Imagen en blanco y negro

Descripción generada automáticamente con confianza baja

Laboratorio 3

Ejercicio – programación concurrente, condiciones de carrera y sincronización de hilos.

Integrantes:

Luisa Fernanda Bermúdez Girón

Karol Daniela Ladino Ladino

Squad:

Inside Out

Profesor:

Javier Iván Toquica Barrera

Curso:

ARSW – 1

Fecha De Entrega:

08-09-2023

1. **Introducción**

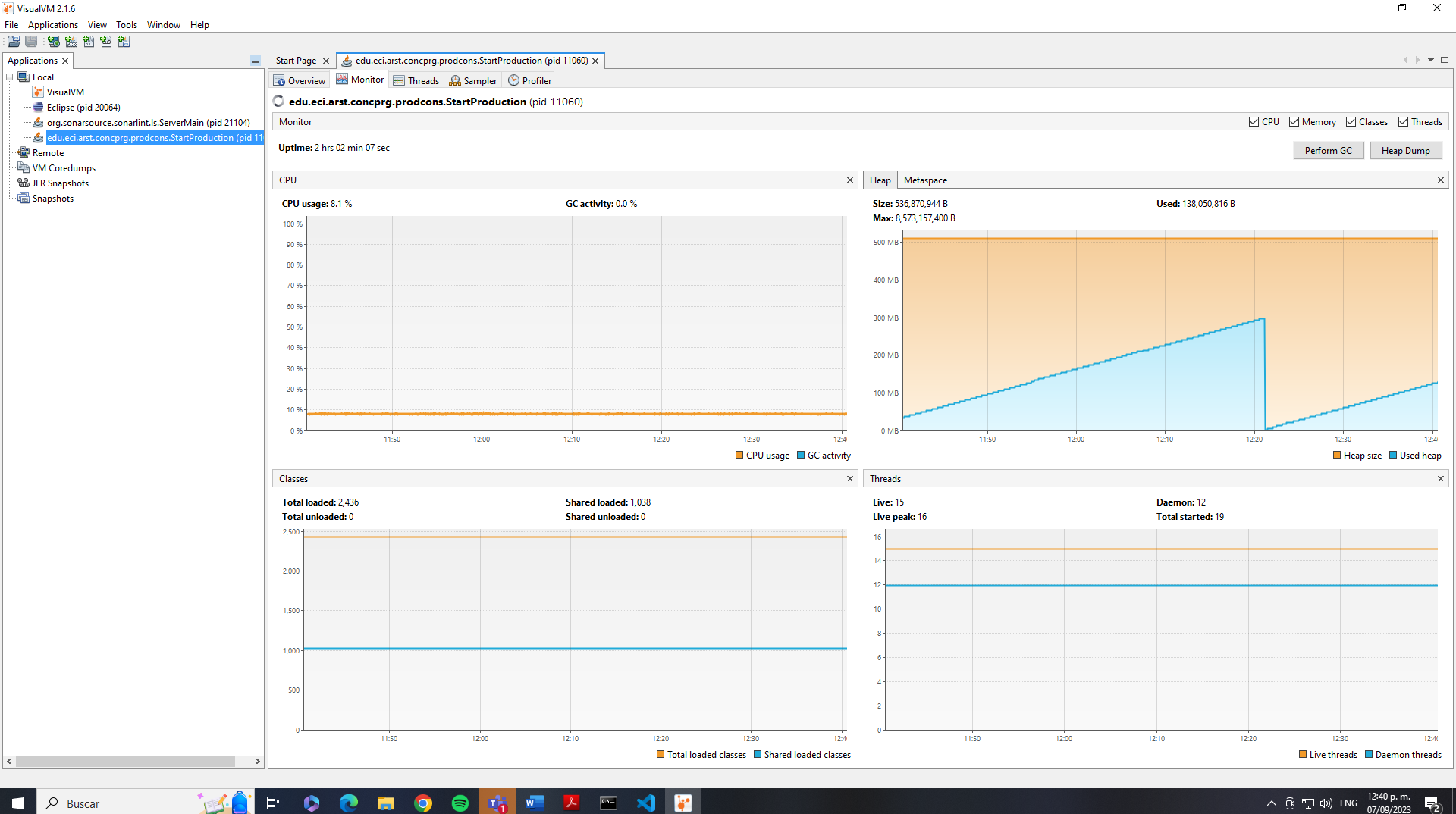
En el ámbito de la programación concurrente y la sincronización de hilos, se presentan desafíos cruciales para el diseño de aplicaciones robustas y eficientes. Este laboratorio se centra en explorar conceptos y técnicas relacionadas con la gestión de hilos, la prevención de condiciones de carrera y la optimización del rendimiento en aplicaciones Java.

1. **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?



Interfaz de usuario gráfica, Aplicación, Tabla

Descripción generada automáticamente

Como podemos observar el pico de la CPU se mantiene por debajo del 10% debido a que la aplicación no consume una gran cantidad de recursos.

Gráfico

Descripción generada automáticamente con confianza baja

Como podemos observar en la gráfica de la memoria tubo un pico a las 12:20, lo cual indica que hubo un consumo mayor de memoria, así mismo en este punto el recolector de basura actúa liberando los objetos que él pueda.

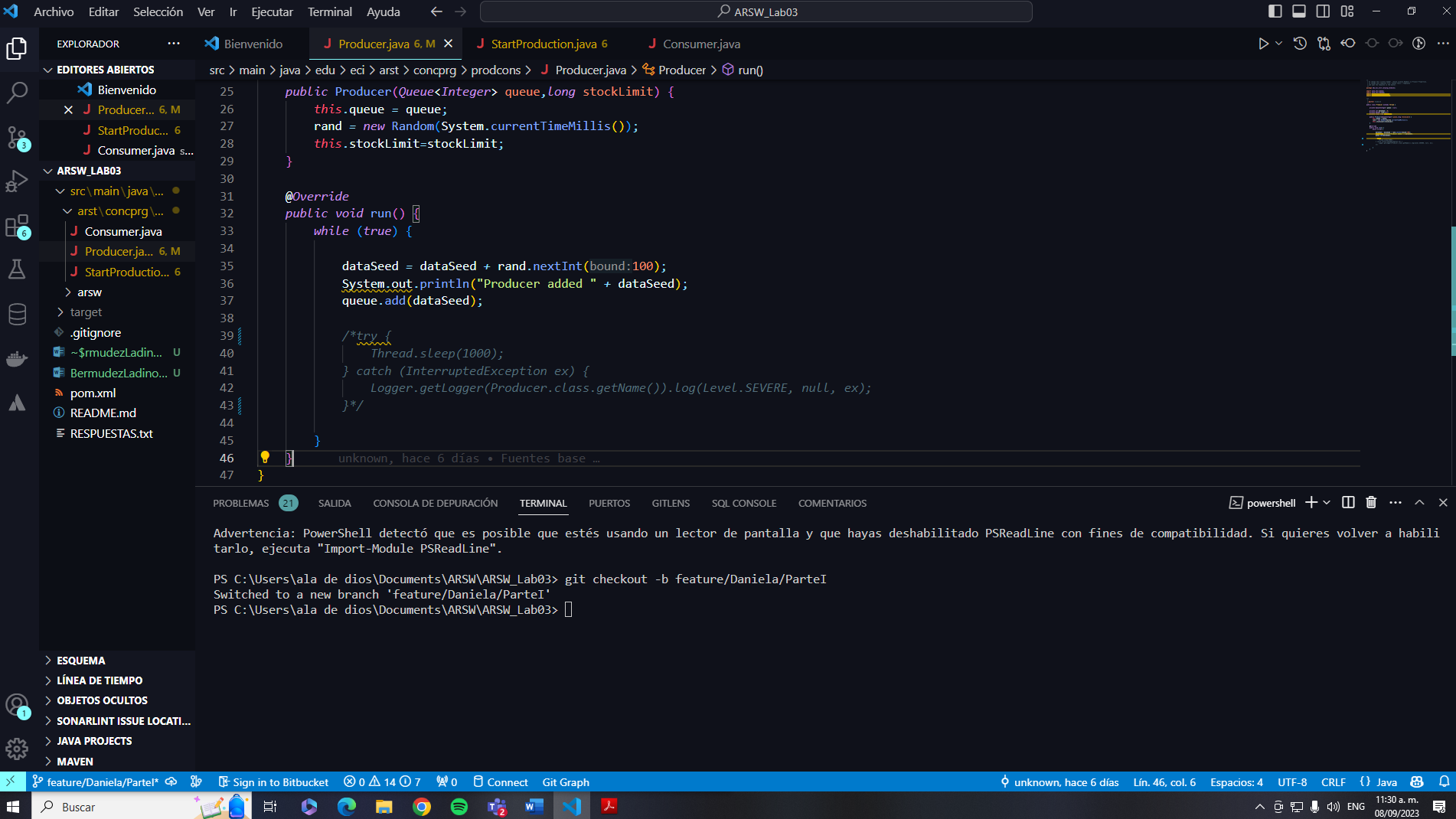
* ¿A qué se debe este consumo?

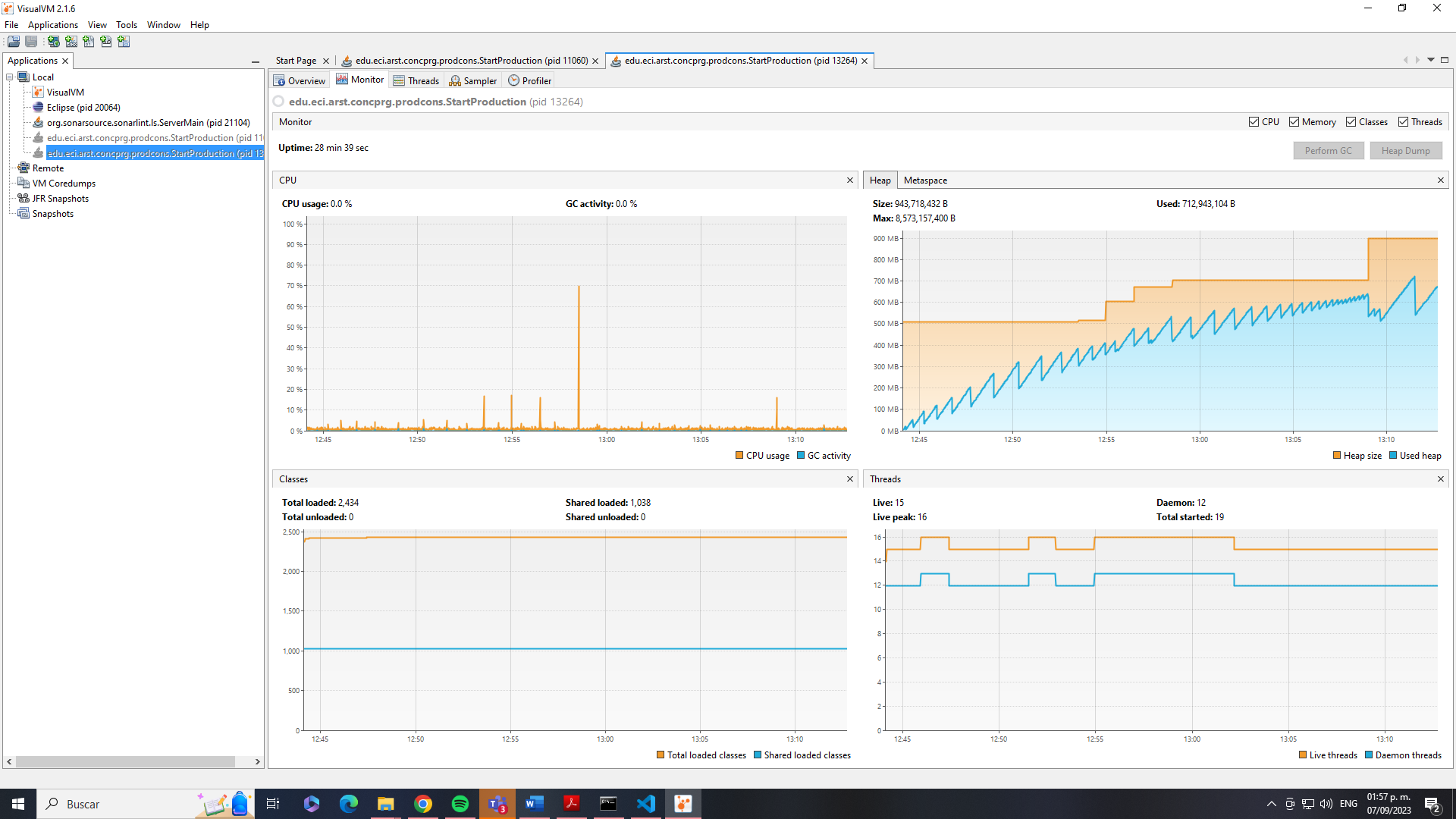
La memoria Heap se va llenando debido a que se está produciendo más de lo que se está consumiendo.

* ¿Cuál es la clase responsable?

La clase responsable es Producer.

1. 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.





Interfaz de usuario gráfica

Descripción generada automáticamente

Como podemos observar la CPU está trabajando más eficientemente debido a que ahora el productor no está esperando y los recursos se están liberando más rápido, es por esto que se presentan picos en un menor tiempo.

Imagen que contiene Gráfico

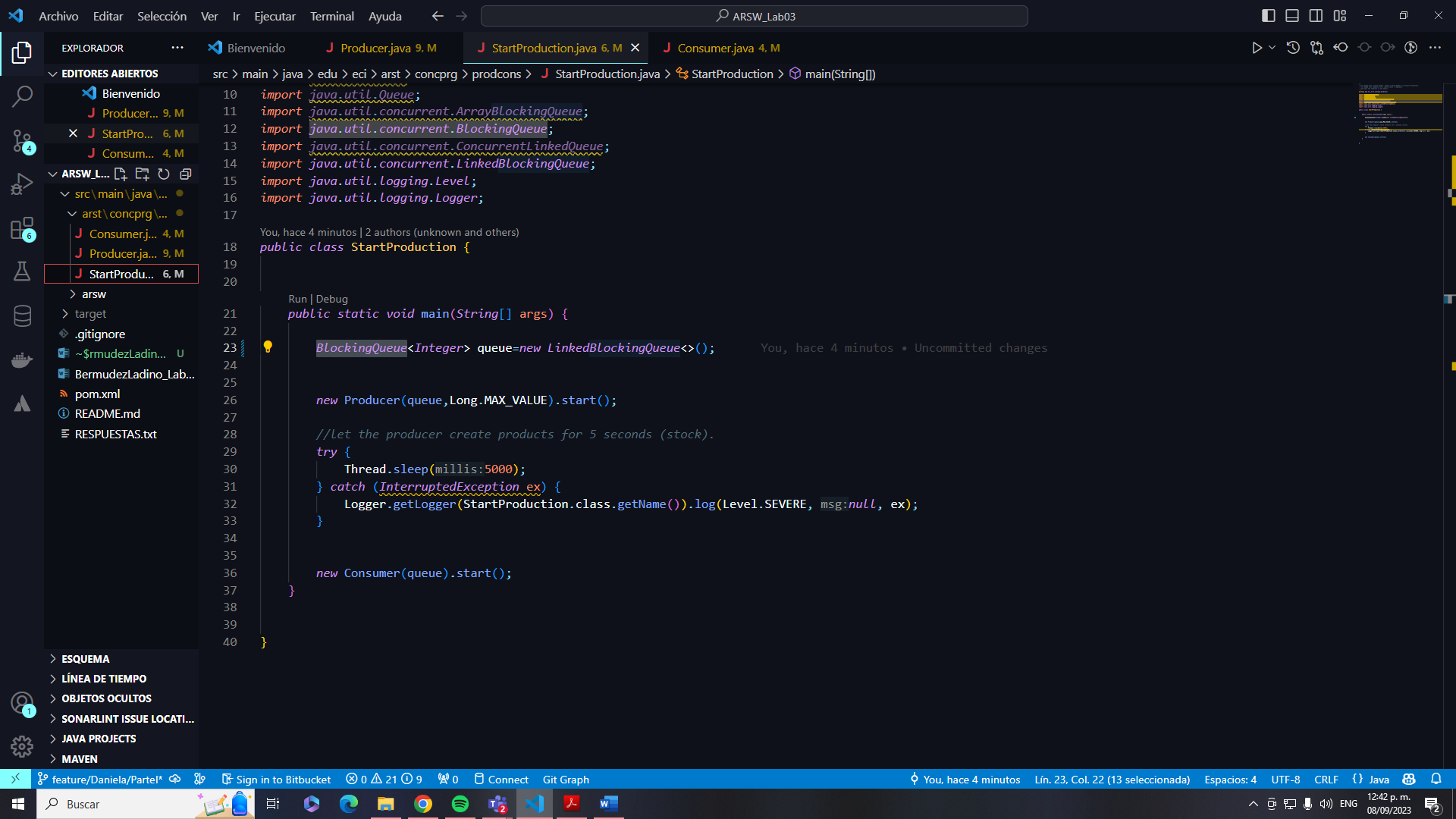
Descripción generada automáticamente

Como podemos observar la memoria registra los picos en un lapso de tiempo más corto y así mismo el recolector de basura libera los objetos de una forma más rápida.

Por otro lado, se puede visualizar que el tamaño del Heap aumento entre las 12:55 y la 1:00 y nuevamente aumenta a la 1:10 debido a que en el momento en que java considere que debe aumentar este lo va a hacer.

1. 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.

* Modificaciones Clase StartProduction:

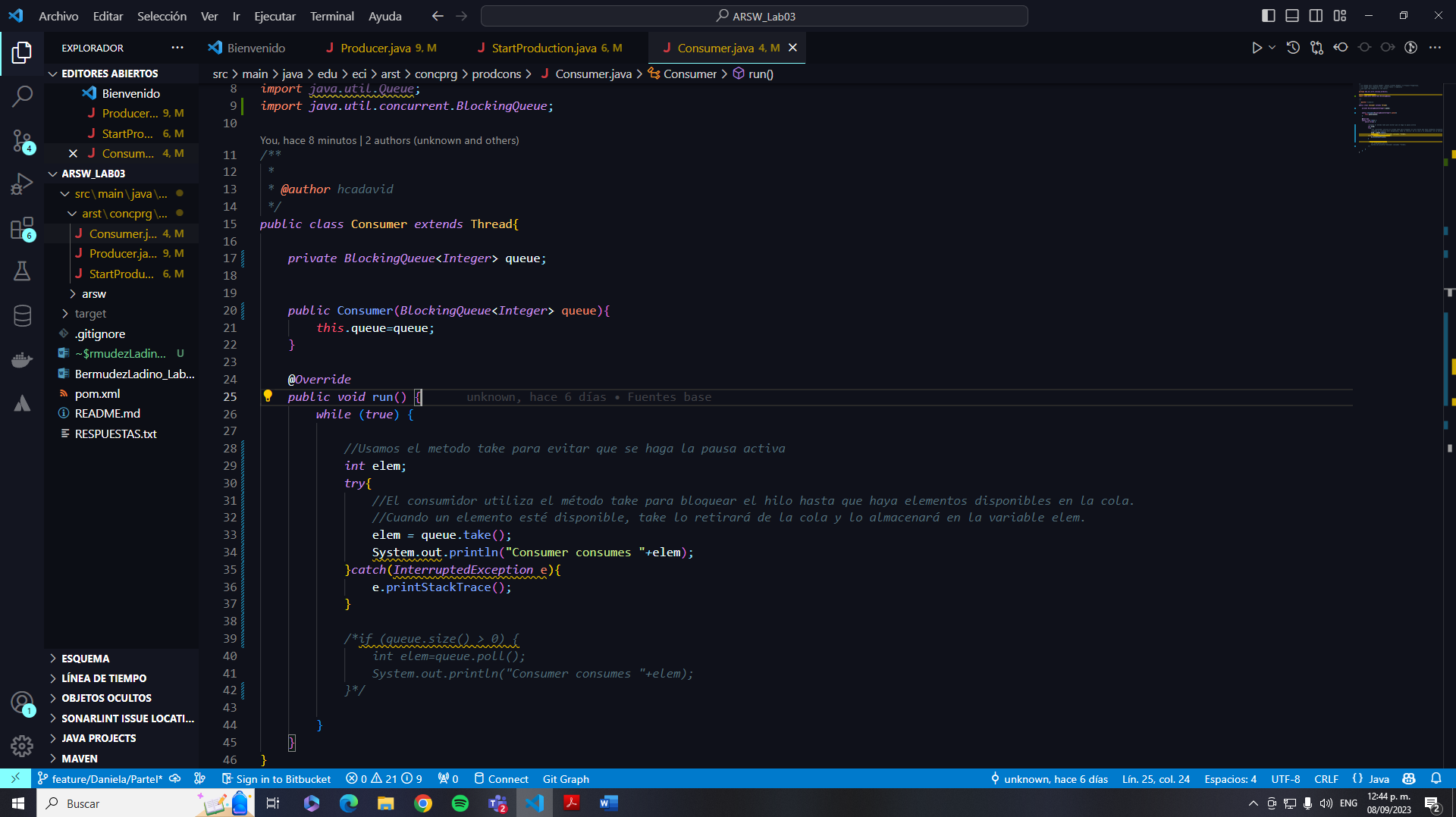


* Modificaciones Clase Producer:

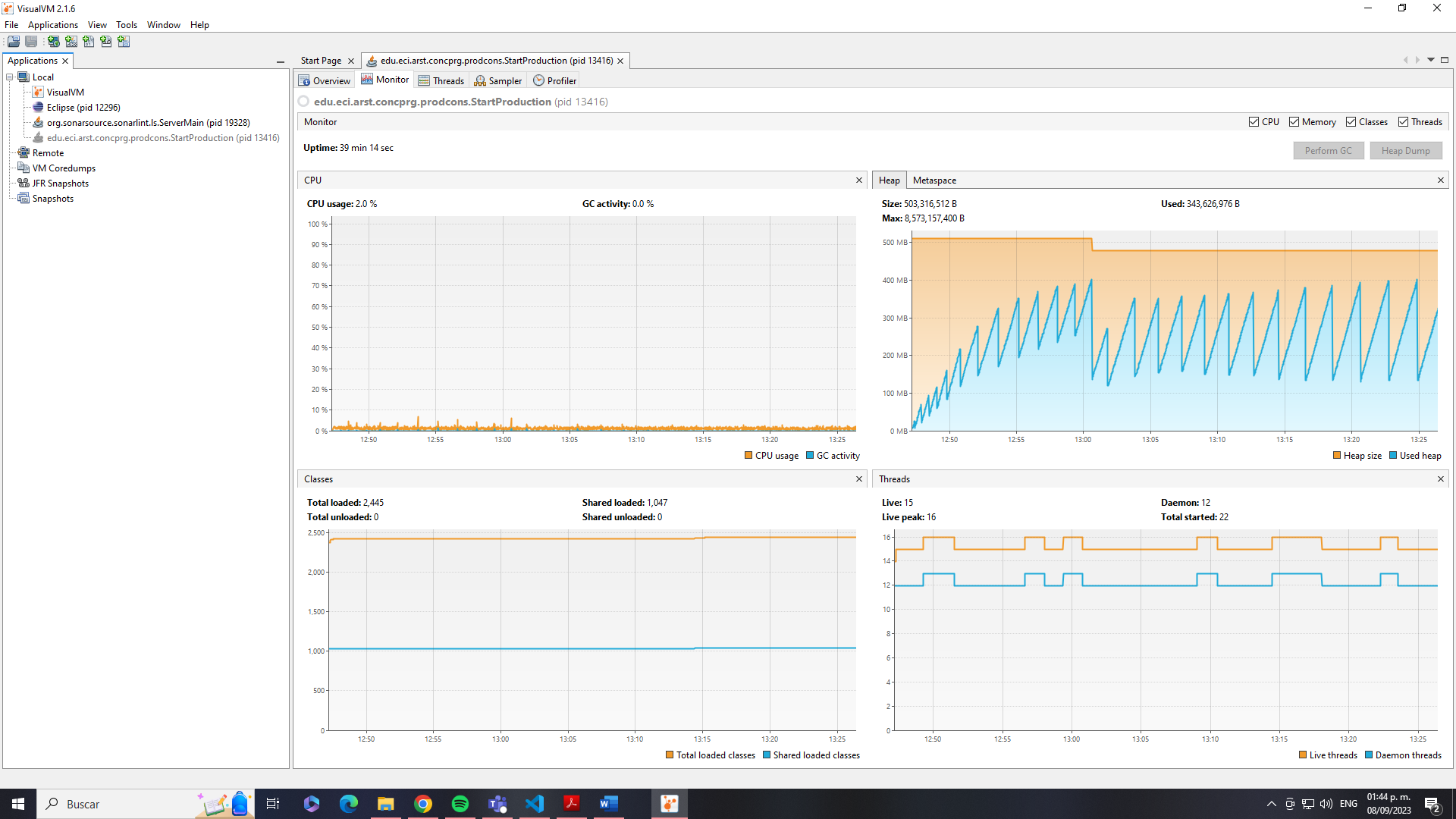
Captura de pantalla de computadora

Descripción generada automáticamente

* Modificaciones Clase Consumer:



* Resultados ejecución:



Interfaz de usuario gráfica, Aplicación, Tabla, Excel

Descripción generada automáticamente

Gráfico, Histograma

Descripción generada automáticamente

**Parte II**

Teniendo en cuenta los conceptos vistos de condición de carrera y sincronización, haga una nueva versión -más eficiente- del ejercicio anterior (el buscador de listas negras). En la versión actual, cada hilo se encarga de revisar el host en la totalidad del subconjunto de servidores que le corresponde, de manera que en conjunto se están explorando la totalidad de servidores. Teniendo esto en cuenta, haga que:

* La búsqueda distribuida se detenga (deje de buscar en las listas negras restantes) y retorne la respuesta apenas, en su conjunto, los hilos hayan detectado el número de ocurrencias requerido que determina si un host es confiable o no (*BLACK\_LIST\_ALARM\_COUNT*).
* Lo anterior, garantizando que no se den condiciones de carrera.

**Parte III**

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, cual debería ser este valor?
3. Ejecute la aplicación y verifique cómo funcionan la opción ‘pause and check’. Se cumple el invariante?.
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’.
5. Verifique nuevamente el funcionamiento (haga clic muchas veces en el botón). Se cumple o no el invariante?.
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){

…

}

}

1. 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.
2. Plantee una estrategia para corregir el problema antes identificado (puede revisar de nuevo las páginas 206 y 207 de *Java Concurrency in Practice*).
3. 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.
4. 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.
5. Para finalizar, implemente la opción STOP.
6. **Conclusiones**