**Escuela Colombiana de Ingeniería Julio Garavito**

Imagen que contiene Texto

Descripción generada automáticamente

**LABORATORIO No.3**

**Presentado por:**

Esteban Aguilera Contreras

Carlos David Barrero Velásquez

Juan Sebastián Beltrán Rodríguez

Iván Santiago Forero Torres

**Asignatura:**

ISIS – ARSW (Arquitectura de Software)

**Grupo:**

Grupo 5

**Docente:**

Javier Ivan Toquica Barrera

**29/08/2025**

TABLA DE CONTENIDO

[OBJETIVOS 3](#_Toc207441975)

[Objetivo General: 3](#_Toc207441976)

[Objetivos Específicos: 3](#_Toc207441977)

[Herramientas Utilizadas: 3](#_Toc207441978)

[Requisitos: 3](#_Toc207441979)

[MARCO TEORICO 4](#_Toc207441980)

[EJECUCIÓN DEL LABORATORIO 5](#_Toc207441981)

[CONCLUSIONES 6](#_Toc207441982)

[WEBGRAFIA 7](#_Toc207441983)

# OBJETIVOS

## Objetivo General:

## Objetivos Específicos:

## Herramientas Utilizadas:

## Requisitos:

# MARCO TEORICO

# EJECUCIÓN DEL LABORATORIO

**Productor/Consumidor con wait/notify (y contraste con busy-wait) (Parte I)**

1. Ejecuta el programa de productor/consumidor y monitorea CPU con jVisualVM. ¿Por qué el consumo alto? ¿Qué clase lo causa?

* Para poder hacer la visualización hacemos los siguientes pasos:

mvn -q -DskipTests package

**(Productor lento / consumidor rápido en modo spin)**

java -cp target/classes -Dmode=spin -Dproducers=1 -Dconsumers=1 -Dcapacity=8 -DprodDelayMs=50 -DconsDelayMs=1 -DdurationSec=30 edu.eci.arsw.pc.PCApp

* -Dproducers=1 🡪 número de hilos productores.
* -Dconsumers=1 🡪 número de hilos consumidores.
* -Dcapacity=8 🡪 capacidad máxima de la cola.
* -DprodDelayMs=50 🡪 demora en milisegundos para que el productor genere un nuevo elemento.
* -DconsDelayMs=1 🡪 demora del consumidor en milisegundos en procesar el elemento.
* -DdurationSec=30 🡪Tiempo total de la ejecución en segundos.

Captura de pantalla de computadora

El contenido generado por IA puede ser incorrecto.

Interfaz de usuario gráfica, Aplicación, Tabla, Excel

El contenido generado por IA puede ser incorrecto.

Gráfico, Gráfico de barras

El contenido generado por IA puede ser incorrecto.

* El hilo main duerme hasta que se cumple el durationSec:

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

* Al verificar los tiempos de hilos de la CPU Podemos ver que hay dos hilos denominados “ForkJoinPool-1-worker-1” y “ForkJoinPool-1-worker-2” los cuales son los que abarcan los recursos de nuestra CPU.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

* En la clase BusyspinQueue los métodos put() y take() porque utiliza bucles infinitos mediate Thread.onSpinWait(), haciendo que un bucle se mantenga en CPU haciendo una pequeña optimización para evitar penalizaciones por rendimiento.

Texto

El contenido generado por IA puede ser incorrecto.

Texto

El contenido generado por IA puede ser incorrecto.

1. Ajusta la implementación para usar CPU eficientemente cuando el productor es lento y el consumidor es rápido. Valida de nuevo con VisualVM.

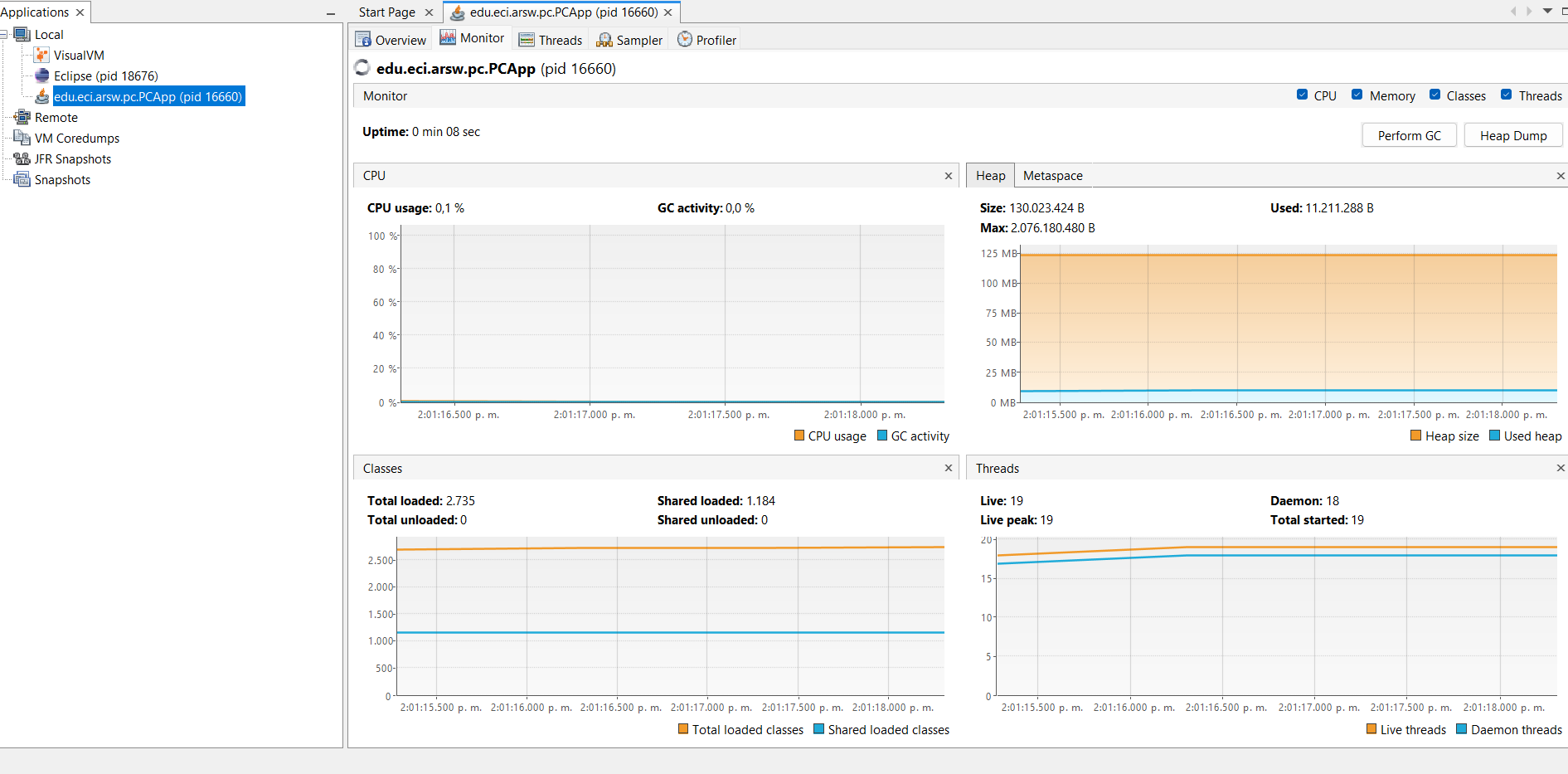
* Para poder hacer la visualización hacemos los siguientes pasos:

mvn -q -DskipTests package

**(Productor lento / consumidor rápido en modo monitor)**

java -cp target/classes -Dmode=monitor -Dproducers=1 -Dconsumers=1 -Dcapacity=8 -DprodDelayMs=50 -DconsDelayMs=1 -DdurationSec=30 edu.eci.arsw.pc.PCApp

* Podemos ver que existe un uso de CPU de 0,1% prácticamente nulo, por lo tanto podemos concluir que la implementación con wait() libero CPU correctamente.



Una captura de pantalla de una red social

El contenido generado por IA puede ser incorrecto.

* El consumidor cuando no hay elementos se encuentra en estado de WAIT color amarillo

Interfaz de usuario gráfica

El contenido generado por IA puede ser incorrecto.

* Indica que la CPU no esta siendo gastada de manera intensiva.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

* La distribución de carga se puede ver que se hace de manera mas equitativa, el consumidor y el productor casi no están ocupando CPU, se puede evidenciar que se bloquean correctamente.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

* Con la ejecución de los monitores (Wait, notifyAll) podemos apreciar que el sistema se comporta de manera eficiente, el consumidor no queda ocioso por información y el productor se “duerme” mientras produce algún tipo de dato, evitando así consumo elevado de CPU.

/////////////////////////////////////////////////////////////////

* Ejemplo de uso bajo ejecución con los siguientes parámetros: ***mvn -q -DskipTests exec:java -Dexec.mainClass=edu.eci.arsw.pc.PCApp \ -Dmode=spin -Dproducers=1 -Dconsumers=1 -Dcapacity=8 -DprodDelayMs=50 -DconsDelayMs=1 -DdurationSec=30*** (Alto CPU).

A screenshot of a computer

AI-generated content may be incorrect.

El apartado de Threads muestra que bajo exigencia inicial el número de hilos crece rápidamente. Sin embargo, entre las 11:16 y 11:17 se observa un descenso que corresponde a la finalización de hilos auxiliares creados durante la fase de carga. Posteriormente, la cantidad de hilos se mantiene estable, lo que evidencia que la JVM libera recursos y conserva únicamente los hilos necesarios para la ejecución estable del proceso.

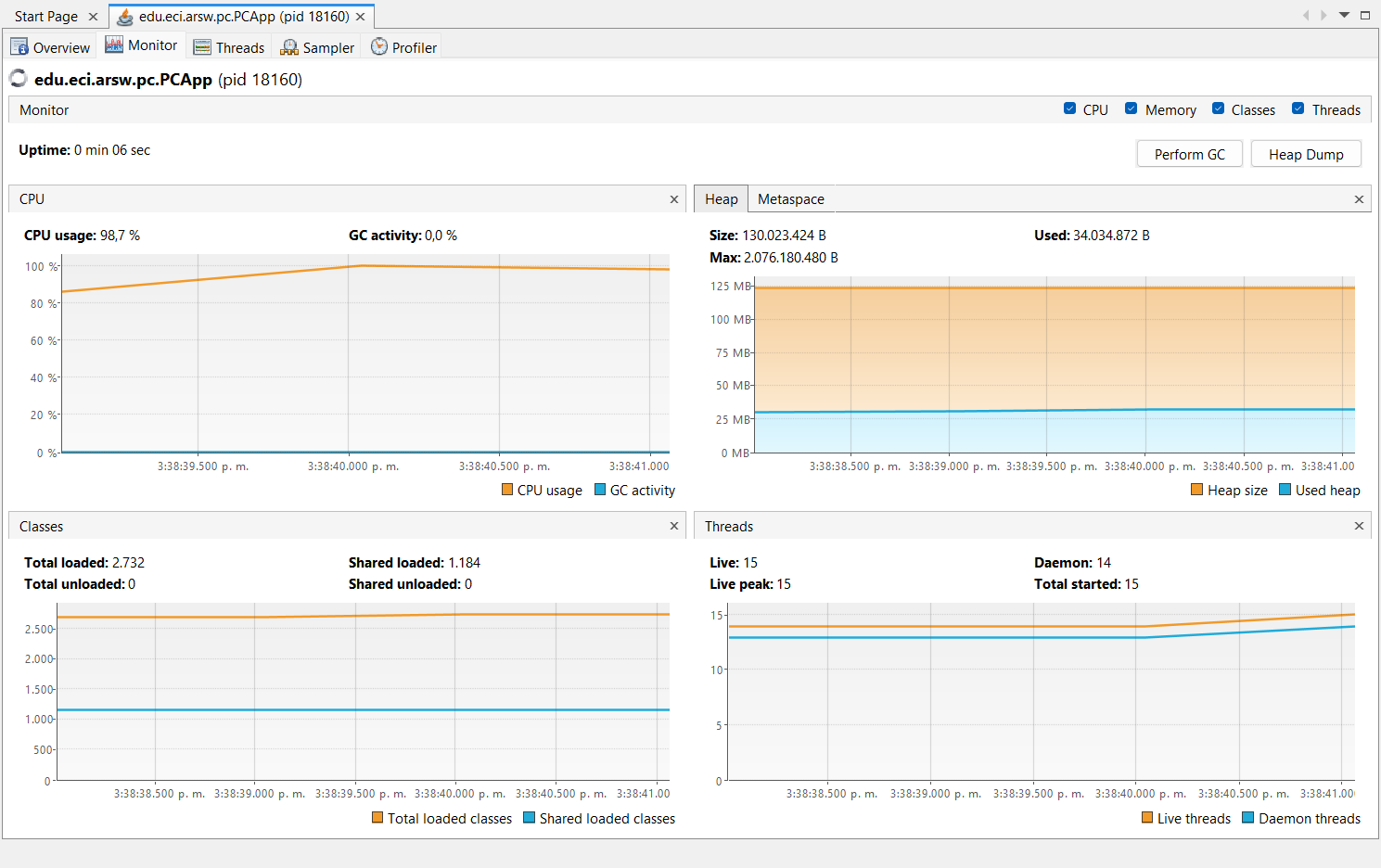
1. Ahora productor rápido y consumidor lento con límite de stock (cola acotada): garantiza que el límite se respete sin espera activa y valida CPU con un stock pequeño.

mvn -q -DskipTests package

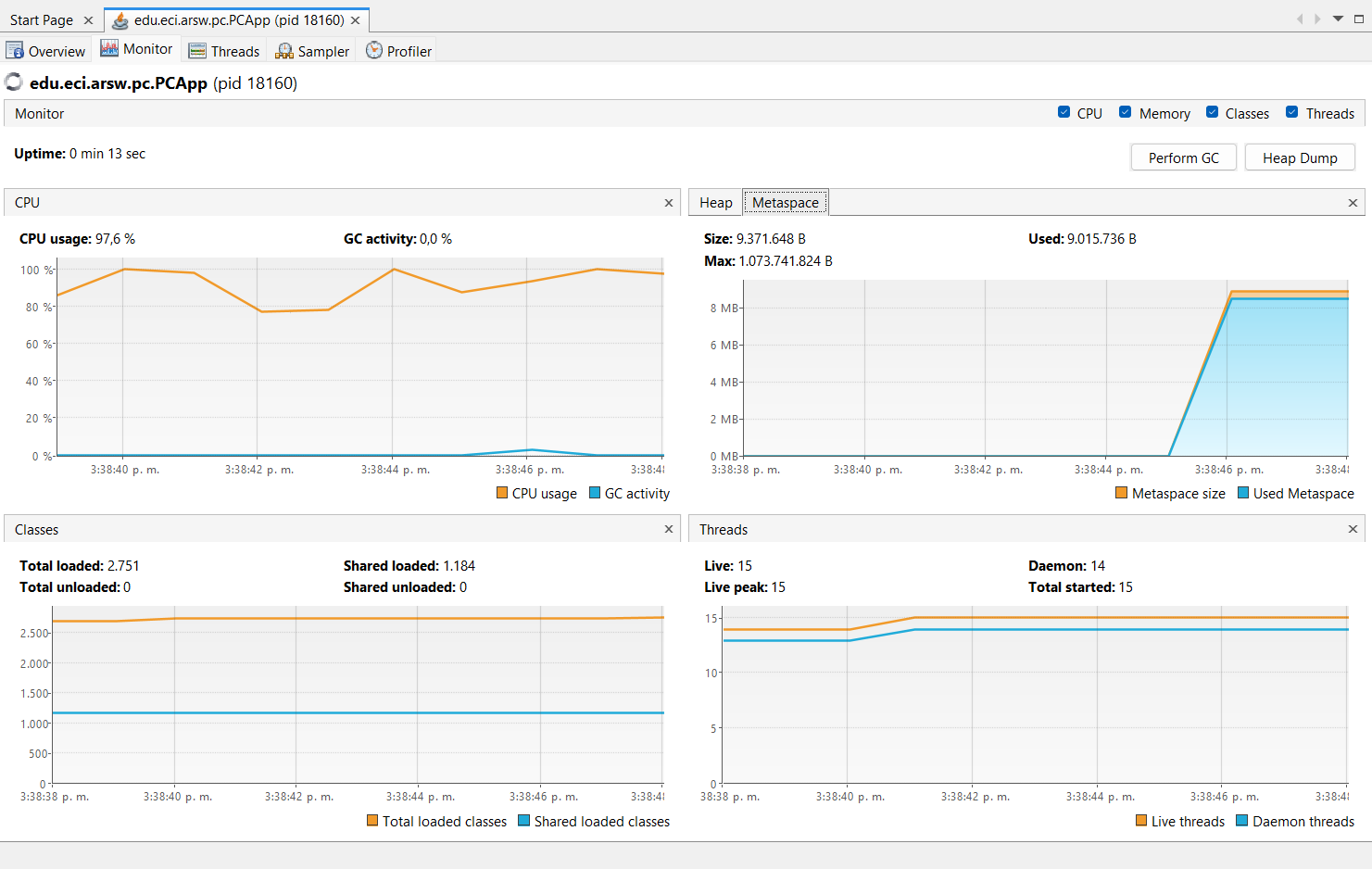
**(modo spin para evidenciar que el proceso queda giratorio, quedan muchos datos en cola puesto que el productor es muy rápido y el consumidor lento)**

java -cp target/classes -Dmode=spin -Dproducers=1 -Dconsumers=1 -Dcapacity=4 -DprodDelayMs=1 -DconsDelayMs=50 -DdurationSec=30 edu.eci.arsw.pc.PCApp

* Podemos evidenciar que tenemos un uso de CPU de 98.7%



* Podemos ver que en la parte superior izquierda una línea de sube y baja este movimiento ocurre puesto que el consumidor procesa cierta cantidad de datos ocasionando que baje, pero el productor produce mas información llenando la cola haciendo que vuelva a subir.



Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

* Al ejecutar el Spin podemos evidenciar que “ForkJoinPool-1-worker-1” es quien ejecuta el spin al quedar llena la cola, esto produce el consumo excesivo de CPU desperdiciando recursos

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

Para ver el contraste podemos ejecutar el mismo comando cambiando spin por monitor para poder ver la diferencia.

* Podemos ver que la CPU tiene un uso de 5% puesto que el productor si se bloquea al llenarse la cola.

Interfaz de usuario gráfica

El contenido generado por IA puede ser incorrecto.

* Con cola llena (capacity=4) y consumidor lento, el productor entra en WAIT dentro de put(); cuando el consumidor libera un slot y hace notify/notifyAll(), el productor despierta y continúa.

Interfaz de usuario gráfica, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

**Nota:** la Parte I se realiza en el repositorio dedicado https://github.com/DECSIS-ECI/Lab\_busy\_wait\_vs\_wait\_notify — clona ese repo y realiza los ejercicios allí; contiene el código de productor/consumidor, variantes con busy-wait y las soluciones usando wait()/notify(), además de instrucciones para ejecutar y validar con jVisualVM.

Usa monitores de Java: synchronized + wait() + notify/notifyAll(), evitando *busy-wait*.

**Búsqueda distribuida y condición de parada (Parte II)**

1. Reescribe el buscador de listas negras para que la búsqueda se detenga tan pronto el conjunto de hilos detecte el número de ocurrencias que definen si el host es confiable o no (BLACK\_LIST\_ALARM\_COUNT). Debe:

Finalizar anticipadamente (no recorrer servidores restantes) y retornar el resultado.

Garantizar ausencia de condiciones de carrera sobre el contador compartido.

Puedes usar **AtomicInteger** o sincronización mínima sobre la región crítica del contador.

**Situación Inicial**

* En la primera versión implementada, cada hilo llevaba un contador local de coincidencias encontradas y al final BlackListChecker tenía que sumar manualmente todos esos resultados

Texto

El contenido generado por IA puede ser incorrecto.

* Esto tenía 2 problemas :
  + Se esperaba a que todos los hilos terminaran incluso cuando ya alguno de los hilos había alcanzado el umbral (threshold)
  + La sincronización de los contadores locales no garantizaba total seguridad frente a la concurrencia

**Cambios Implementados**

* Se creó la variable atómica globalCounter compartida entre todos los hilos. Así, cada vez que un hilo detecta una coincidencia, en lugar de limitarse a su contador local, actualiza este contador global de manera segura.





* Se modificó la condición de la bandera atómica stop. Ahora, en vez de detener el hilo cuando el contador local del hilo superara el threshold, lo detendrá cuando el contador global supere el threshold

Texto

El contenido generado por IA puede ser incorrecto.

* De esta manera, la condición de parada ya no depende de un cálculo posterior sobre los contadores de cada hilo, sino de una verificación inmediata en cada iteración de estos

**Resultados**

* Antes había que recorrer todos los hilos, sumar sus contadores locales y mirar si alguno superó el umbral

Texto

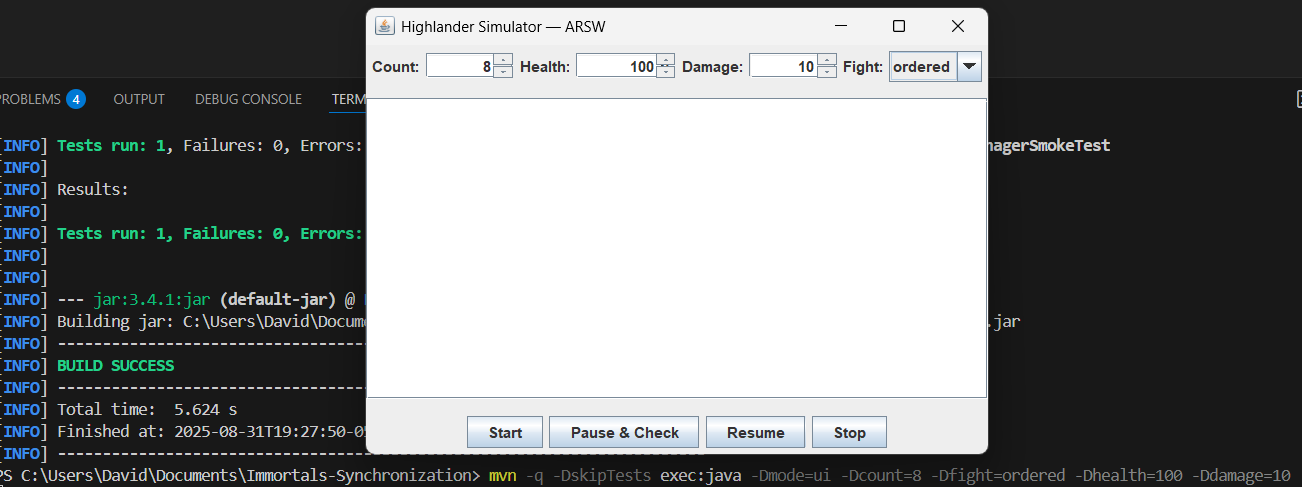
El contenido generado por IA puede ser incorrecto.

* Ahora si el contador globalCounter superó el umbral, automáticamente todos los hilos se detienen y se muestran las IPs no seguras



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

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.



1. Interfaz de usuario gráfica, Aplicación, Tabla

   El contenido generado por IA puede ser incorrecto.Interfaz de usuario gráfica, Tabla

   El contenido generado por IA puede ser incorrecto.Invariante: con N y salud inicial H, la suma total debería permanecer constante (salvo durante un update). Calcula ese valor y úsalo para validar.
   * Acá probamos el proyecto con menos inmortales y menos salud para que sea más fácil ver el invariante. Como empezamos con 4 inmortales y cada uno con 10 de vida, la suma total debería dar 40. En la prueba de la izquierda se ve que incluso hay inmortales con vida negativa y al final la suma queda en 20, o sea claramente algo está mal. En la de la derecha ya no aparecen negativos, pero igual la suma total no da 40, así que sigue sin cumplirse el invariante.
2. Ejecuta la UI y prueba “Pause & Check”. ¿Se cumple el invariante? Explica.
   * El invariante como se expresó en el punto anterior no se cumple y esto es debido a la condición de que ataca M y recibe la vida pero en M/2 para solucionar esto lo que hicimos fue modificar el código en la clase inmortal   
     y cambiar esto para que no se pueda perder más vida después de 0.  
     
3. Pausa correcta: asegura que todos los hilos queden pausados antes de leer/imprimir la salud; implementa Resume (ya disponible).

En la implementación de PauseController se realizaron las siguientes modificaciones para garantizar la pausa correcta de todos los hilos:

* + **Nuevas variables de control**pausedThreads: lleva la cuenta de cuántos hilos están detenidos en un momento dado.
    - totalThreads: define el número total de hilos que deben sincronizarse.
    - allPaused: nueva condición que permite notificar cuando todos los hilos ya están pausados.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

* + **Cambio en awaitIfPaused()**
    - Antes, este método solo hacía que cada hilo esperara mientras la bandera paused estaba activa.
    - Ahora, cada vez que un hilo entra en pausa, incrementa pausedThreads. Si ese número llega a ser igual a totalThreads, se lanza la señal allPaused.signalAll().
    - Al reanudarse (resume ()), el hilo decrementa el contador al salir del await().
    - Esto asegura que el mánager sepa exactamente cuándo todos los hilos están detenidos.

Texto

El contenido generado por IA puede ser incorrecto.

* + **Nuevo método waitUntilAllPaused()**
    - Se añadió para que el controlador pueda bloquearse hasta que pausedThreads == totalThreads. De esta forma, no se leen ni imprimen resultados hasta estar seguros de que absolutamente todos los hilos están pausados.

Texto

El contenido generado por IA puede ser incorrecto.

* + **Nuevo método setTotalThreads(int n)**
    - Permite inicializar el valor de totalThreads según el número de hilos creados. Esto es esencial para que el control de pausa funcione de forma dinámica con distintas cantidades de inmortales.

Texto

El contenido generado por IA puede ser incorrecto.

Además , en el constructor de ImmortalManager se agregó la llamada a pauseController.setTotalThreads(population.size()) para inicializar el número de hilos que deben ser pausados. 

Con esta implementación se evita que la interfaz gráfica se bloquee y se asegura que todos los hilos queden detenidos de forma ordenada antes de leer o imprimir información.

1. Haz *click* repetido y valida consistencia. ¿Se mantiene el invariante?
2. Texto

   El contenido generado por IA puede ser incorrecto.Regiones críticas: identifica y sincroniza las secciones de pelea para evitar carreras; si usas múltiples *locks*, anida con orden consistente:
3. Si la app se detiene (posible *deadlock*), usa jps y jstack para diagnosticar.
4. Aplica una estrategia para corregir el *deadlock* (p. ej., orden total por nombre/id, o tryLock(timeout) con reintentos y *backoff*).
5. Valida con N=100, 1000 o 10000 inmortales. Si falla el invariante, revisa la pausa y las regiones críticas.
6. 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).
   * **Problema**
     + Al manejar la población de inmortales como List<inmortal> , si varios hilos intentaban leer y escribir al mismo tiempo, se producían condiciones de carrera o inclusio excepciones de concurrencia ( ConcurrentModificationException)
   * **Solución concurrencia**
     + En **ImmortalManager** e **Immortal**, se cambió el tipo de colección de List<Immortal> a **ConcurrentLinkedQueue<Immortal>**. Esto permite que:
       1. Varios hilos agreguen o eliminen elementos al mismo tiempo sin necesidad de bloqueos globales.
       2. La simulación pueda seguir corriendo en paralelo, aunque se estén removiendo inmortales muertos.
       3. Además, para mostrar el estado de la población (populationSnapshot()), se devuelve una copia inmutable con Collections.unmodifiableList(new ArrayList<>(population)), asegurando que la GUI solo vea un snapshot estable, sin riesgo de modificaciones concurrentes.



* + - Además, se implementó un nuevo método doFight para los dos tipos de peleas para agregar el caso de:
      1. Cuando un inmortal muere, se elimina directamente de la ConcurrentLinkedQueue con population.remove(this) o population.remove(opponent).

Texto

El contenido generado por IA puede ser incorrecto.

* + - 1. Después de remover, se actualiza el contador en el PauseController con setTotalThreads(population.size()), garantizando que la pausa siga funcionando, aunque se reduzca el número de hilos activos.
      2. Toda esta lógica va en un solo método ya que las dos peleas contemplan el mismo caso de eliminación y no contienen ninguna eliminación especial
  + **Cambios en GUI**
    - Se agregó un método updateDisplay() que obtiene un snapshot de la población viva desde el ImmortalManager y lo muestra en la GUI.

Texto

El contenido generado por IA puede ser incorrecto.

* + - Antes: la GUI no se actualizaba dinámicamente cuando los inmortales morían.
    - Ahora: cuando un inmortal muere y es removido de la lista, automáticamente desaparece de la visualización sin necesidad de detener la simulación.
    - Esto evita que la interfaz quede desincronizada con el estado real del modelo.
  + **Resultado**
    - Ahora los inmortales muertos se eliminan de forma **segura y no bloqueante.**
    - Se evita el uso de sincronización global costosa.
    - La simulación continúa fluida, incluso con muchos hilos peleando y muriendo al mismo tiempo.

1. Implementa completamente STOP (apagado ordenado).
   * **Antes**:
     + El stop() solo llamaba a im.stop() sobre cada inmortal.
     + Usaba exec.shutdownNow(), que mata los hilos abruptamente.
     + No había control de pausa ni garantía de que todos los hilos terminaran bien.
     + Dejaba el population con referencias viejas

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

* + **Ahora:**
    - Se llama a im.stopImmortal() en cada inmortal, deteniendo la lógica de forma controlada.
    - Si los hilos estaban en pausa, se hace resume() para que puedan salir del bloqueo y terminar.
    - Se usa executorService.shutdown() y awaitTermination() para darles tiempo a cerrar ordenadamente.
    - Si alguno no termina, se aplica shutdownNow() como medida de seguridad.
    - Se limpia population.clear() para reiniciar el estado interno del mánager.

Texto

El contenido generado por IA puede ser incorrecto.

* + **Resultado**
    - El apagado ahora es ordenado y seguro, evitando hilos zombis, bloqueos o inconsistencias.
    - La simulación puede reiniciarse limpia después de un stop().

# CONCLUSIONES

# WEBGRAFIA

Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., & Lea, D. (2006). Java concurrency in practice. Addison-Wesley.