Escuela Colombiana De Ingeniería
Julio Garavito

Javier Ivan Toquica Barrera

Juan Daniel Murcia Sanchez

Juan David Parroquiano Roldan

Andres Felipe Montes Ortiz

Arquitecturas de software

Laboratorio No.3

2024-2

Introducción

Este laboratorio se centra en la programación concurrente, específicamente en el control y la sincronización de hilos. A través de ejercicios prácticos, se busca que los estudiantes comprendan cómo se presentan y resuelven problemas como las condiciones de carrera, el uso eficiente de los recursos del sistema y la prevención de bloqueos o "deadlocks".

El laboratorio abarca múltiples escenarios, desde el clásico problema de productor-consumidor hasta la simulación de peleas entre inmortales, abordando conceptos clave como el uso de mecanismos de sincronización y estrategias de bloqueo para asegurar que las operaciones concurrentes se ejecuten de manera segura y eficiente.

Además, se exploran herramientas como jVisualVM para monitorear el consumo de recursos y se introduce la importancia de manejar correctamente las regiones críticas para evitar errores o sobrecarga en los sistemas multihilo.

Arquitecturas de Software – ARSW

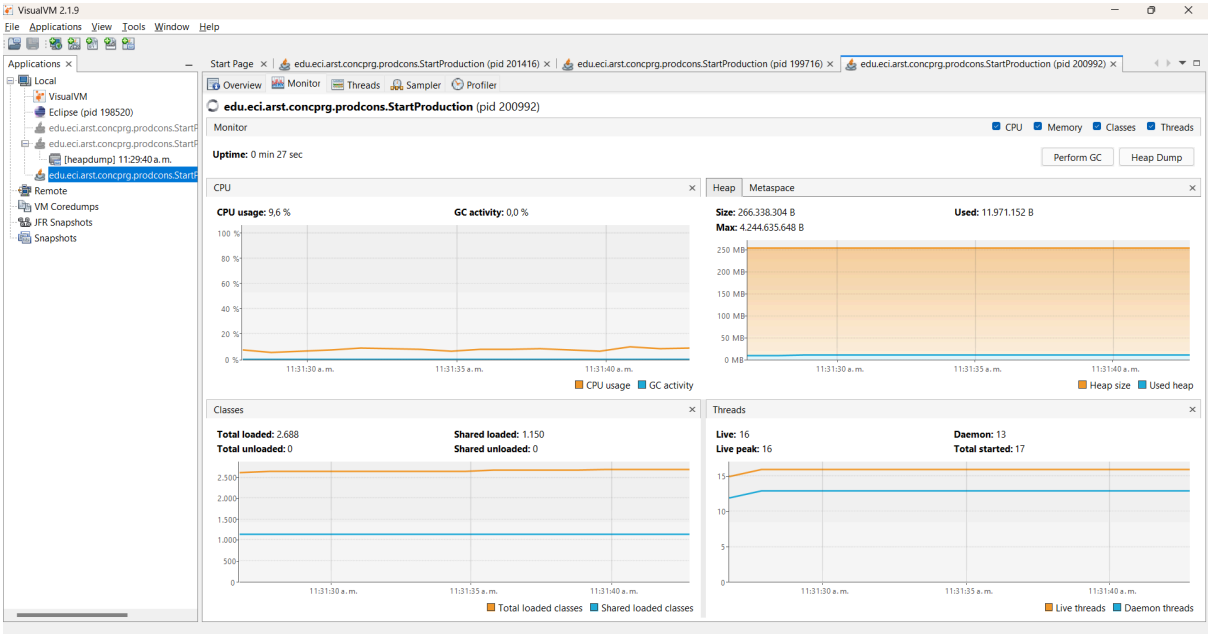
Parte I
Control de hilos con wait/notify. Productor/consumidor.

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

Como podemos apreciar el consumo de cpu es moderadamente alto y las clases responsables de esto son producer y consumer ya que:

Producer: ejecuta un ciclo infinito en el que no deja de producir, esto puede llegar a causar un alto uso en la cpu.

Consumer: como en el anterior esta clase también termina consumiendo mucha cpu debido a que no tiene una manera de verificar cuando la cola está vacía (espera activa) y sigue buscando en un ciclo infinito.



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 VisualVM que el consumo de CPU se reduzca.

Los cambios que realizamos para mejorar el consumo de cpu y el rendimiento general del programa fueron:

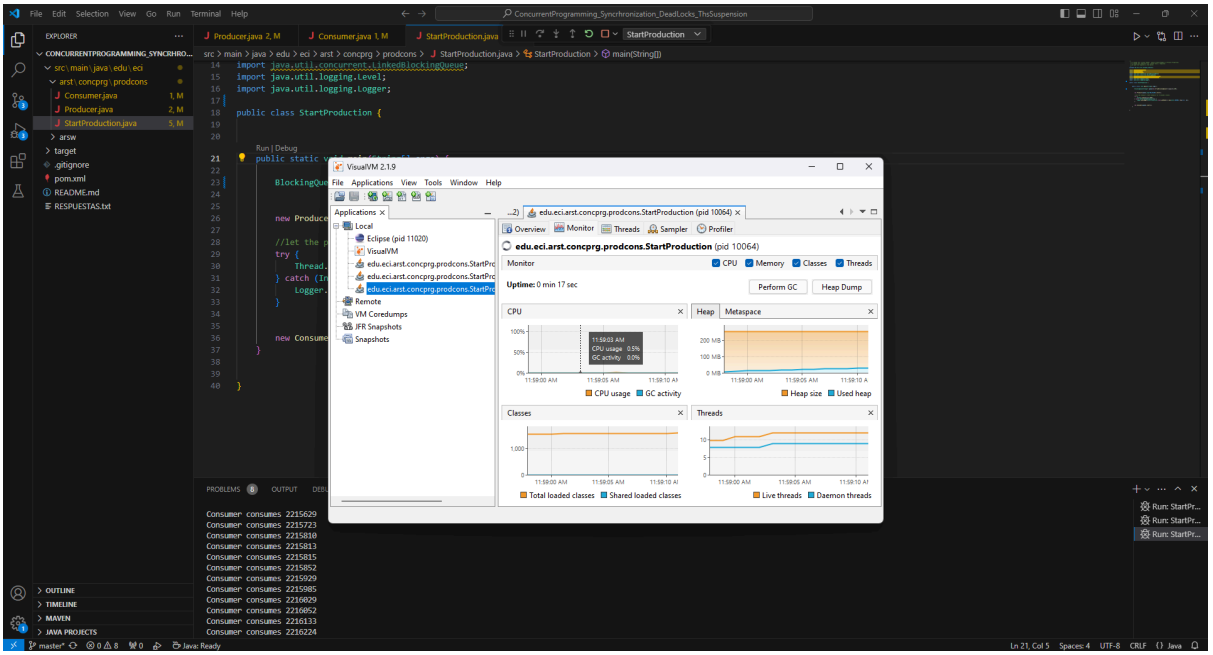
El consumidor ahora utiliza el método queue.take(), lo que bloquea el hilo hasta que haya un elemento disponible en la cola.

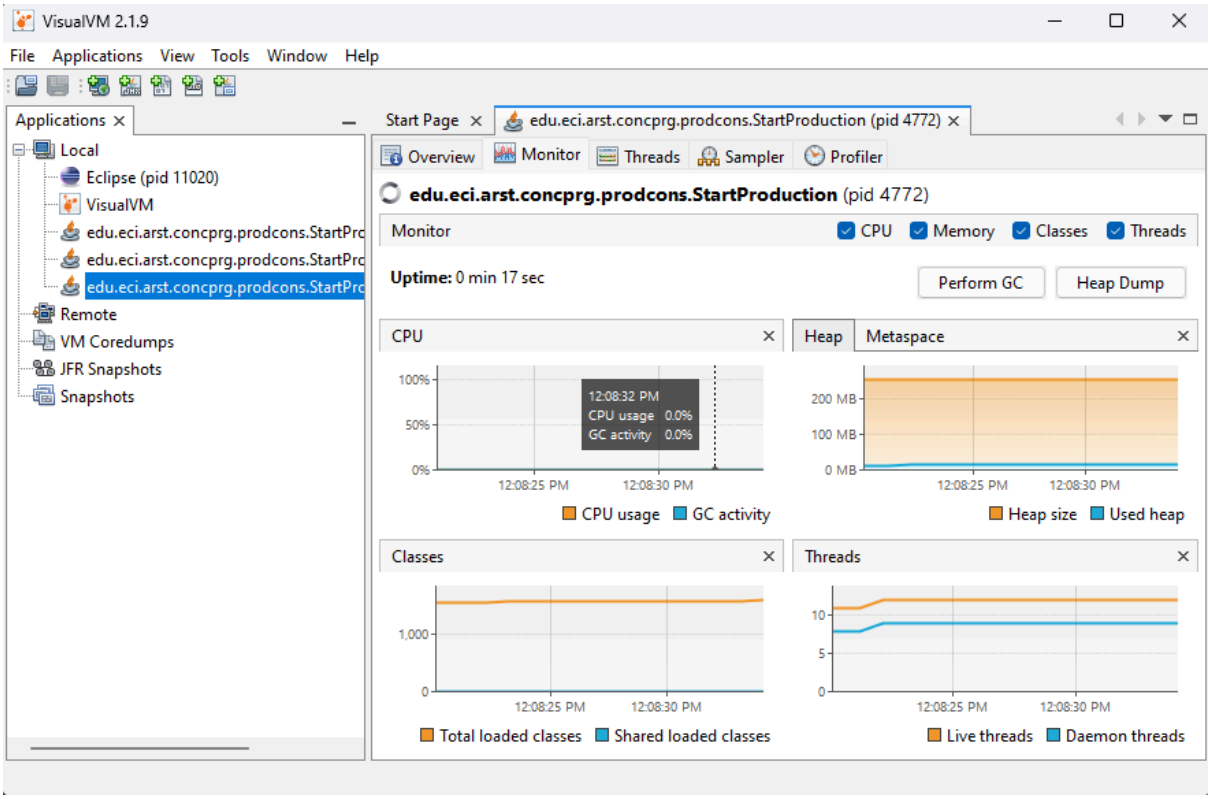
Este bloqueo evita que el consumidor esté constantemente revisando la cola, reduciendo el consumo de CPU.

```
1 usage   murcia0421
public Consumer(BlockingQueue<Integer> queue) { this.queue= queue; }
```

Por último el productor ahora tiene una cantidad límite que producir esto le permite parar cuando acaba y controla restringe el uso de cpu

```
2 usages
private BlockingQueue<Integer> queue = null;

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

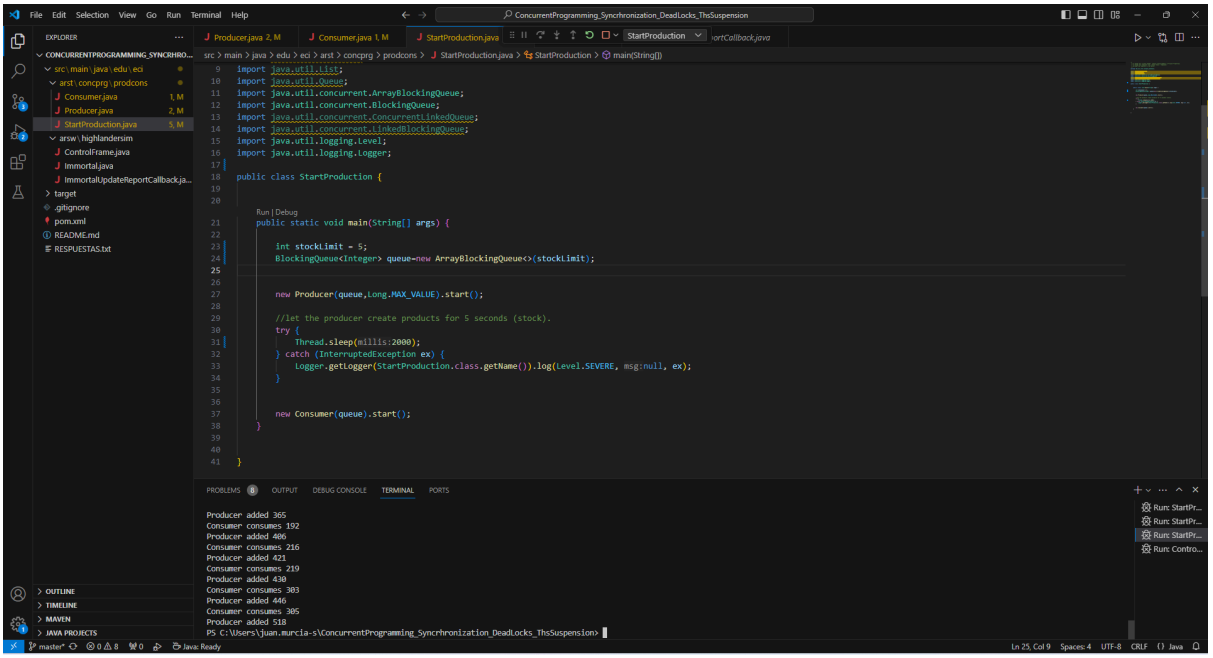




Como podemos apreciar de esta manera el consumo de cpu se redujo prácticamente a 0 lo cual indica un mejor rendimiento del programa.

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.

Ya que el productor tiene un límite en su producción y se optimizó la creación de elementos, solo se agregó una pausa .sleep en el consumidor para que este consuma más lento de esta manera se logra tener un productor muy rápido y un consumidor lento.

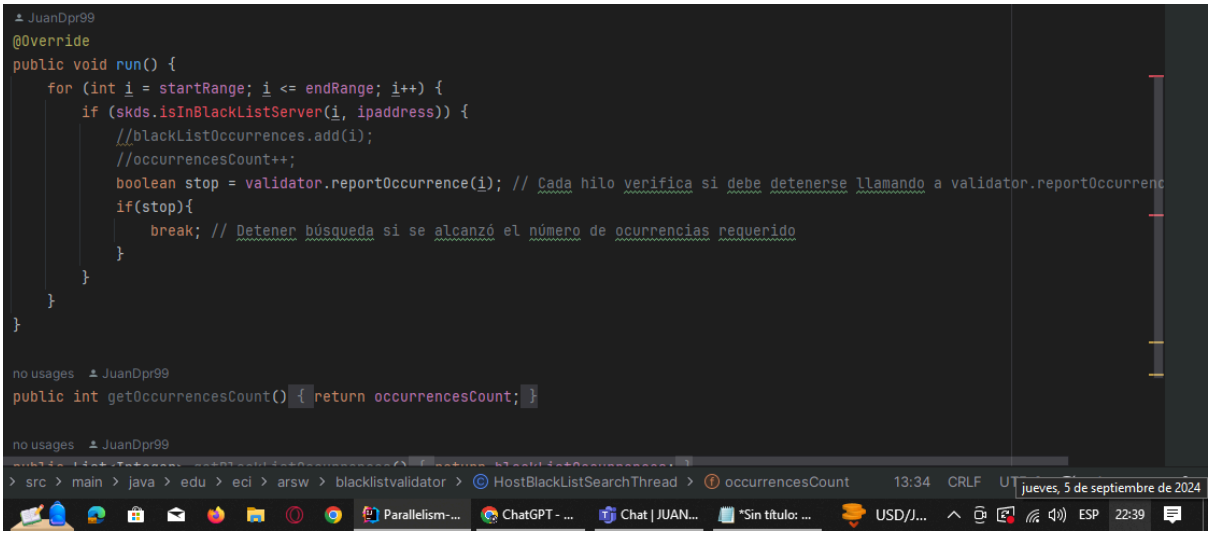


Parte II.

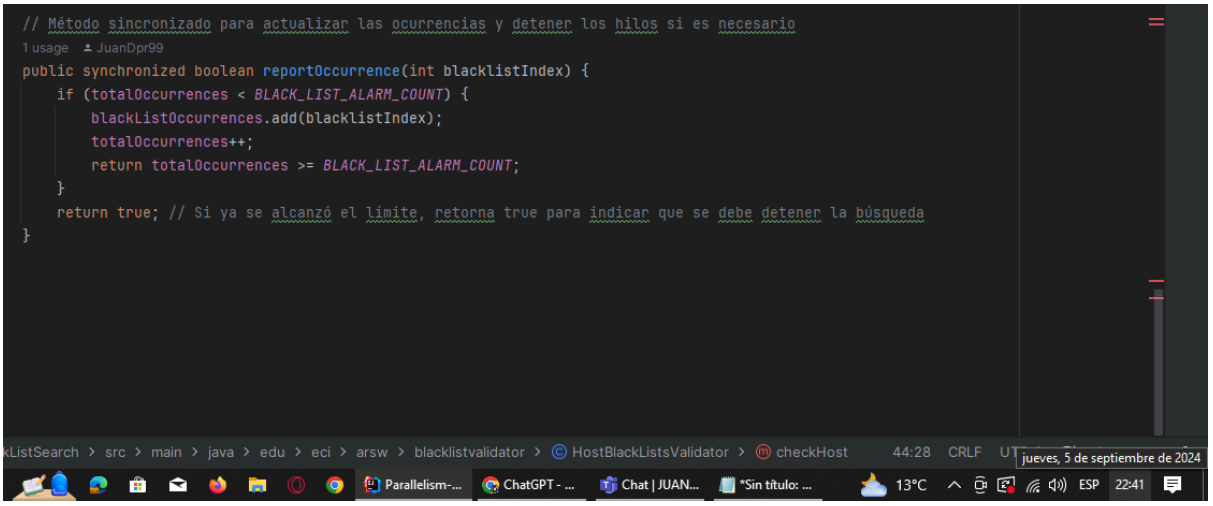
Teniendo en cuenta los conceptos vistos de condición de carrera y sincronización, haga una nueva versión -más eficiente- del ejercicio anterior (el buscador de listas negras). En la versión actual, cada hilo se encarga de revisar el host en la totalidad del subconjunto de servidores que le corresponde,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

1. 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).

En cada hilo se verifica si ya se encontraron los 5 registros y se le avisa al hilo si debe para su ejecución.



Este método es synchronized para asegurar que solo un hilo a la vez pueda actualizar el número total de ocurrencias y la lista de las listas negras donde se encontró la IP
Los hilos verifican este valor de retorno en su ejecución, y si reciben true, terminan su búsqueda, lo que evita seguir procesando listas negras innecesariamente.



2. Lo anterior, garantizando que no se den condiciones de carrera.

Parte III.
Sincronización y Dead-Locks.

- Revise el programa "highlander-simulator", dispuesto en el paquete edu.eci.arsw.highlandersim. Este es un juego en el que:

1. Se tienen N jugadores inmortales.

En clase ControlFrame se leen los números de inmortales ingresados por el usuario:

```java
public List<Immortal> setupInmortals() {

    ImmortalUpdateReportCallback ucb=new TextAreaUpdateReportCallback(output,scrollPane);

    try {
        int ni = Integer.parseInt(numOfImmortals.getText());//Número de inmortales ingresados

        List<Immortal> il = new LinkedList<Immortal>();

        for (int i = 0; i < ni; i++) {
            Immortal i1 = new Immortal("im" + i, il, DEFAULT_IMMORTAL_HEALTH, DEFAULT_DAMAGE_VALUE,ucb);
            il.add(i1);
        }
```

2. Cada jugador conoce a los N-1 jugador restantes.

Cada inmortal se crea con la información de todos los inmortales que existen:

```java
public Immortal(String name, List<Immortal> immortalsPopulation, int health, int defaultDamageValue, ImmortalUpda
    super(name);
    this.updateCallback=ucb;
    this.name = name;
    this.immortalsPopulation = immortalsPopulation;
    this.health = health;
    this.defaultDamageValue=defaultDamageValue;
}
```

3. 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.

Método run de Inmortal contiene un ciclo infinito en donde se llama al método de ataque (fight()).

```java
public void run() {

    while (true) {
        Immortal im;

        int myIndex = immortalsPopulation.indexOf(this);//Posición en el arreglo de Inmortales

        int nextFighterIndex = r.nextInt(immortalsPopulation.size());//Algún inmortal aleatorio dentro del tamaño

        //avoid self-fight
        if (nextFighterIndex == myIndex) {
            nextFighterIndex = ((nextFighterIndex + 1) % immortalsPopulation.size());
        }

        im = immortalsPopulation.get(nextFighterIndex);

        //Lucha contra el inmortal "im"
        this.fight(im);

        try {
            Thread.sleep(millis:5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }
```

4. 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.

Se disminuye el valor del ataque y se aumenta el nivel de vida.

```
public void fight(Immortal i2) {

    if (i2.getHealth() > 0) {
        i2.changeHealth(i2.getHealth() - defaultDamageValue);
        this.health += defaultDamageValue;
        updateCallback.processReport("Fight: " + this + " vs " + i2+"\n");
    } else {
        updateCallback.processReport(this + " says:" + i2 + " is already dead!\n");
    }
```

- 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 la misma(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?.

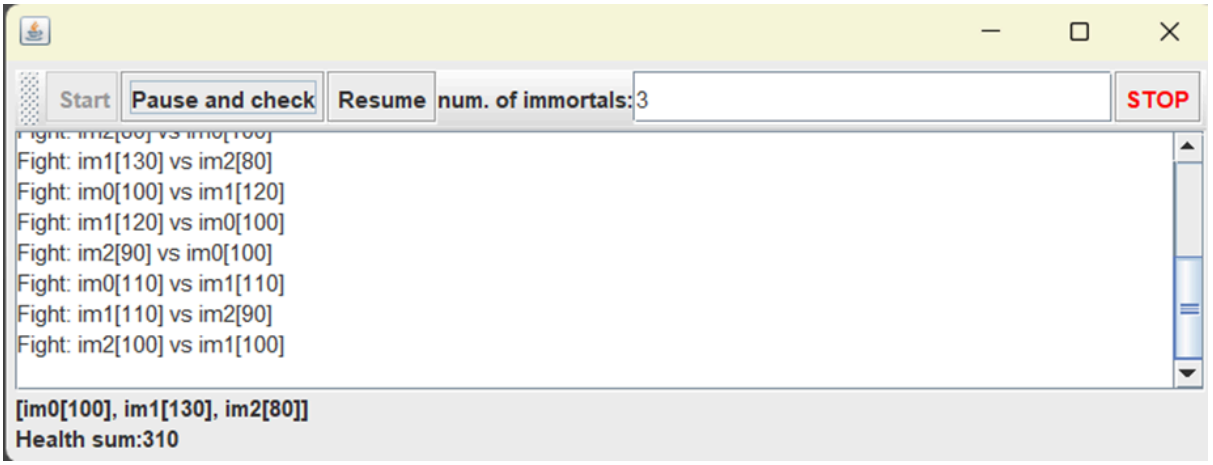
Para un juego con N inmortales, donde cada uno comienza con 100 puntos de vida, la sumatoria invariante de los puntos de vida de todos los jugadores debería ser:

Invariante = N * 100

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

En la ejecución de la aplicación podemos ver que no se está cumpliendo el invariante.

La suma de la salud para este caso con N=3 inmortales, es de 310 lo cual es incorrecto ya que la suma total debe ser 300.



- 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'.

Pausar todos los hilos

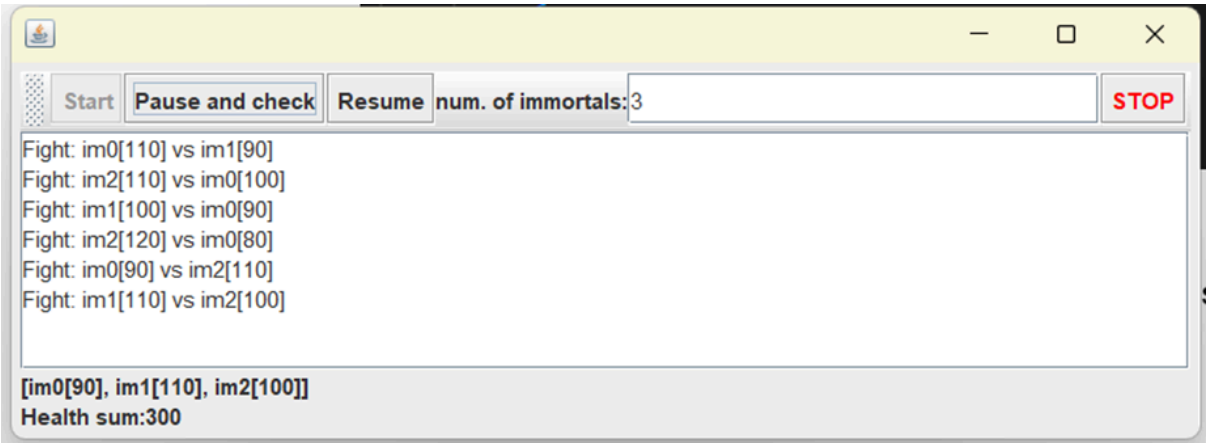
```
//__________Botón Pause and check__________
JButton btnPauseAndCheck = new JButton(text:"Pause and check");
btnPauseAndCheck.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {

        int sum = 0;
        for (Immortal im : immortals) {
            im.setPaused(newPaused:true); //Pausamos hilos
            sum += im.getHealth();
        }

        statisticsLabel.setText("<html>"+immortals.toString()+"<br>Health sum:"+ sum);
    }
});
toolBar.add(btnPauseAndCheck);
```

Implementación botón Resume recorriendo todos los inmortales(hilos) y reanudando cada uno:

```java
//___________Botón Resume___________
JButton btnResume = new JButton(text:"Resume");
btnResume.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        synchronized (immortals) {
            for (Immortal im : immortals) {
                synchronized (im) {
                    im.setPaused(newPaused:false);
                    im.notify();  // Reanuda cada hilo
                }
            }
        }

    }
});
```

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

Efectivamente se cumple con el invariante: Número de hilos 3 -> invariante = 300



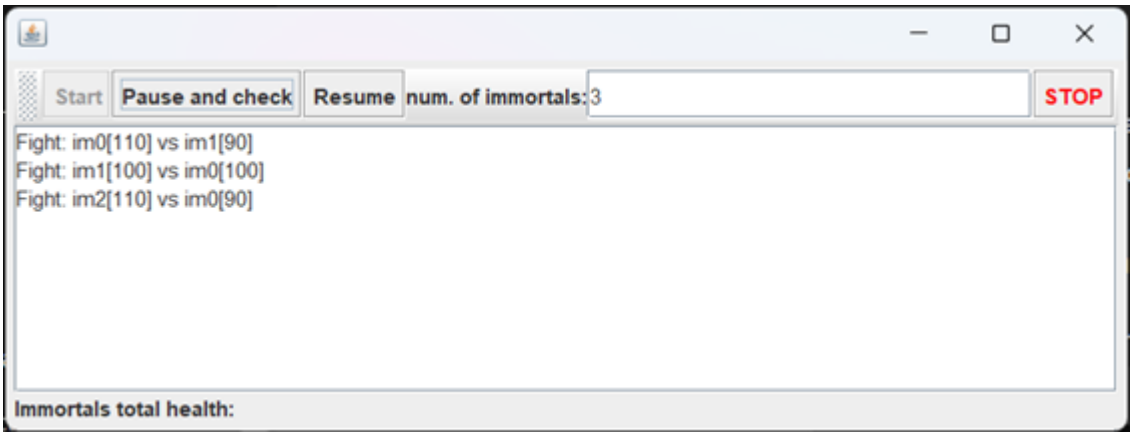
- Identifique posibles regiones críticas en lo que respecta a la pelea de los inmortales. Implemente una estrategia de bloqueo que evite las condiciones de carrera. Recuerde que si usted requiere usar dos o más 'locks' simultáneamente, puede usar bloques sincronizados anidados:

Método fight() se bloquean tanto el hilo que se le suma vida como al que se le resta:

```java
public void fight(Immortal i2) {
    synchronized(this){
        synchronized(i2){
            if (i2.getHealth() > 0) {
                i2.changeHealth(i2.getHealth() - defaultDamageValue);
                this.health += defaultDamageValue;
                updateCallback.processReport("Fight: " + this + " vs " + i2+"\n");
            } else {
                updateCallback.processReport(this + " says:" + i2 + " is already dead!\n");
            }
        }
    }
}
```

- 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.

Ejecutamos nuestro programa y vemos que se detiene:



Abrimos un terminal y ejecutamos el programa jps para identificar el id de muestro proceso Java:

```
C:\Users\JuanDavidParroquiano>jps
58212
67044 ControlFrame
48204 org.eclipse.equinox.launcher_1.6.900.v20240613-2009.jar
66972 Jps
```

Una vez identificado nuestro ID de proceso ejecutamos el programa ‘jstack’ para mostrar una traza completa de los hilos que están ejecutándose en ese momento.

Se encuentra un deadlock (bloqueo mutuo), y nos dice lo siguiente:

```
Found one Java-level deadlock:
=============================
"im0":
  waiting to lock monitor 0x000002884b5ff300 (object 0x00000007124e7180, a edu.eci.arsw.highlandersim.Immortal),
  which is held by "im1"

"im1":
  waiting to lock monitor 0x000002884b5ff220 (object 0x00000007124e7348, a edu.eci.arsw.highlandersim.Immortal),
  which is held by "im2"

"im2":
  waiting to lock monitor 0x000002884b5ff300 (object 0x00000007124e7180, a edu.eci.arsw.highlandersim.Immortal),
  which is held by "im1"
```

El inmortal1(hilo) y el inmortal2(hilo) se están bloqueando mutuamente lo que provoca que el programa no avance.

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

Estrategia de solución:

1. En cada Inmortal creamos un objeto nuevo para identificar el bloqueo del hilo de manera compartida

```java
private boolean paused = false;// Control de pausa
private static final Object pauseLock = new Object(); // Bloqueo compartido
```

```java
public class Immortal extends Thread {
    public void run() {
        while (true) {
            //Establecer condiciones de carrera
            synchronized(pauseLock){
                while (paused) {
                    try {
                        pauseLock.wait();// Poner en espera hasta que sea notificado
                    }catch(InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
```

1. En botón ‘Pause and check’ controlamos el pausar de los hilos con ayuda de un método setPaused de cada hilo:

Método que setea la variable ‘paused’ para indicar si el hilo esta pausado o no. Lo llamamos en la ejecución del botón ‘Pause and check’

```
public class Immortal extends Thread {
    }

    public void setPaused(boolean newPaused){
        this.paused = newPaused;
    }
```

```
//__________Botón Pause and check__________
JButton btnPauseAndCheck = new JButton(text:"Pause and check");
btnPauseAndCheck.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {

        for (Immortal im : immortals) {
            im.setPaused(newPaused:true); //Pausamos hilos
        }

        int sum = 0;
        for (Immortal im : immortals) {
            sum += im.getHealth(); //Suma total salud
        }

        statisticsLabel.setText("<html>"+immortals.toString()+"<br>Health sum:"+ sum);
    }
});
toolBar.add(btnPauseAndCheck);
```

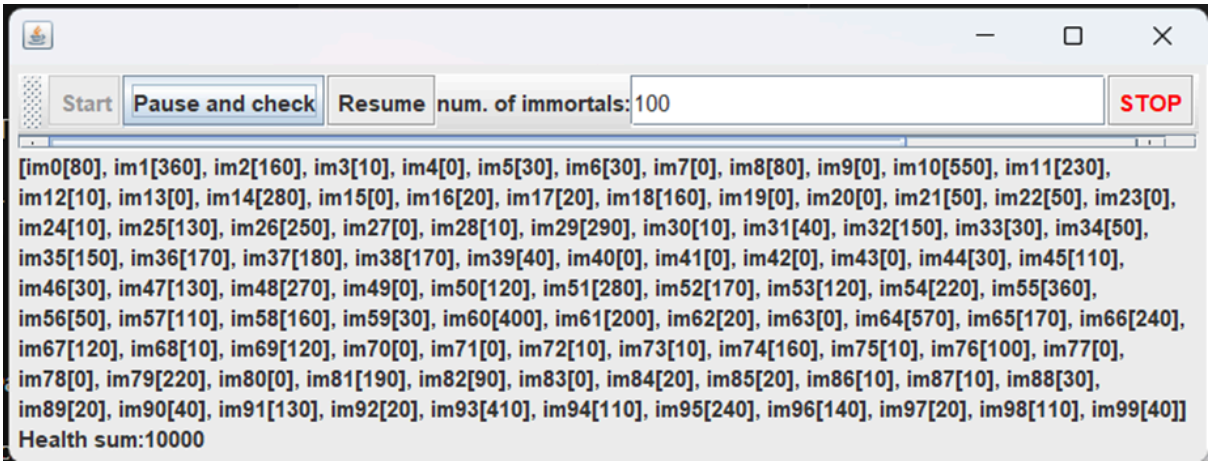
1. Controlamos los bloqueos de los hilos

```java
public void fight(Immortal i2) {
    Immortal first, second;

    // Determinar el orden de bloqueo para evitar deadlock
    if(this.threadId() < i2.threadId()){
        first = this;
        second = i2;
    } else {
        first = i2;
        second = this;
    }

    synchronized(first){
        synchronized(second){
            if (i2.getHealth() > 0) {
                i2.changeHealth(i2.getHealth() - defaultDamageValue);
                this.health += defaultDamageValue;
                updateCallback.processReport("Fight: " + this + " vs " + i2+"\n");
            } else {
                updateCallback.processReport(this + " says:" + i2 + " is already dead!\n");
            }
        }
    }
}
```

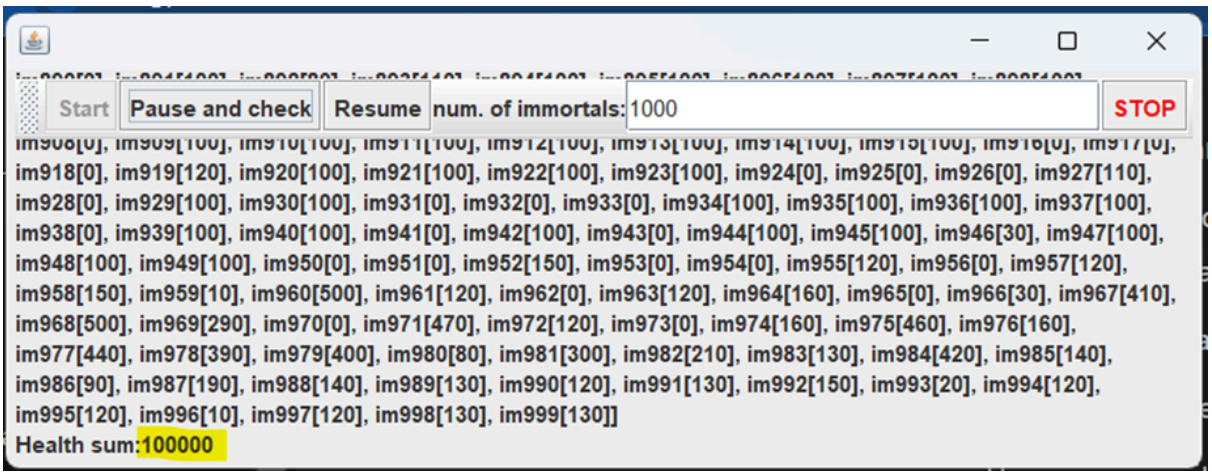
Siempre bloqueamos primero el hilo que fue credo primero por la aplicación para que no hayan deadlocks

- 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.

- con 100 hilos ejecución correcta.



- Con 1000 hilos ejecución correcta:



- 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:

1. 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.

2. 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.

```
        // avoid self-fight
        if (nextFighterIndex == myIndex) {
            nextFighterIndex = ((nextFighterIndex + 1) % immortalsPopulation.size());
        }

        im = immortalsPopulation.get(nextFighterIndex);

        // Lucha contra el inmortal "im"
        this.fight(im);

        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        if (this.health <= 0) {
            immortalsPopulation.remove(this);
            updateCallback.processReport(this.name + " ha muerto y ha sido eliminado de la simulación.\n");
            break;
        }

    }
}
```

- Para finalizar, implemente la opción STOP.

Ejecutamos el programa y luego damos a STOP y vemos que detiene todos los hilos: