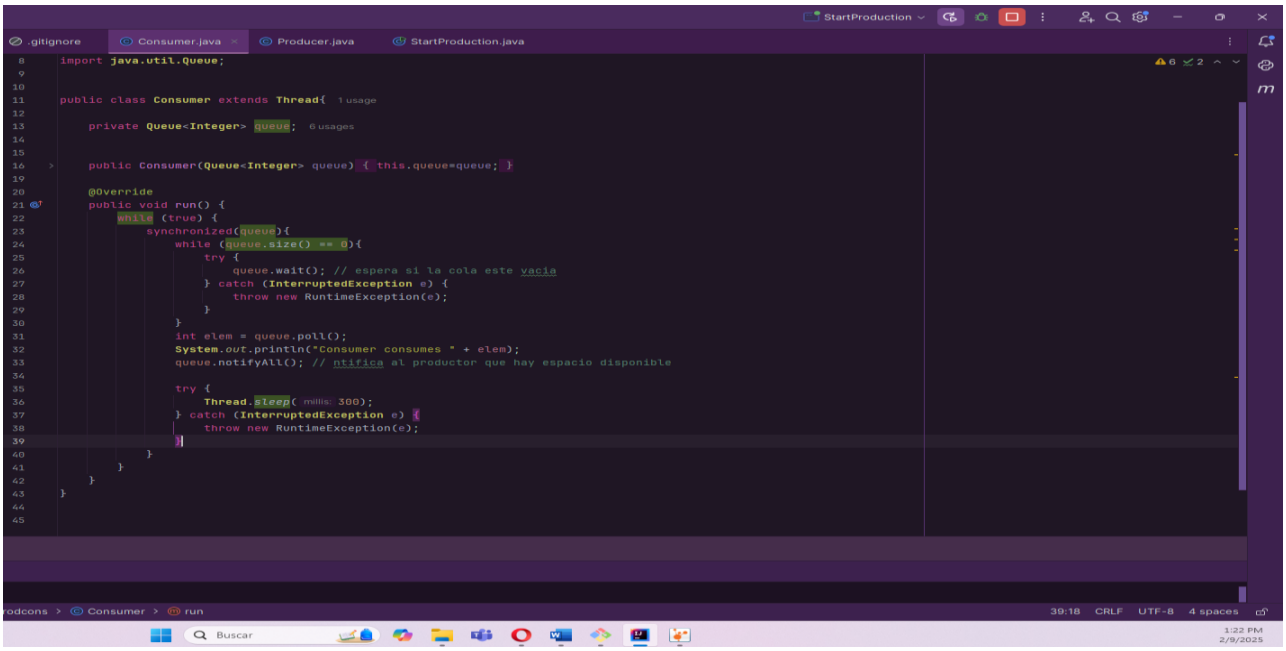
Parte I – *Control de hilos con wait/notify. Productor/consumidor.*

1. *Revise el funcionamiento del programa y ejecútelo. Mientras esto ocurren, ejecute jVisualVM y revise el consumo de CPU del proceso correspondiente. ¿A qué se debe este consumo?, cual es la clase responsable?*

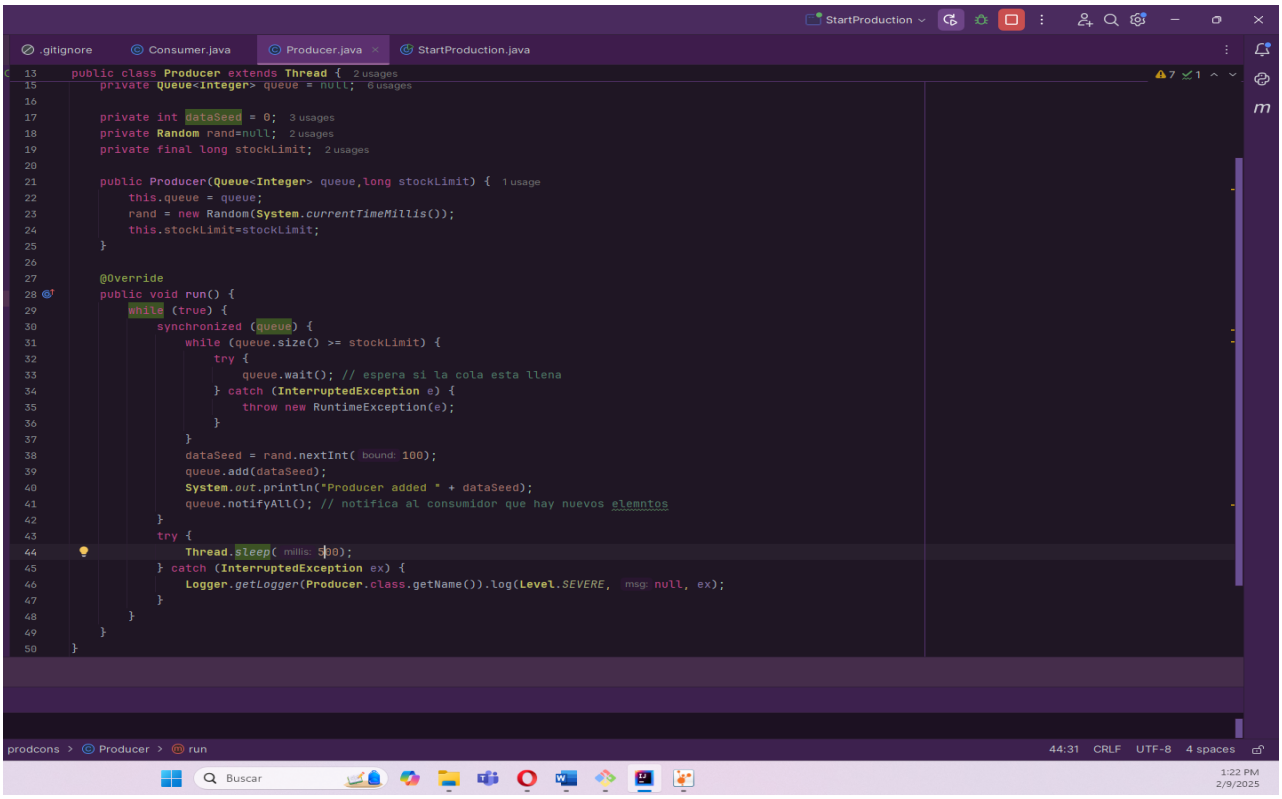


Aunque no hay un alto consumo de la CPU (10-30%), igualmente su incremental uso, se debe a que el Consumer entra en un bucle infinito sin pausas cuando la cola está vacía, ya que no utiliza wait() y notify(), lo que provoca iteraciones innecesarias sin hacer trabajo útil. Por otro lado, el aumento constante de la memoria heap ocurre porque el Producer sigue añadiendo elementos sin respetar un límite, lo que hace que la cola crezca indefinidamente y genere un uso excesivo de memoria.

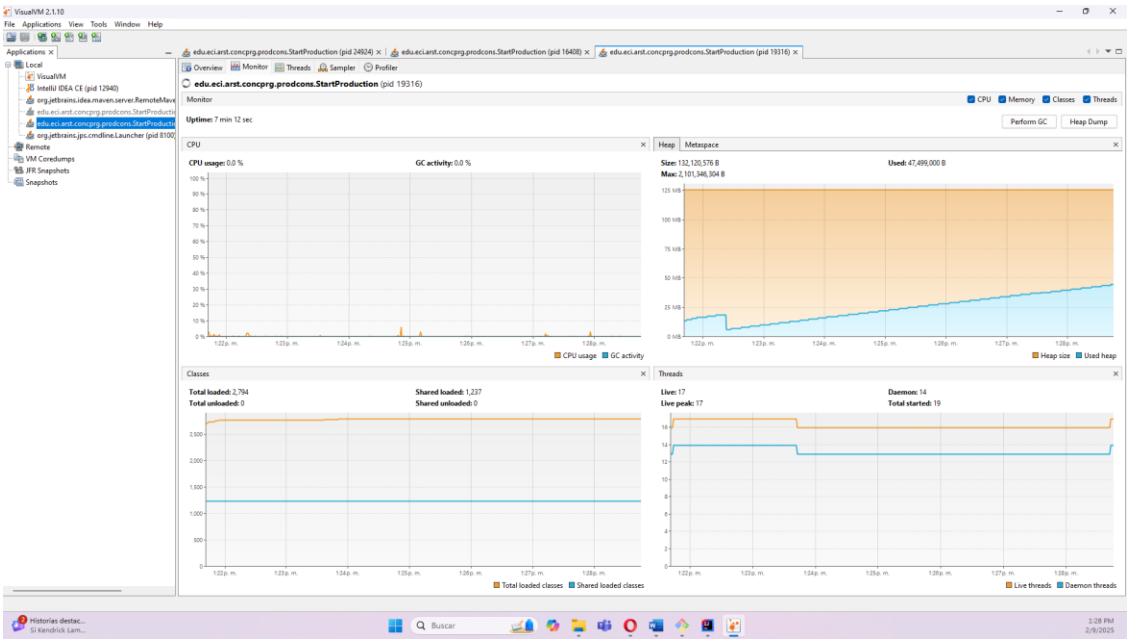
2. *Haga los ajustes necesarios para que la solución use más eficientemente la CPU, teniendo en cuenta que -por ahora- la producción es lenta y el consumo es rápido. Verifique con JVisualVM que el consumo de CPU se reduzca.*



```java
import java.util.Queue;


public class Consumer extends Thread{  1 usage

    private Queue<Integer> queue;  6 usages


    public Consumer(Queue<Integer> queue) { this.queue=queue; }

    @Override
    public void run() {
        while (true) {
            synchronized(queue){
                while (queue.size() == 0){
                    try {
                        queue.wait(); // espera si la cola este vacia
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
                int elem = queue.poll();
                System.out.println("Consumer consumes " + elem);
                queue.notifyAll(); // ntifica al productor que hay espacio disponible

                try {
                    Thread.sleep( millis: 300);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
        }
    }
}
```

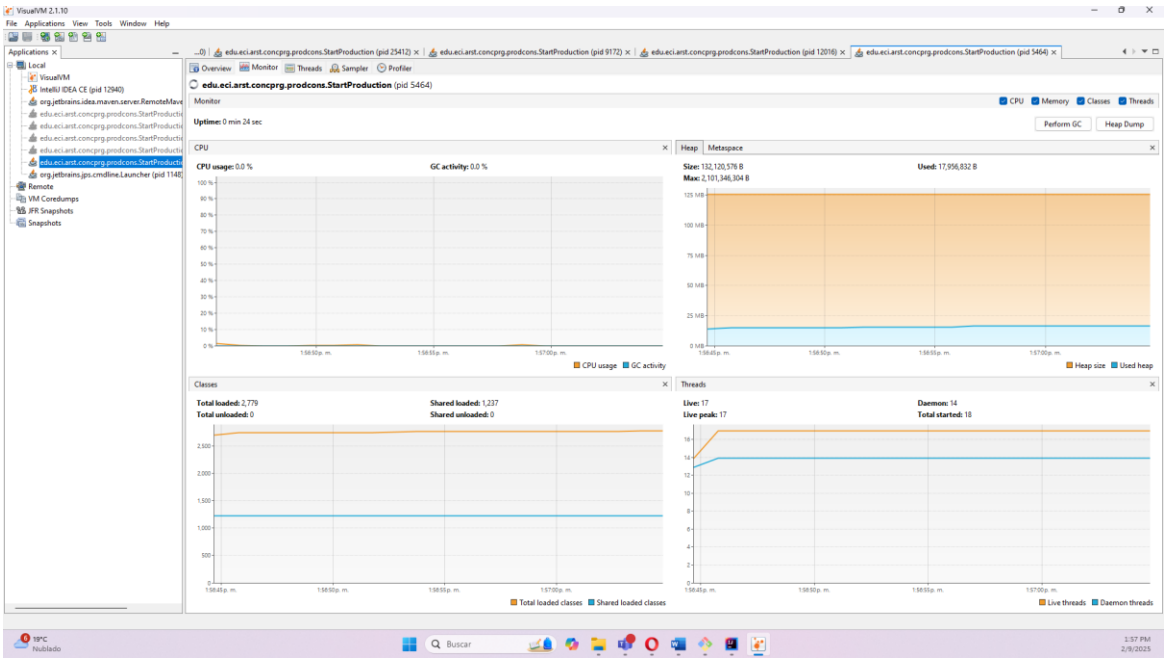



```java
public class Producer extends Thread {  2 usages
    private Queue<Integer> queue = null;  6 usages

    private int dataSeed = 0;  3 usages
    private Random rand=null;  2 usages
    private final long stockLimit;  2 usages

    public Producer(Queue<Integer> queue,long stockLimit) {  1 usage
        this.queue = queue;
        rand = new Random(System.currentTimeMillis());
        this.stockLimit=stockLimit;
    }

    @Override
    public void run() {
        while (true) {
            synchronized (queue) {
                while (queue.size() >= stockLimit) {
                    try {
                        queue.wait(); // espera si la cola esta llena
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
                dataSeed = rand.nextInt( bound: 100);
                queue.add(dataSeed);
                System.out.println("Producer added " + dataSeed);
                queue.notifyAll(); // notifica al consumidor que hay nuevos elemntos
            }
            try {
                Thread.sleep( millis: 500);
            } catch (InterruptedException ex) {
                Logger.getLogger(Producer.class.getName()).log(Level.SEVERE,  msg: null, ex);
            }
        }
    }
}
```



Después de implementar wait() y notifyAll() en la comunicación entre el Productor y el Consumidor, el consumo de CPU disminuyó significativamente a un rango mayor a 1%, ya que el Consumer ahora espera cuando la cola está vacía en lugar de ejecutar un bucle infinito sin pausas. Además, al establecer un límite en la cantidad de elementos en la cola, el Producer evita agregar datos de manera incontrolada, reduciendo el uso innecesario de memoria. Con estos cambios, la solución es más eficiente, manteniendo un equilibrio entre producción y consumo sin generar una carga excesiva en la CPU.

3. *Haga que ahora el productor produzca muy rápido, y el consumidor consuma lento. Teniendo en cuenta que el productor conoce un límite de Stock (cuantos elementos debería tener, a lo sumo en la cola), haga que dicho límite se respete. Revise el API de la colección usada como cola para ver cómo garantizar que dicho límite no se supere. Verifique que, al poner un límite pequeño para el 'stock', no haya consumo alto de CPU ni errores.*



El programa ahora es muy eficiente, con un uso de CPU cercano al 0% cuando no hay trabajo. El productor espera si la cola está llena y el consumidor si está vacía, usando wait() para evitar gastar CPU. Se comunican con notifyAll(): el productor avisa al consumidor al agregar un elemento, y el consumidor avisa al productor al retirarlo. La cola tiene un límite (stockLimit), evitando que crezca demasiado. Aunque el productor es rápido y el consumidor lento, el sistema se mantiene estable y eficiente.

Parte II. *Sincronización y Dead-Locks.*

1. *Revise el programa "highlander-simulator", dispuesto en el paquete edu.eci.arsw.highlandersim. Este es un juego en el que:*
   - *Se tienen N jugadores inmortales.*
   - *Cada jugador conoce a los N-1 jugador restantes.*
   - *Cada jugador, permanentemente, ataca a algún otro inmortal. El que primero ataca le resta M puntos de vida a su contrincante, y aumenta en esta misma cantidad sus propios puntos de vida.*

- *El juego podría nunca tener un único ganador. Lo más probable es que al final* sólo queden dos, peleando indefinidamente quitando y sumando puntos de vida.

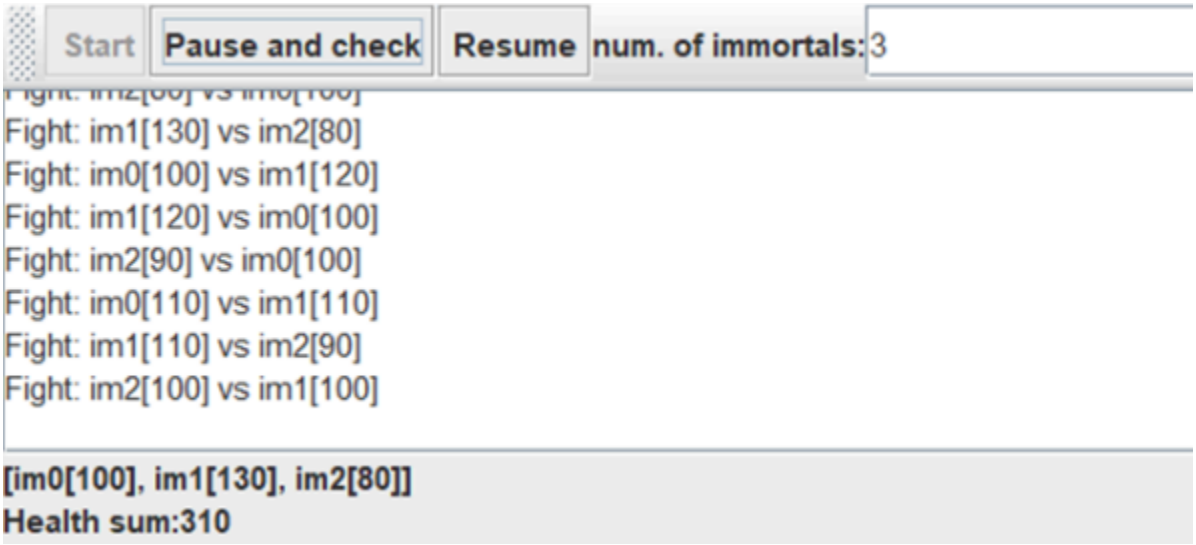
2. *Revise el código e identifique cómo se implementó la funcionalidad antes indicada. Dada la intención del juego, un invariante debería ser que la sumatoria de los puntos de vida de todos los jugadores siempre sea el mismo(claro está, en un instante de tiempo en el que no esté en proceso una operación de incremento/reducción de tiempo). Para este caso, para N jugadores, ¿cuál debería ser este valor?*

Si cada jugador tiene 100 puntos de vidas entonces el valor que no debe cambiar es N*100

3. *Ejecute la aplicación y verifique cómo funcionan las opciones 'pause and check'. ¿Se cumple el invariante?*

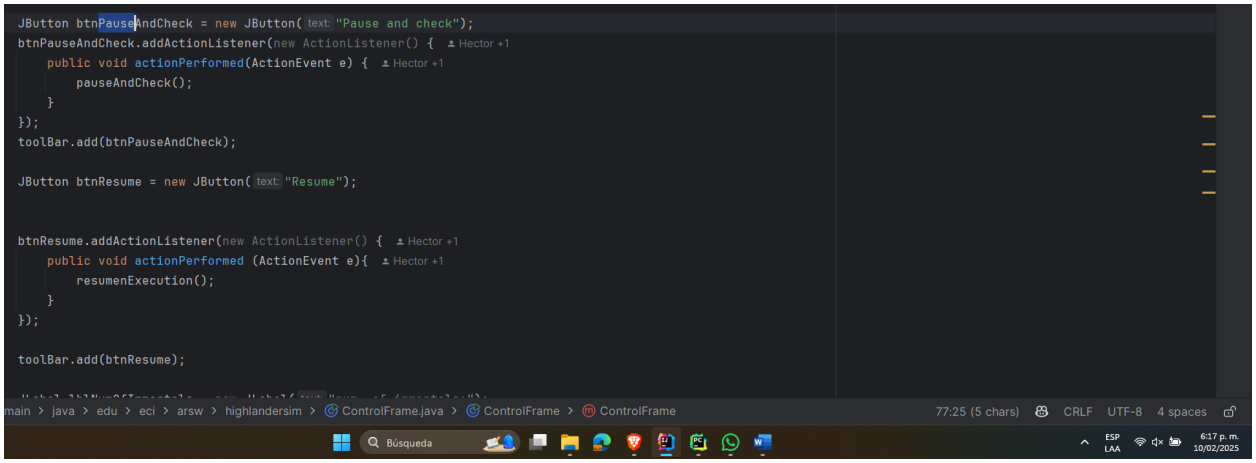
No, no se cumple la invariante, en varios casos se ve que la invariante es menor o es mayor a la que debería ser.

Como en este caso que para 3 jugadores debería ser 310, pero no es así.

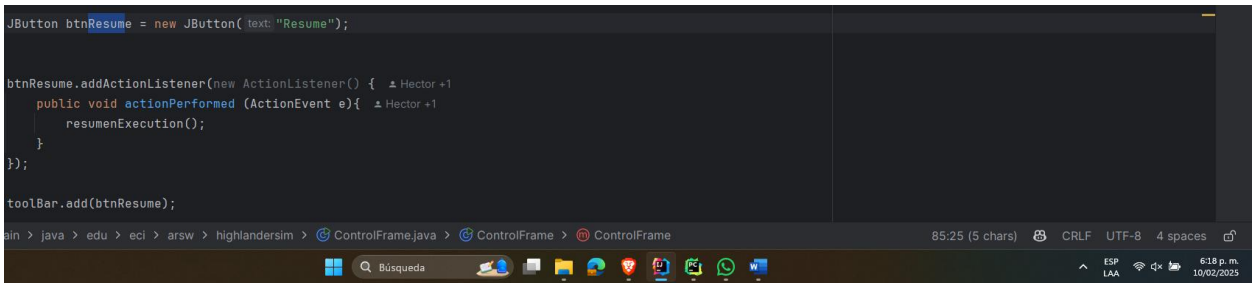


4. *Una primera hipótesis para que se presente la condición de carrera para dicha función (pause and check), es que el programa consulta la lista cuyos valores va a imprimir, a la vez que otros hilos modifican sus valores. Para corregir esto, haga lo que sea necesario para que efectivamente, antes de imprimir los resultados actuales, se pausen todos los demás hilos. Adicionalmente, implemente la opción 'resume'.*

Implementación para el botón pause and check.

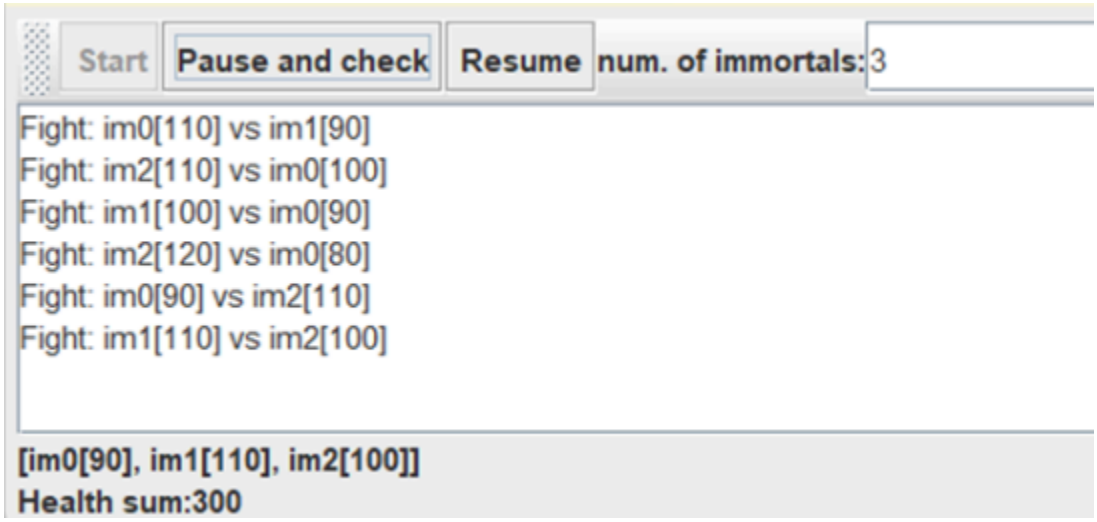


Implementación para el botón resume.



5. *Verifique nuevamente el funcionamiento (haga clic muchas veces en el botón). ¿Se cumple o no el invariante?*

La invariante se cumple



6. *Identifique posibles regiones críticas en lo que respecta a la pelea de los inmortales. Implemente una estrategia de bloqueo que evite las condiciones de carrera. Recuerde que si usted requiere usar dos o más 'locks' simultáneamente, puede usar bloques sincronizados anidados:*

synchronized(locka){

synchronized(lockb){

```
            ...
            }
    }
```

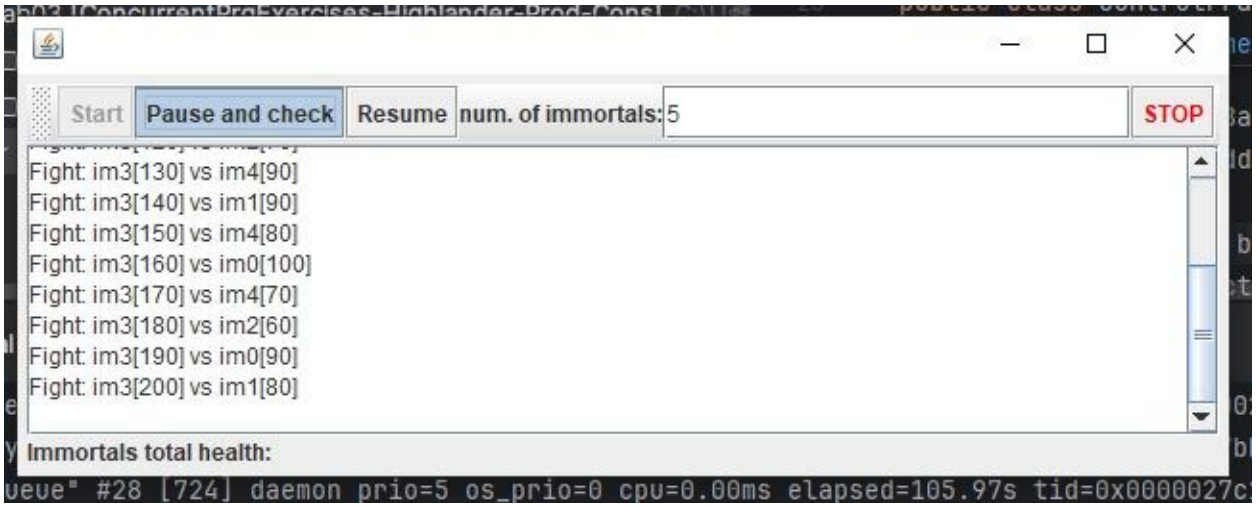
```
public void fight(Immortal i2) {  1 usage   SebastianCardona-P +2
    Immortal first = this;
    Immortal second = i2;
    // Asegurar orden consistente de bloqueo
    if (first.getName().compareTo(second.getName()) > 0) {
        first = i2;
        second = this;
    }
    synchronized(first) {
        synchronized (second) {
            if (i2.getHealth() > 0) {
                i2.changeHealth( v: i2.getHealth() - defaultDamageValue);
                this.changeHealth( v: getHealth() + defaultDamageValue);
                updateCallback.processReport("Fight: " + this + " vs " + i2 + "\n");
            } else {
                updateCallback.processReport(this + " says:" + i2 + " is already dead!\n");
            }
        }
```


En el método fight(), Contiene la región crítica, específicamente en la actualización de su.
Si dos hilos intentan modificar la salud de los mismos dos inmortales al mismo tiempo, pueden ocurrir condiciones de carrera, resultando en inconsistencias en los valores de salud.

Para evitar este problema, se ha implementado una estrategia de bloqueo utilizando synchronized anidado.
La sincronización doble asegura que solo un hilo a la vez pueda modificar los atributos de los dos inmortales involucrados en la pelea.
Este mecanismo impide que un hilo adquiera el bloqueo de Inmortal A mientras otro hilo intenta bloquear Inmortal B en orden inverso, eliminando la posibilidad de interbloqueo.

7. *Tras implementar su estrategia, ponga a correr su programa, y ponga atención a si éste se llega a detener. Si es así, use los programas jps y jstack para identificar por qué el programa se detuvo.*

Efectivamente el programa se detuvo, para e análisis se usó:
jps: Se utilizó para listar los procesos de Java en ejecución, identificando el PID del proceso del programa.
jstack <PID>: Se empleó para obtener el estado de los hilos del programa.




```
TimerQueue" #28 [724] daemon prio=5 os_prio=0 cpu=0.00ms elapsed=40.58s tid=0x0000027c1d2597a0 nid=724 waiting on condition  [0x000000a278efe000]
im0" #30 [14968] prio=6 os_prio=0 cpu=0.00ms elapsed=2.74s tid=0x0000027c1d356130 nid=14968 waiting on condition  [0x000000a2792ff000]
im1" #31 [16344] prio=6 os_prio=0 cpu=0.00ms elapsed=2.74s tid=0x0000027c1c0697b0 nid=16344 waiting for monitor entry  [0x000000a2793ff000]
im2" #32 [14904] prio=6 os_prio=0 cpu=0.00ms elapsed=2.74s tid=0x0000027c1c069e40 nid=14904 waiting for monitor entry  [0x000000a2794ff000]
im3" #33 [21568] prio=6 os_prio=0 cpu=0.00ms elapsed=2.74s tid=0x0000027c1c069120 nid=21568 waiting for monitor entry  [0x000000a2795ff000]
im4" #34 [12916] prio=6 os_prio=0 cpu=0.00ms elapsed=2.74s tid=0x0000027c1c067d70 nid=12916 waiting for monitor entry  [0x000000a2796ff000]
VM Thread" os_prio=2 cpu=0.00ms elapsed=41.10s tid=0x0000027bd6bb0170 nid=16784 runnable
VM Periodic Task Thread" os_prio=2 cpu=0.00ms elapsed=41.11s tid=0x0000027bd6b99370 nid=11448 waiting on condition
```

Al analizar la salida de jstack, se observó que varios hilos estaban en "waiting for monitor entry", lo que indica que estaban bloqueados esperando acceder a un recurso que otro hilo ya tenía bloqueado. Esto sugiere un posible interbloqueo (deadlock).

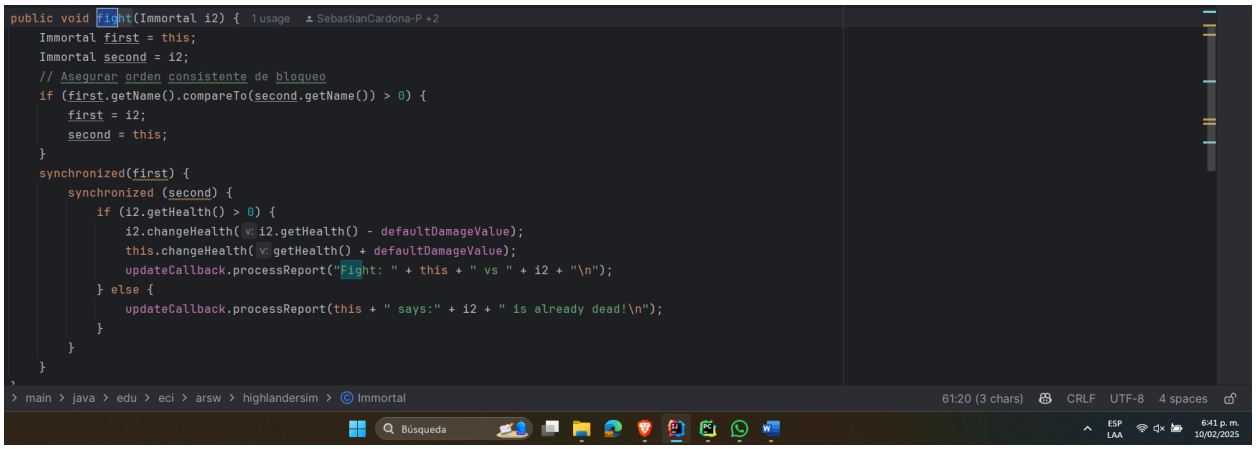
Este problema ocurre porque, aunque se implementó un orden de bloqueo en fight(), es posible que otro hilo haya adquirido primero el segundo objeto (second) antes del primero (first), lo que genera un ciclo de espera.

```
"TimerQueue" #28 [724] daemon prio=5 os_prio=0 cpu=0.00ms elapsed=51.84s tid=0x0000027c1d2597a0 nid=724 waiting on condition  [0x000000a278efe000]
"im0" #30 [14968] prio=6 os_prio=0 cpu=0.00ms elapsed=14.01s tid=0x0000027c1d356130 nid=14968 waiting for monitor entry  [0x000000a2792ff000]
"im1" #31 [16344] prio=6 os_prio=0 cpu=0.00ms elapsed=14.01s tid=0x0000027c1c0697b0 nid=16344 waiting for monitor entry  [0x000000a2793ff000]
"im2" #32 [14904] prio=6 os_prio=0 cpu=0.00ms elapsed=14.01s tid=0x0000027c1c069e40 nid=14904 waiting for monitor entry  [0x000000a2794ff000]
"im3" #33 [21568] prio=6 os_prio=0 cpu=0.00ms elapsed=14.01s tid=0x0000027c1c069120 nid=21568 waiting on condition  [0x000000a2795ff000]
"im4" #34 [12916] prio=6 os_prio=0 cpu=0.00ms elapsed=14.01s tid=0x0000027c1c067d70 nid=12916 waiting for monitor entry  [0x000000a2796ff000]
"VM Thread" os_prio=2 cpu=0.00ms elapsed=52.37s tid=0x0000027bd6bb0170 nid=16784 runnable
```

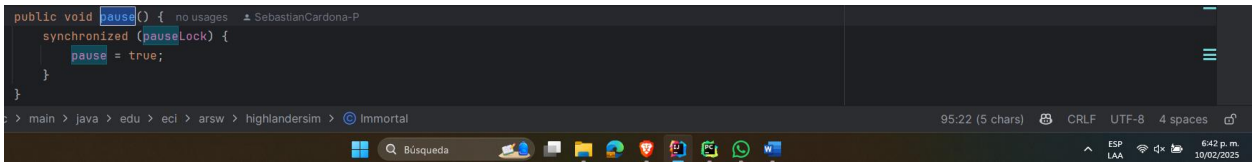
```
TimerQueue" #28 [724] daemon prio=5 os_prio=0 cpu=0.00ms elapsed=68.69s tid=0x0000027c1d2597a0 nid=724 waiting on condition  [0x000000a278efe000]
im0" #30 [14968] prio=6 os_prio=0 cpu=0.00ms elapsed=30.86s tid=0x0000027c1d356130 nid=14968 waiting for monitor entry  [0x000000a2792ff000]
im1" #31 [16344] prio=6 os_prio=0 cpu=0.00ms elapsed=30.86s tid=0x0000027c1c0697b0 nid=16344 waiting for monitor entry  [0x000000a2793ff000]
im2" #32 [14904] prio=6 os_prio=0 cpu=0.00ms elapsed=30.86s tid=0x0000027c1c069e40 nid=14904 waiting for monitor entry  [0x000000a2794ff000]
im3" #33 [21568] prio=6 os_prio=0 cpu=0.00ms elapsed=30.86s tid=0x0000027c1c069120 nid=21568 waiting on condition  [0x000000a2795ff000]
im4" #34 [12916] prio=6 os_prio=0 cpu=0.00ms elapsed=30.86s tid=0x0000027c1c067d70 nid=12916 waiting for monitor entry  [0x000000a2796ff000]
VM Thread" os_prio=2 cpu=0.00ms elapsed=69.22s tid=0x0000027bd6bb0170 nid=16784 runnable
```

8. *Plantee una estrategia para corregir el problema antes identificado (puede revisar de nuevo las páginas 206 y 207 de Java Concurrency in Practice).*

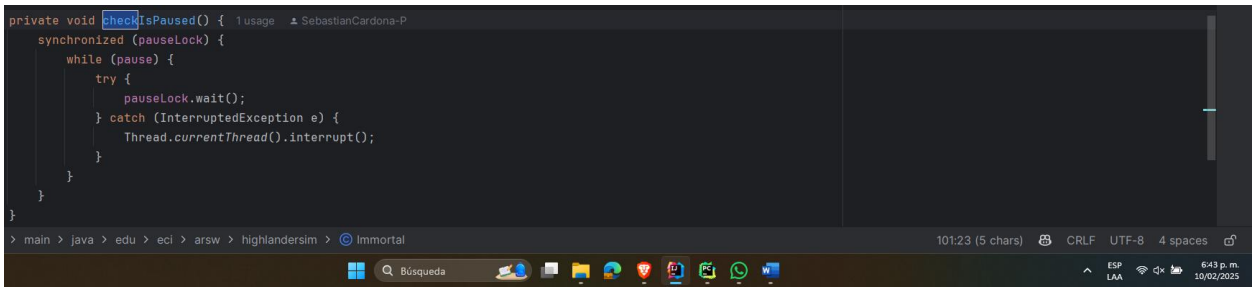
se estableció un orden consistente de adquisición de bloqueos, asegurando que siempre se adquieran en el mismo orden basado en el nombre del hilo:



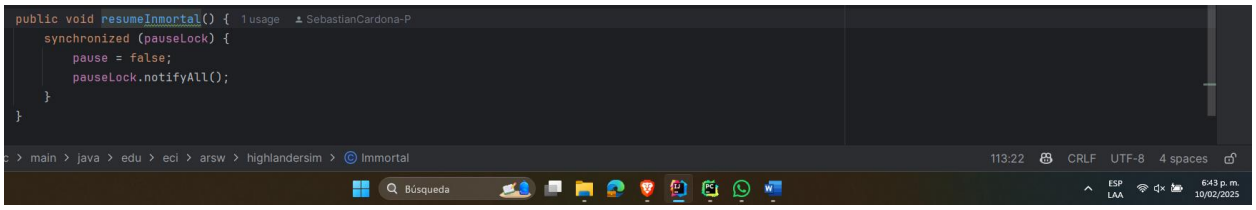
Se añadió un mecanismo para pausar y reanudar los hilos de forma segura mediante wait() y notifyAll(). Esto evita interrumpir los hilos abruptamente y permite un control más estable de la ejecución.



Método para verificar si el hilo está pausado y esperar:



Método para reanudar un hilo:

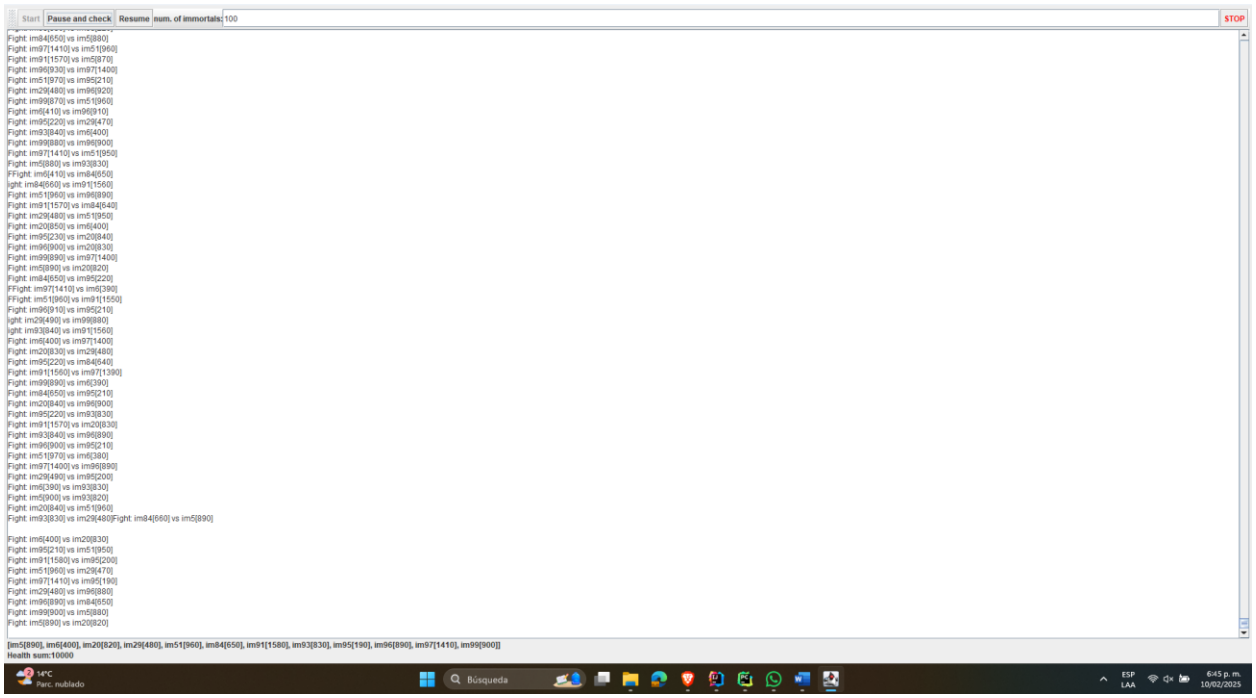


Se implementó pauseLock.wait() para suspender los hilos de forma segura y notifyAll() para reanudarlos.

Para evitar condiciones de carrera, se usó una lista sincronizada con Collections.synchronizedList(), y se protegió el acceso a la lista con synchronized.

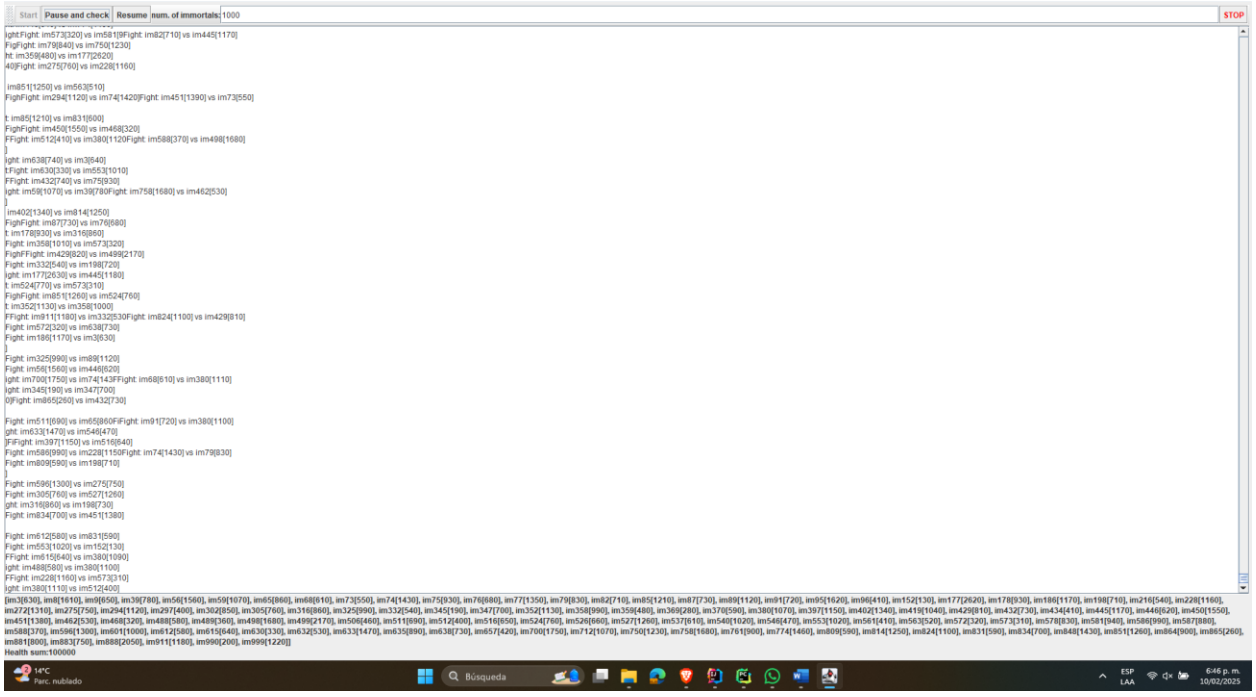
9. *Una vez corregido el problema, rectifique que el programa siga funcionando de manera consistente cuando se ejecutan 100, 1000 o 10000 inmortales. Si en estos casos grandes se empieza a incumplir de nuevo el invariante, debe analizar lo realizado en el paso 4.*

N=100



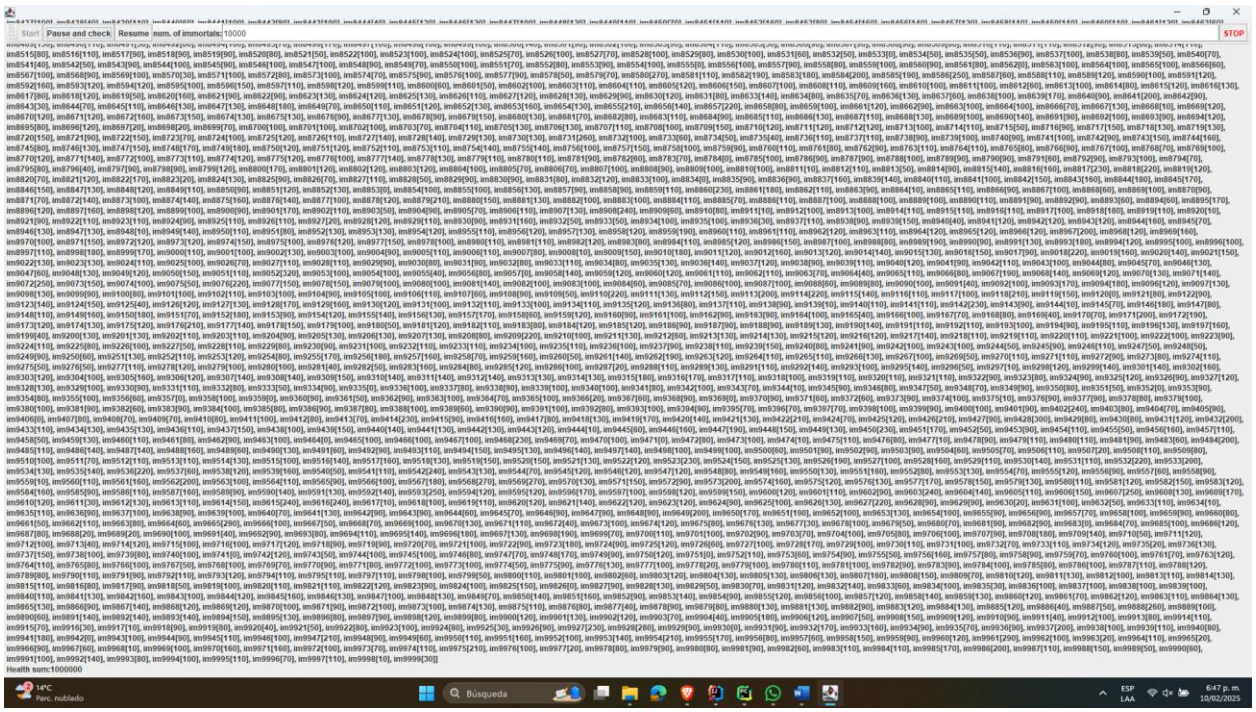
Efectivamente la invariante se cumplió.

N=1000



De igual forma la invariante se cumple.

N=*10000*



También se cumple la invariante para este caso.

10. *Un elemento molesto para la simulación es que en cierto punto de la misma hay pocos 'inmortales' vivos realizando peleas fallidas con 'inmortales' ya muertos. Es necesario ir suprimiendo los inmortales muertos de la simulación a medida que van muriendo. Para esto:*
    - *¿Analizando el esquema de funcionamiento de la simulación, esto podría crear una condición de carrera? Implemente la funcionalidad, ejecute la simulación y observe qué problema se presenta cuando hay muchos 'inmortales' en la misma. Escriba sus conclusiones al respecto en el archivo RESPUESTAS.txt.*

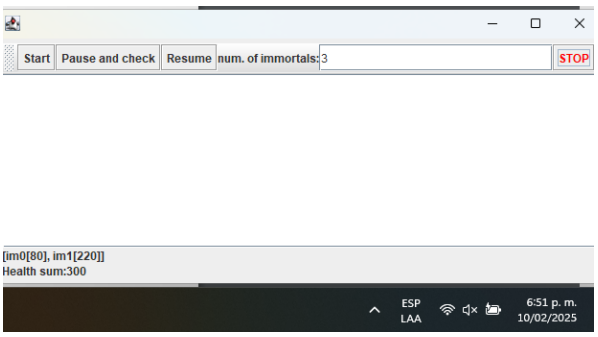
    - *Corrija el problema anterior **SIN hacer uso de sincronización**, pues volver secuencial el acceso a la lista compartida de inmortales haría extremadamente lenta la simulación.*

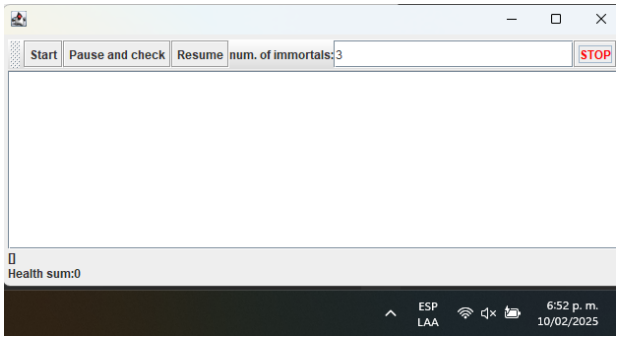
```java
public void run() {  Hector +1

    while (health > 0) {
        Immortal im;
        checkIsPaused();
        int myIndex = immortalsPopulation.indexOf(this);
        int nextFighterIndex = r.nextInt(immortalsPopulation.size());
        //avoid self-fight
        if (nextFighterIndex == myIndex) {
            nextFighterIndex = ((nextFighterIndex + 1) % immortalsPopulation.size());
        }
        if(nextFighterIndex < immortalsPopulation.size()){
            im = immortalsPopulation.get(nextFighterIndex);
            if(im.getHealth() != 0){
                this.fight(im);
            }
            try {
                Thread.sleep( millis: 1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            if(health == 0){
                immortalsPopulation.remove( o: this);
            }
        }
    }

}
```

11. Para finalizar, implemente la opción STOP.

Implementación del botón STOP.

```java
JButton btnStop = new JButton( text: "STOP");
btnStop.setForeground(Color.RED);

toolBar.add(btnStop);

btnStop.addActionListener(new ActionListener() {  ZayraGS1403
    public void actionPerformed (ActionEvent e){  ZayraGS1403
        output.setText("");
        stopExecution();
        btnStart.setEnabled(true);
    }
});
```





Quedaría listo para iniciar otra batalla.

Conclusiones

La correcta sincronización de hilos es esencial para evitar problemas como el uso excesivo de CPU y la sobrecarga de memoria. En la implementación del patrón Productor-Consumidor, el uso de wait() y notifyAll() optimizó significativamente el rendimiento del sistema al evitar bucles innecesarios y garantizar un flujo eficiente de datos.

Se evidenció que la falta de sincronización adecuada puede generar inconsistencias en los datos compartidos, como ocurrió en la simulación de los inmortales donde el invariante de salud no se mantenía estable. La solución implementada, que pausaba los hilos antes de la verificación, permitió mantener la integridad del sistema.

El uso de bloques sincronizados anidados ayudó a evitar condiciones de carrera, pero también generó interbloqueos. Para corregir este problema, se estableció un orden de adquisición de bloqueos, asegurando que los recursos fueran tomados de manera consistente y evitando ciclos de espera infinita entre hilos.