

Laboratorio 3 - programación concurrente, condiciones de carrera y sincronización de hilos.

Andrés Felipe Arias Ajiaco

Cesar David Amaya Gómez

Johan Sebastián Gracia Martínez

Sebastián David Blanco Rodríguez

Universidad Escuela Colombiana de ingeniería Julio Gravito

Arquitecturas de software

Ing. Javier Iván Toquica Barrera

16 de febrero de 2024

Introducción

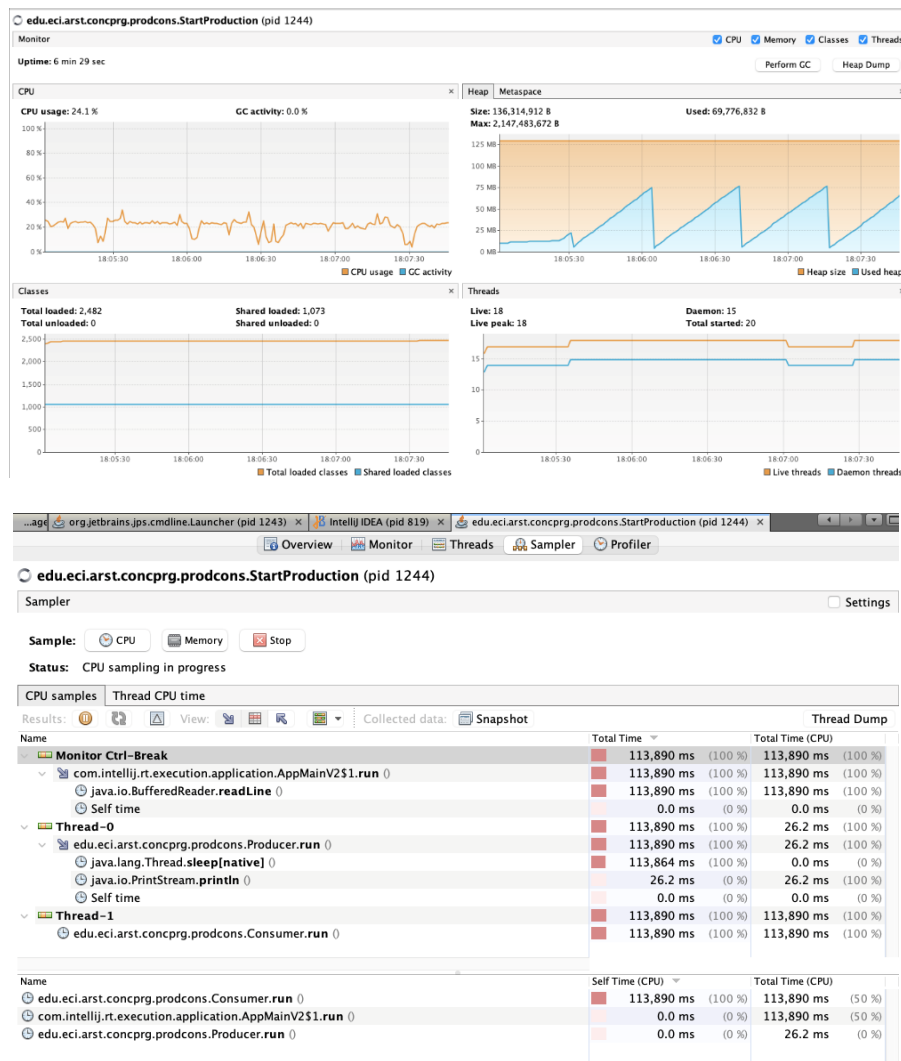
En este informe se aborda la complejidad de la programación concurrente en Java, centrándose en dos problemas fundamentales, el control de hilos en un escenario de productor/consumidor y la gestión de condiciones de carrera en un juego de simulación de combate entre inmortales.

En la primera parte del laboratorio, se revisará y ajustará un programa que implementa un modelo de productor/consumidor, optimizando su eficiencia y controlando el acceso a recursos compartidos. Luego, se analizará un juego llamado "highlander-simulator", donde múltiples hilos representan a inmortales que se enfrentan entre sí, y se identificarán y solucionarán posibles condiciones de carrera que puedan surgir durante la ejecución del programa.

Desarrollo del laboratorio

PARTE 1

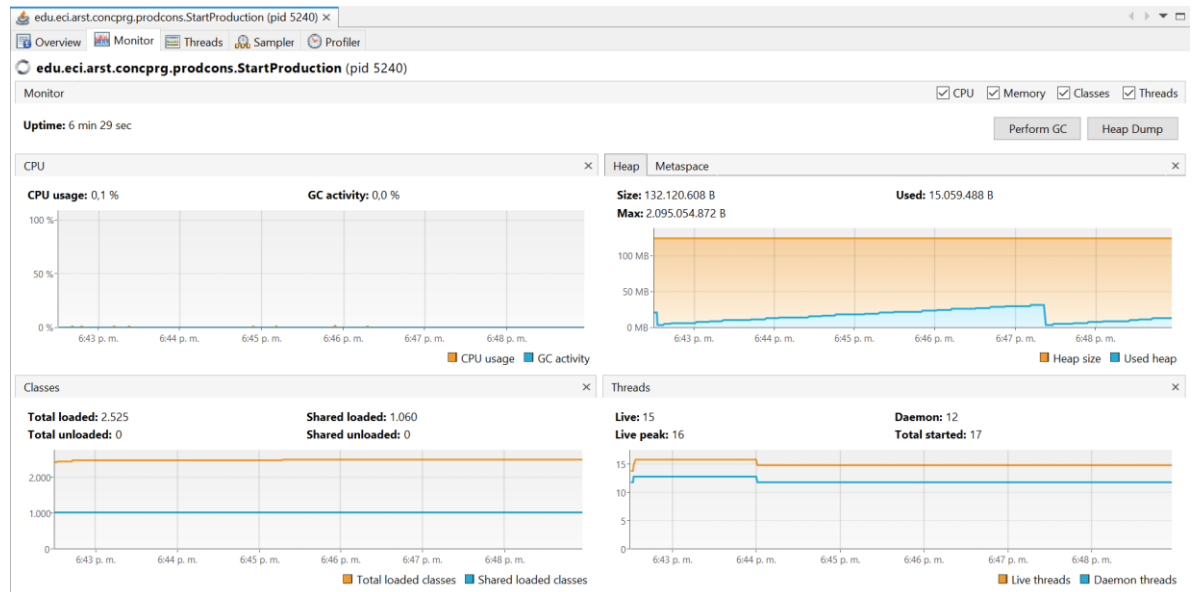
1. Revise el funcionamiento del programa y ejecútelo. Mientras esto ocurren, ejecute jVisualVM y revise el consumo de CPU del proceso correspondiente. ¿A qué se debe este consumo? ¿cuál es la clase responsable?



Rta: El consumo se debe a que la cola es larga y por esto tiene que estar verificando más información y la clase responsable es **Consumer**.

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.



```
25  @Override
26  public void run() {
27      while (true) {
28
29          if (queue.size() > 0) {
30              int elem=queue.poll();
31              System.out.println("Consumer consumes "+elem);
32          }
33          try {
34              Thread.sleep(millis:1000);
35          } catch (InterruptedException ex) {
36              Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, msg:null, ex);
37          }
38      }
39  }
40
41 }
```

Rta: Dado que la producción era lenta y el consumo era rápido, el consumidor podía ejecutarse más rápido lo que generaba un cuello de botella en el uso de la CPU, la solución implementada fue equilibrar los tiempos de la clase consumidor y productor.

3. Haga que ahora el productor produzca muy rápido, y el consumidor consuma lento.

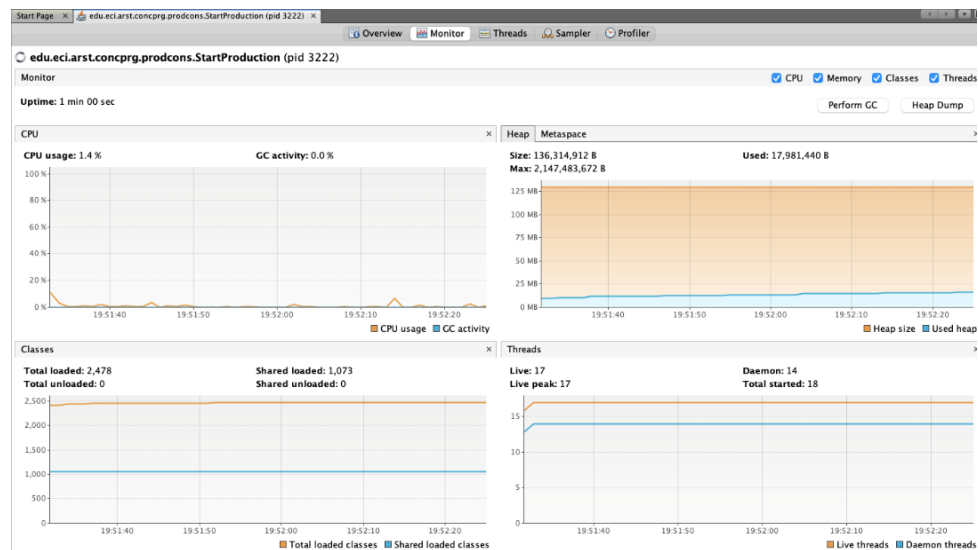
Teniendo en cuenta que el productor conoce un límite de Stock (cuantos elementos debería tener, a lo sumo en la cola), haga que dicho límite se respete. Revise el API de la colección usada como cola para ver cómo garantizar que dicho límite no se supere. Verifique que, al poner un límite pequeño para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



```
@Override
public void run() {
    while (true) {

        if(queue.size() < stockLimit){
            dataSeed = dataSeed + rand.nextInt( bound: 100);
            System.out.println("Producer added " + dataSeed);
            queue.add(dataSeed);
            System.out.println("tamaño de la cola " + queue.size());
        }

        try {
            Thread.sleep( millis: 500);
        } catch (InterruptedException ex) {
            Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, msg: null, ex);
        }
    }
}
```



Rta: se verifica que la cola no supere el limite y se baja el tiempo de sleep en producción a la mitad del de consumidor para que este sea más rápido que el productor y la CPU no se ve afectada por estos cambios.

Parte 3



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?

Rta: la vida se debe calcula según la cantidad de jugadores y la vida inicial de cada uno de ellos, por lo tanto, se debe aplicar una fórmula similar a la siguiente:

$$N * VidaTotal$$

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

Start	Pause and check	Resume	num. of immortals: 3
Fight: im0[130] vs im2[100] Fight: im1[70] vs im2[100] Fight: im0[130] vs im1[70] Fight: im2[100] vs im1[70] Fight: im2[90] vs im1[70] Fight: im1[70] vs im2[90] Fight: im0[140] vs im1[70]			
[im0[120], im1[90], im2[100]] Health sum:310			

Start	Pause and check	Resume	num. of immortals: 3
Fight: im0[180] vs im2[100] Fight: im2[150] vs im0[170] Fight: im1[40] vs im2[150] Fight: im0[180] vs im2[140] Fight: im2[150] vs im0[170] Fight: im1[50] vs im0[170] Fight: im0[180] vs im2[140]			
[im0[180], im1[30], im2[150]] Health sum:360			

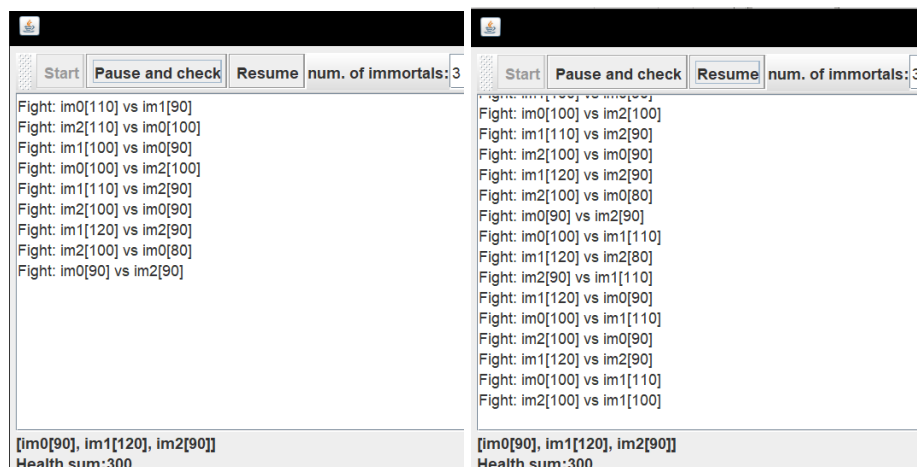
Rta: No se está cumpliendo el invariante la vida total está variando después de cada ataque.

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’.

```
public class ControlFrame extends JFrame {
    public ControlFrame() {
        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.setFlag(flag:true);
                }

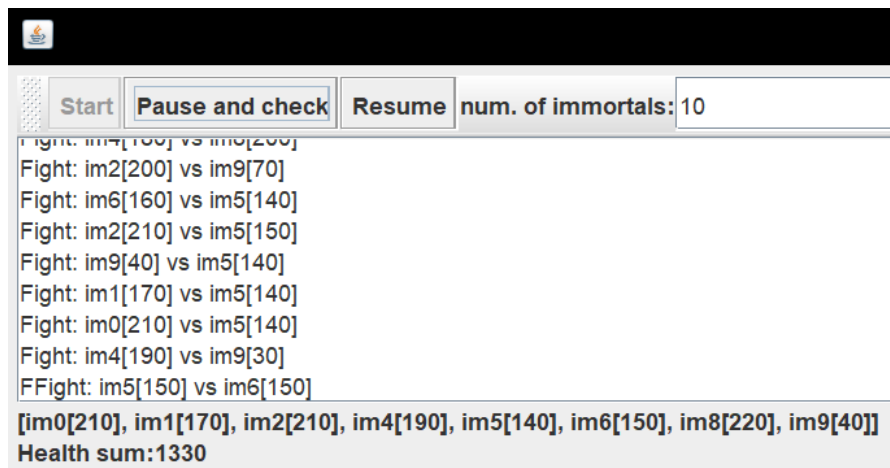
                for (Immortal im : immortals) {
                    sum += im.getHealth();
                }
                statisticsLabel.setText("<html>" + immortals.toString() + "<br>Health sum:" + sum);
            }
        });
        toolBar.add(btnPauseAndCheck);

        JButton btnResume = new JButton(text:"Resume");
        btnResume.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                synchronized(immortals) {
                    for (Immortal im : immortals) {
                        im.setFlag(flag:false);
                    }
                    immortals.notifyAll();
                }
            }
        });
    }
};
```



Rta: Se implementa el código para los botones de “Pause&Check” y “Resume” los cuales funcionan correctamente.

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



Rta: No se cumple el invariante “Health sum” debería ser 1000 y no debe cambiar con el paso del tiempo.

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:

```
public class Immortal extends Thread {
    public void fight(Immortal i2) {
        synchronized (immortalsPopulation) {
            if (i2.getHealth() > 0) {
                if (this.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");
                immortalsPopulation.remove(i2);
                i2.setStop(stop:true);
            }
        }
    }
}
```


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.

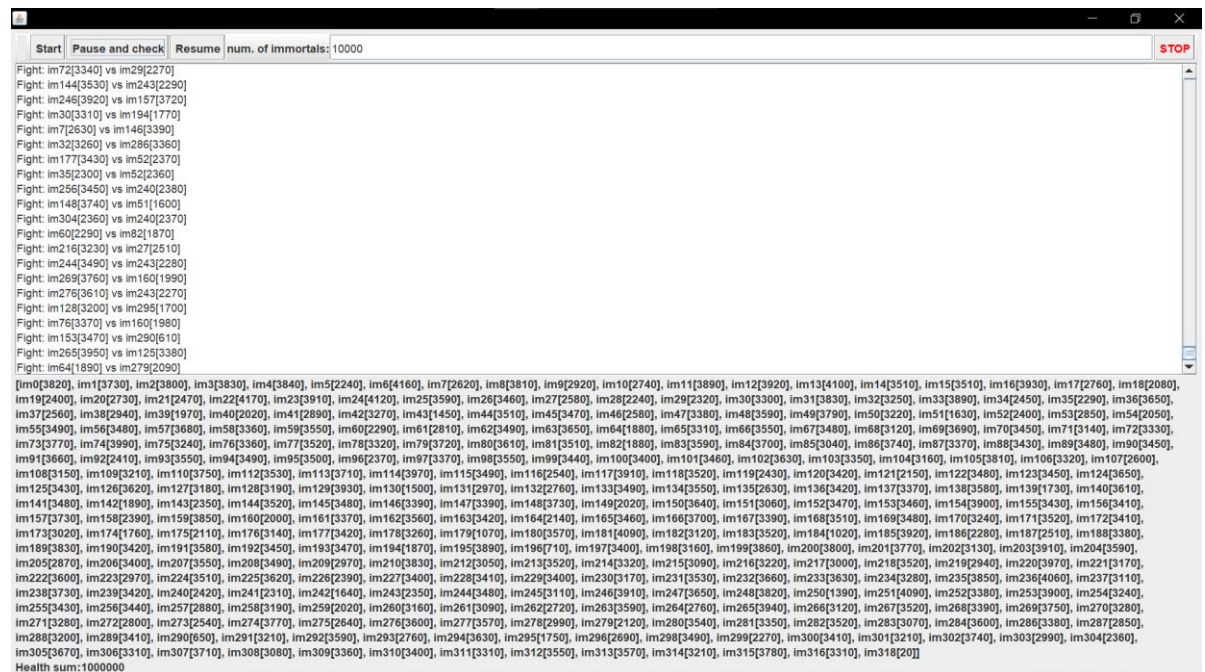
Rta: No se detuvo.

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

Rta: No se realizó, la estrategia utilizada fue correcta

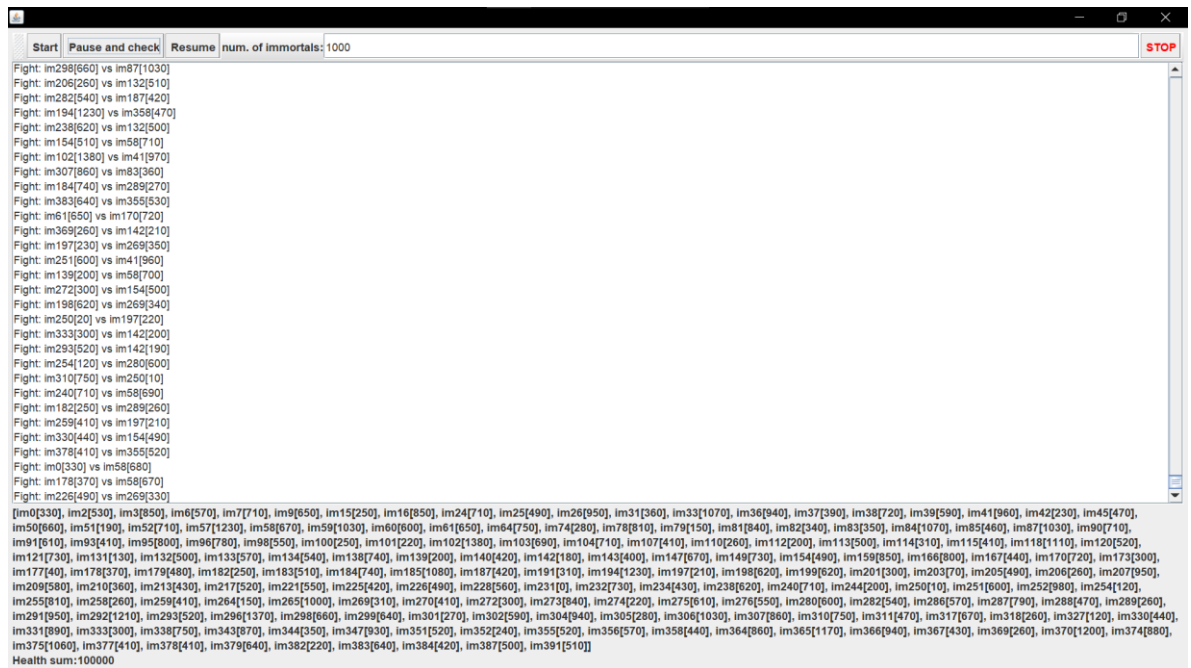
9. Una vez corregido el problema, rectifique que el programa siga funcionando de manera consistente cuando se ejecutan 100, 1000 o 10000 inmortales. Si en estos casos grandes se empieza a incumplir de nuevo el invariante, debe analizar lo realizado en el paso 4.

Para 10000

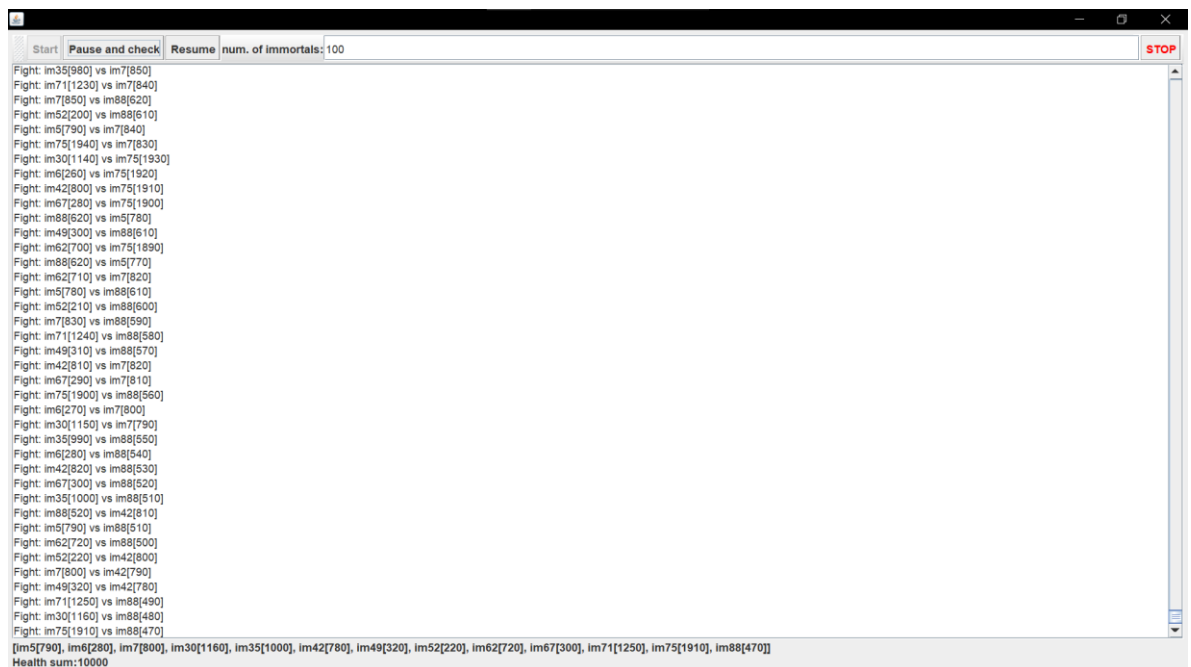


```
Start Pause and check Resume num. of Immortals: 10000 STOP
Fight: im72[3340] vs im29[2270]
Fight: im144[3530] vs im243[2290]
Fight: im246[3920] vs im157[3720]
Fight: im30[3310] vs im194[1770]
Fight: im7[2630] vs im146[3390]
Fight: im32[3260] vs im286[3360]
Fight: im177[3430] vs im52[2370]
Fight: im35[2300] vs im52[3260]
Fight: im256[3450] vs im240[2380]
Fight: im148[3740] vs im51[1600]
Fight: im304[2360] vs im240[2370]
Fight: im60[2290] vs im82[1870]
Fight: im216[3230] vs im27[2510]
Fight: im244[3490] vs im243[2280]
Fight: im269[3760] vs im160[1990]
Fight: im276[3610] vs im243[2270]
Fight: im128[3200] vs im295[1700]
Fight: im76[3370] vs im160[1980]
Fight: im153[3470] vs im290[610]
Fight: im265[3950] vs im125[3380]
Fight: im64[1890] vs im279[2090]
[im0[3820], im1[3730], im2[3800], im3[3830], im4[3840], im5[2240], im6[4160], im7[2620], im8[3810], im9[2920], im10[2740], im11[3890], im12[3920], im13[4100], im14[3510], im15[3510], im16[3930], im17[2760], im18[2080], im19[2400], im20[2730], im21[2470], im22[4170], im23[3910], im24[4120], im25[3590], im26[3460], im27[2580], im28[2240], im29[2320], im30[3300], im31[3830], im32[3250], im33[3890], im34[2450], im35[2290], im36[3650], im37[2560], im38[2940], im39[1970], im40[2020], im41[2890], im42[3270], im43[1450], im44[3510], im45[3470], im46[2580], im47[3380], im48[3590], im49[3790], im50[3220], im51[1630], im52[2400], im53[2850], im54[2050], im55[3490], im56[3480], im57[3680], im58[3360], im59[3550], im60[2290], im61[2810], im62[3490], im63[3650], im64[1880], im65[3310], im66[3550], im67[3480], im68[3120], im69[3690], im70[3450], im71[3140], im72[3330], im73[3770], im74[3990], im75[3240], im76[3360], im77[3520], im78[3320], im79[3720], im80[3610], im81[3510], im82[1880], im83[3590], im84[3700], im85[3040], im86[3740], im87[3370], im88[3430], im89[3480], im90[3450], im91[3660], im92[2410], im93[3550], im94[3490], im95[3500], im96[2370], im97[3370], im98[3550], im99[3440], im100[3400], im101[3460], im102[3630], im103[3350], im104[3160], im105[3810], im106[3320], im107[2600], im108[3150], im109[3210], im110[3750], im111[3530], im112[3530], im113[3710], im114[3970], im115[3490], im116[2540], im117[3910], im118[3520], im119[2430], im120[3420], im121[2150], im122[3480], im123[3450], im124[3650], im125[3430], im126[3620], im127[3180], im128[3190], im129[3930], im130[1500], im131[2970], im132[2760], im133[3490], im134[3550], im135[2630], im136[3420], im137[3370], im138[3580], im139[1730], im140[3610], im141[2480], im142[1890], im143[2350], im144[3520], im145[3480], im146[3390], im147[3390], im148[3730], im149[2020], im150[3640], im151[3060], im152[3470], im153[3460], im154[3900], im155[3430], im156[3410], im157[3730], im158[2390], im159[3850], im160[2000], im161[3370], im162[3560], im163[3420], im164[2140], im165[3460], im166[3700], im167[3390], im168[3510], im169[3480], im170[3240], im171[3520], im172[3410], im173[3020], im174[1760], im175[2110], im176[3140], im177[3420], im178[3260], im179[1070], im180[3570], im181[4090], im182[3120], im183[3520], im184[1020], im185[3920], im186[2280], im187[2510], im188[3380], im189[3830], im190[3420], im191[3580], im192[3450], im193[3470], im194[1870], im195[3890], im196[710], im197[3400], im198[3160], im199[3860], im200[3800], im201[3770], im202[3130], im203[3910], im204[3590], im205[2870], im206[3400], im207[3550], im208[3490], im209[2970], im210[3830], im212[3050], im213[3520], im214[3320], im215[3090], im216[3220], im217[3000], im218[3520], im219[2940], im220[3970], im221[3170], im222[3600], im223[2970], im224[3510], im225[3620], im226[2390], im227[3400], im228[3410], im229[3400], im230[3170], im231[3530], im232[3660], im233[3630], im234[3280], im235[3850], im236[4060], im237[3110], im238[3730], im239[3420], im240[2420], im241[2310], im242[1640], im243[2350], im244[3480], im245[3110], im246[3910], im247[3650], im248[3820], im250[1390], im251[4090], im252[3380], im253[3900], im254[3240], im255[3430], im256[3440], im257[2880], im258[3190], im259[2020], im260[3160], im261[3090], im262[2720], im263[3590], im264[2760], im265[3940], im266[3120], im267[3520], im268[3390], im269[3750], im270[3280], im271[3280], im272[2800], im273[2540], im274[3770], im275[2640], im276[3600], im277[3570], im278[2990], im279[2120], im280[3540], im281[3350], im282[3520], im283[3070], im284[3600], im286[3380], im287[2850], im288[3200], im289[3410], im290[650], im291[3210], im292[3590], im293[2760], im294[3630], im295[1750], im296[2690], im298[3490], im299[2270], im300[3410], im301[3210], im302[3740], im303[2990], im304[2360], im305[3670], im306[3310], im307[3710], im308[3080], im309[3360], im310[3400], im311[3310], im312[3550], im313[3570], im314[3210], im315[3780], im316[3310], im318[20]]
Health sum: 1000000
```

Para 1000



Para 100



Rta: Para cualquier caso funciona correctamente el invariante, los distintos botones, cuando se pausar se observan los que quedan vivos y sus respectivas vidas si se usa el resumen los hilos continúan sin tener problemas.

10. Un elemento molesto para la simulación es que en cierto punto de esta 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.

```
updateCallback.processReport(this + " says:" + i2 + " is already dead!\n");
immortalsPopulation.remove(i2);
i2.setStop(stop:true);
}
```

11. Para finalizar, implemente la opción STOP.

```
btnStop.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        synchronized (immortals){
            for(Immortal im : immortals){
                im.setStart(start:false);
            }
        }
        clearOutput();
        btnStart.setEnabled(b:true);
    }
});
```

Rta: Se implementa la opción de “stop”, esta detiene todos los hilos borra todos los mensajes del proceso y vuelve a activar el botón de “start” para ejecutar un nuevo caso.

Conclusiones

- La optimización del consumo de CPU y la sincronización de hilos son aspectos cruciales en la programación concurrente. En el ejercicio del productor/consumidor, se demostró la importancia de equilibrar los tiempos de producción y consumo para evitar cuellos de botella y maximizar la eficiencia del sistema.
- La identificación y solución de condiciones de carrera son fundamentales para garantizar la consistencia y la integridad de los datos en aplicaciones multihilo. En el caso del juego de simulación de combate entre inmortales, se abordaron las condiciones de carrera que surgieron durante la ejecución del programa, implementando estrategias de bloqueo para evitar problemas de concurrencia.
- Los invariantes son propiedades críticas que deben mantenerse durante la ejecución de un programa concurrente. En el juego de simulación, se destacó la importancia de mantener el invariante relacionado con la suma total de puntos de vida de los inmortales, asegurando que se mantenga constante incluso durante las operaciones de incremento/reducción de vida.
- La implementación de soluciones eficientes y escalables es esencial para garantizar el rendimiento óptimo de las aplicaciones concurrentes, especialmente cuando se enfrentan a un gran número de hilos o procesos. Las pruebas realizadas con diferentes cantidades de inmortales demostraron la capacidad del programa para mantener la consistencia y el rendimiento incluso en escenarios de alta concurrencia.

- La utilización adecuada de la palabra clave `synchronized` en Java puede ser efectiva para evitar deadlocks en programas concurrentes. Al aplicar la sincronización en secciones críticas del código, se puede garantizar que solo un hilo tenga acceso a esos recursos compartidos a la vez, evitando conflictos y potenciales bloqueos mutuos entre hilos.

Bibliografia

- *Peierls, T., Goetz, B., Bloch, J., Bowbeer, J., Lea, D., & Holmes, D. (2006). Java Concurrency in Practice.* <http://ci.nii.ac.jp/ncid/BA78043355>
- *ChatGPT.* (s. f.). <https://chat.openai.com/>