Laboratorio N°3. Programación concurrente, condiciones de carrera y sincronización de hilos

Santiago Cárdenas Amaya

Juan Pablo Fonseca

Escuela Colombiana de ingeniería Julio Garavito

Bogotá 2023

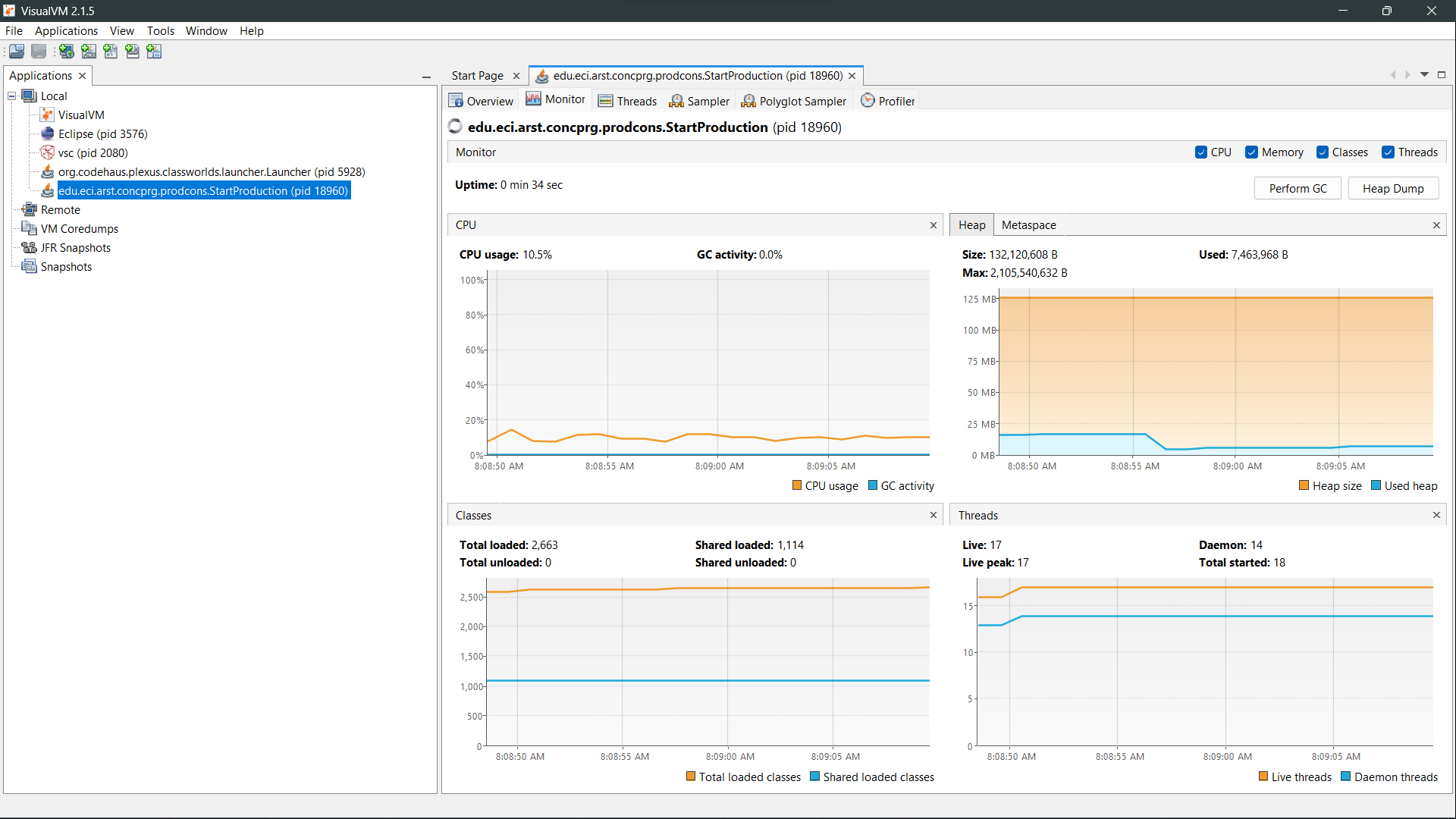
Introducción

Alguna vez se a preguntado como puedo optimizar un programa optimizado aun mas? Cree que su programa es demasiado lento? Esta cansado de tiempos de espera largos? Esta leyendo esto con voz de presentardor?. Tranquilo le tenemos la solución, evite deadlocks haciendo uso de la magnificas variables atomicas y anidando syncronizeds de forma responsable.

Las variables atómicas son útiles en situaciones donde varios hilos de ejecución (threads) acceden y modifican el mismo valor. Si dos o más hilos intentan modificar el valor de una variable al mismo tiempo, pueden producirse condiciones de carrera (race conditions) que generen resultados inesperados. Las variables atómicas evitan estas condiciones de carrera al garantizar que solo una operación se lleve a cabo a la vez.

# Parte I

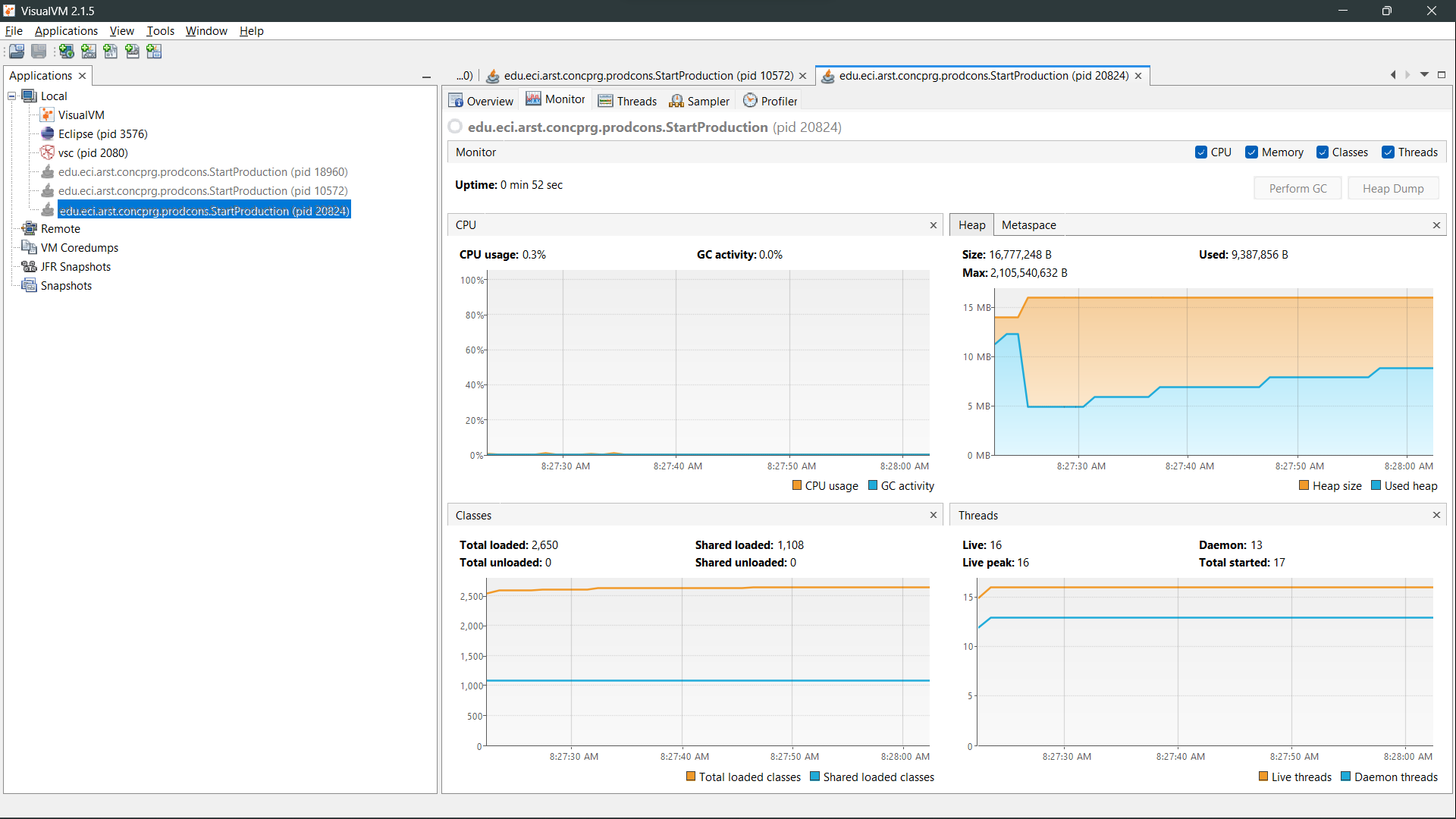
1. Revise el funcionamiento del programa y ejecútelo. Mientras esto ocurren, ejecute jVisualVM y revise el consumo de CPU del proceso correspondiente. ¿A qué se debe este consumo?, cual es la clase responsable?



El promedio del uso de la CPU basado en datos es un 10% aproximadamente. Esto se debe a que ambas clases están intentando ingresar al mismo arreglo y este no está sincronizado.

Tanto consumer como producer son las clases culpables .

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.



Sincronizando el objeto “queue” al que estaban accediendo tanto el consumidor como el productor y haciendo que el consumidor espere mientras se llena la lista y una vez esto pase el consumidor le notifique logramos bajar el consumo de CPU a un promedio de 0.3%

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

# 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.

Se creo un booleano atomico haciendo el papel de observador, el cual una ves que el productor termina de generar sus numeros le notifica al consumidor para que este sepa que debe termianar

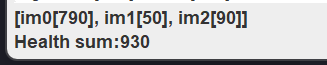
# Parte III

Sincronización y Dead-Locks.

1. Revise el programa “highlander-simulator”, dispuesto en el paquete edu.eci.arsw.highlandersim
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?

El valor debe ser

1. Ejecute la aplicación y verifique cómo funcionan la opción ‘pause and check’. ¿Se cumple el invariante?

 Texto

Descripción generada automáticamente con confianza media

No, no se cumple, en las 2 ejecuciones se obtuvieron 2 valores totalmente distintos.

1. 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’.

Para el tema del valor modificado por varios hilos, inicialmente se pensó que al sincronizar los métodos getHealth() y setHealth() y/o usar un valor atómico para la vida sería suficiente, pero se intentó y no funcionó. Creemos que esto es a causa de que la lista no es concurrente, por lo cual llegamos a la segunda solución, la cual consistió en sincronizar los 2 inmortals en la función fight.

1. Verifique el funcionamiento muchas veces.

Se detuvo en distintos momentos y aún así arrojó el mismo health sum en todos los casos como se puede apreciar en la siguientes imágenes.

Imagen que contiene Logotipo

Descripción generada automáticamente 

1. Identifique posibles regiones críticas en lo que respecta a la pelea de los inmortales.

La región crítica es la lectura y modificación de Immortal.health. Esto se solucionó con lo explicado en el punto 4.

1. Tras implementar su estrategia, ponga a correr su programa, y ponga atención a si este se llega a detener.

Se dejó correr por aproximadamente 5 minutos y en ningún momento se detuvo.

1. Plantee una estrategia para corregir el problema antes identificado (puede revisar de nuevo las páginas 206 y 207 de \_Java Concurrency in Practice\_.

No tuvimos problemas con la ejecución. Algo extra que se podría decir es que inicialmente implementamos la sincronización de los inmortales uno dentro de otro, pero a causa de la aleatoriedad esto tendía a bloquearse, por lo cual se solucionó esto tomando los locks de forma secuencial.

1. Una vez corregido el problema, rectifique que el programa siga funcionando de manera consistente cuando se ejecutan 100, 1000 o 10000 inmortales.

Prueba con 100 inmortales: exitosa.

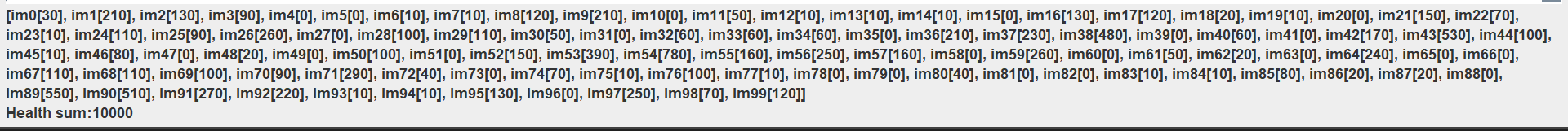
Imagen que contiene Logotipo

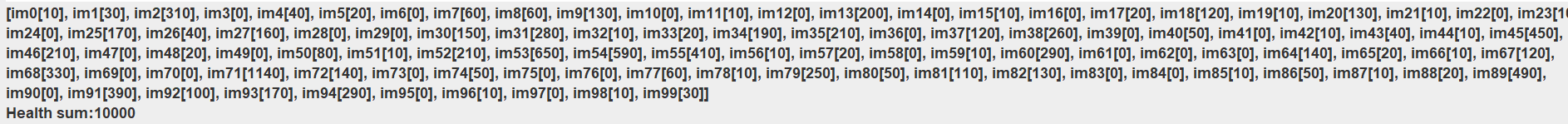
Descripción generada automáticamente

Imagen que contiene Logotipo

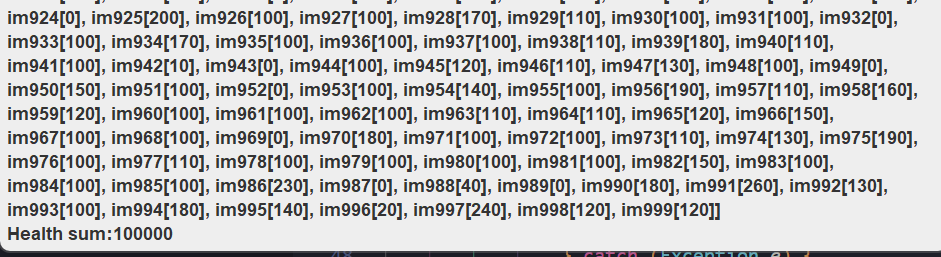
Descripción generada automáticamente

Prueba con 1000 inmortales: se tuvieron inconvenientes, por lo cual se sincronizó el objeto de pausa al momento de pelear, y aparte se sincronizaron los métodos relacionados Immortal.health.





Prueba con 10000 inmortales: exitosa.



Texto

Descripción generada automáticamente

1. Eliminar los inmortales que van muriendo. No pudimos realizarlo sin hacer uso del synchronized.
2. Implemente la función stop.

Función implementada.

# conclusiones

Se puede evidenciar que, el hecho de que los programas funcionen bien con pocos hilos no implica que funcione así con muchos más, puede que no se haya analizado una sección crítica y eso pase factura.