Informe del Laboratorio

Programación Concurrente

Jacobo Diaz Alvarado
Santiago Carmona Pineda

February 2026

1. Parte I — Productor/Consumidor

1.1. 1. Análisis de consumo de CPU

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

De acuerdo con el monitoreo, el consumo alto de CPU se debe a la implementación de **busy-wait** en la clase `BusySpinQueue`, donde los hilos productores y consumidores verifican constantemente el estado del buffer sin ceder el control, generando un uso intensivo del procesador.

```
public void put(T item) {
    // spin hasta que haya espacio
    while (true) {
        if (q.size() < capacity) {
            q.addLast(item);
            return;
        }
        Thread.onSpinWait();
    }
}

public T take() {
    // spin hasta que haya elementos
    while (true) {
        T v = q.pollFirst();
        if (v != null)
            return v;
        Thread.onSpinWait();
    }
}
```

Como se observa, ambos métodos utilizan un bucle infinito para verificar el estado del buffer, provocando alto consumo de CPU.

Consumo de CPU antes de optimización

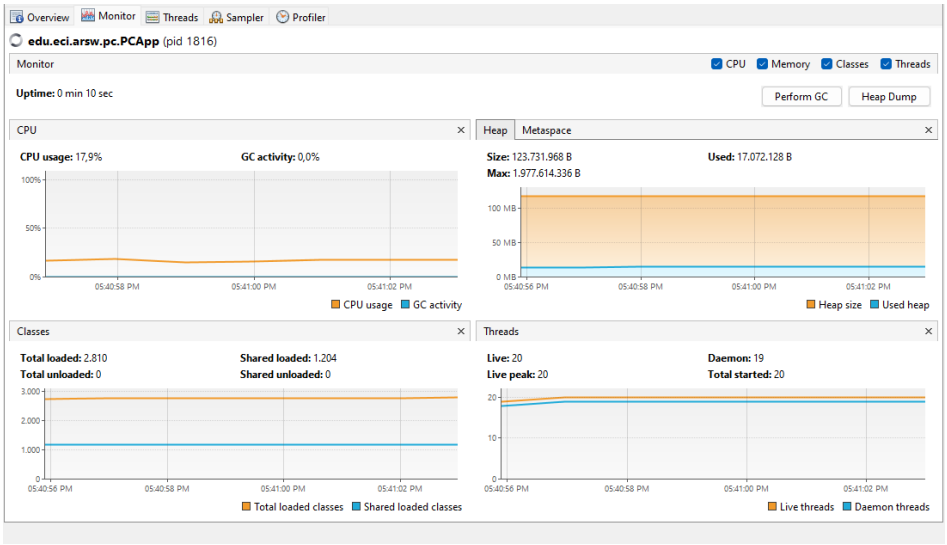

Figura 1: Consumo de CPU antes de optimización (1)

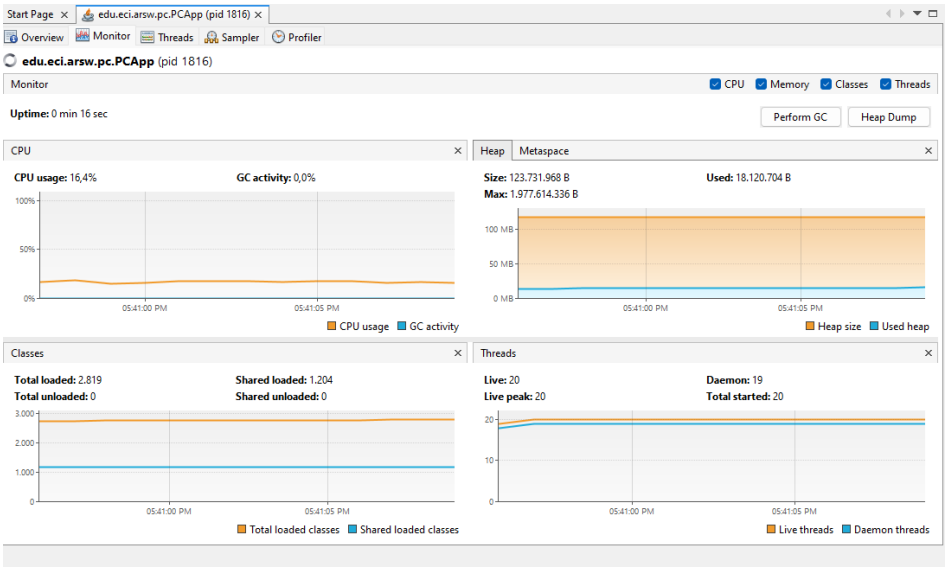

Figura 2: Consumo de CPU antes de optimización (2)

1.2. 2. Optimización con monitores

Ajusta la implementación para usar CPU eficientemente cuando el productor es lento y el consumidor es rápido.

Se implementa la clase `BoundedBuffer` utilizando monitores con `wait()` y `notifyAll()`, permitiendo que los hilos esperen sin consumir CPU.

```
public void put(T item) throws InterruptedException {
    synchronized (this) {
        while (q.size() == capacity) {
            this.wait();
        }
        q.addLast(item);
        this.notifyAll();
    }
}

public T take() throws InterruptedException {
    synchronized (this) {
        while (q.isEmpty()) {
            this.wait();
        }
        T v = q.removeFirst();
        this.notifyAll();
        return v;
    }
}
```

El consumo de CPU disminuye notablemente, ya que los hilos solo se activan cuando hay cambios en el buffer. Se ajustaron los delays (productor 30ms, consumidor 1ms) para evidenciar mejor el comportamiento.

Consumo de CPU después de optimización

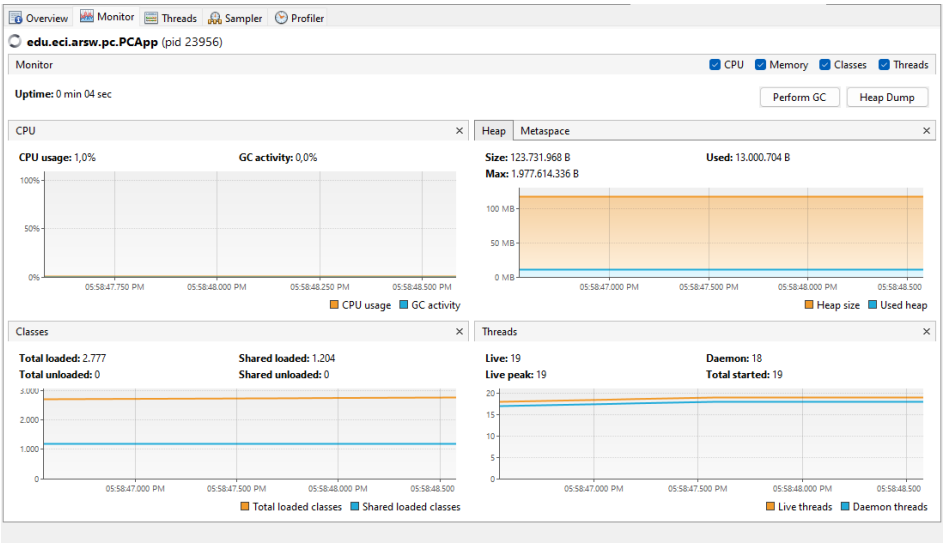

Figura 3: Consumo de CPU después de optimización (1)

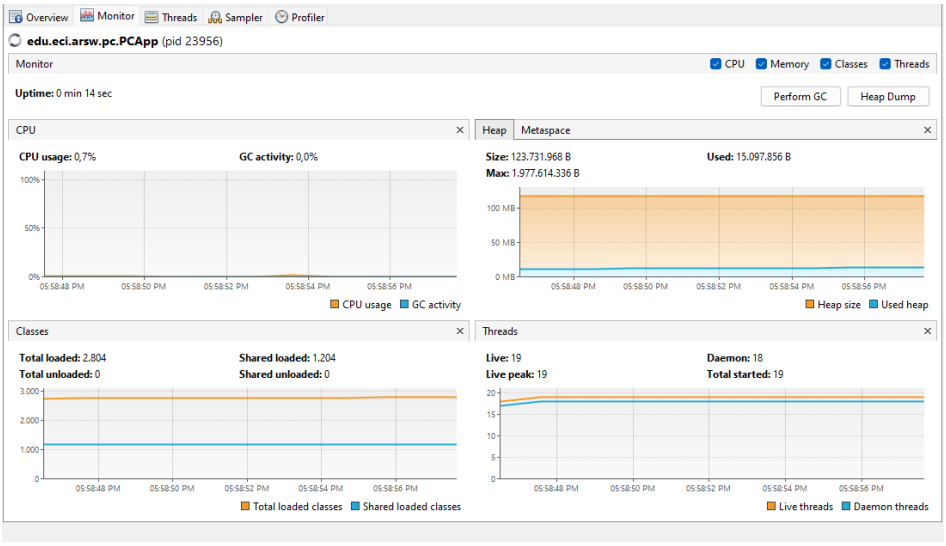

Figura 4: Consumo de CPU después de optimización (2)

1.3. 3. Productor rápido y consumidor lento

Se mantiene `BoundedBuffer` con monitores, configurando productor rápido (1ms), consumidor lento (50ms) y capacidad pequeña.

```
String mode = System.getProperty("mode", "monitor");
int producers = Integer.getInteger("producers", 1);
```

```
int consumers = Integer.getInteger("consumers", 1);
int capacity = Integer.getInteger("capacity", 8);
long prodDelay = Long.getLong("prodDelayMs", 1);
long consDelay = Long.getLong("consDelayMs", 50);
int duration = Integer.getInteger("durationSec", 3);
```

El productor respeta el límite sin espera activa y el consumo de CPU se mantiene bajo.

Evidencia con stock pequeño

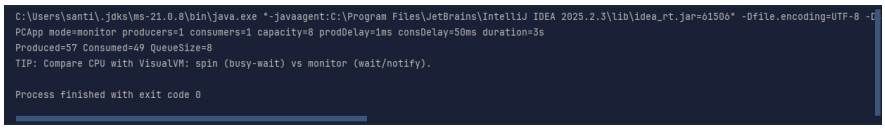

Figura 5: Elementos en el Buffer

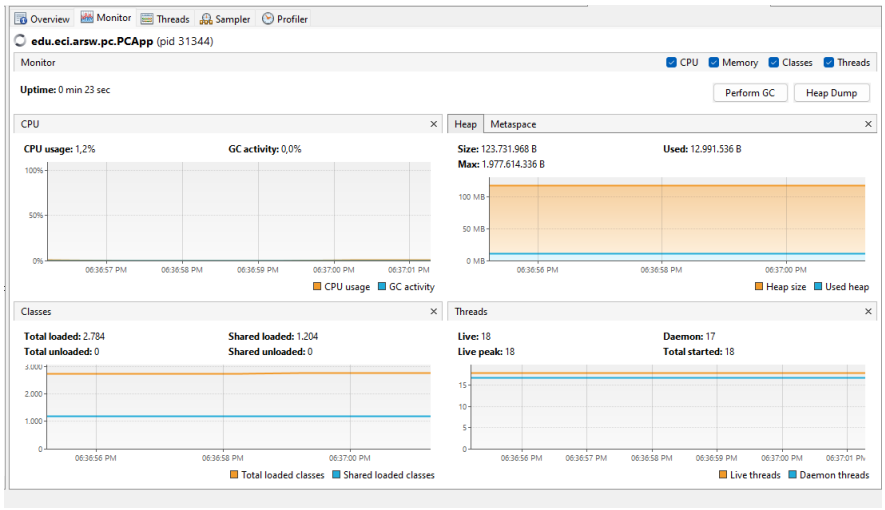

Figura 6: Bajo consumo de CPU

2. Parte II — Búsqueda distribuida y condición de parada

Se utilizó la clase *AtomicInteger*, que permite operaciones atómicas seguras en entornos concurrentes, evitando sincronización explícita y reduciendo condiciones de carrera.

A continuación, se presenta la solución planteada.

```
    public static AtomicInteger countApparences = new AtomicInteger
        ();
```

```
    @Override
    public void run() {
        for (int i = start; i < end && countApparences.get() <
           HostBlackListsValidator.BLACK_LIST_ALARM_COUNT; i++) {
            checkedListsCount++;
            if (skds.isInBlackListServer(i, ipAddress)) {
                int currentCount = countApparences.getAndIncrement()
                   ;
                if (currentCount < HostBlackListsValidator.
                   BLACK_LIST_ALARM_COUNT) {
                    blackListOcurrences.add(i);
                }
            }
        }
    }
```

Con esta nueva implementación, se garantiza que solo se recorran las listas necesarias gracias a la variable estática. Además, el uso de variables atómicas permite que no hayan condiciones de carrera y que solo un hilo pueda incrementar el valor del contador de manera simultanea

3. Parte III — Inmortales

3.1. Reglas de juego

1. Hay N inmortales.
2. Cada uno tiene una vida H.
3. Cada uno tiene un daño D.
4. Si un inmortal recibe daño D, la vida del agresor aumenta D/2.
5. Total de peleas T.

3.2. Detección del invariante

$$\textbf{Salud Total} = N \times H - T \times \left(\frac{D}{2}\right)$$

Ejemplo:
N = 8 H = 100 D = 10 T = 140
Salud total = 100

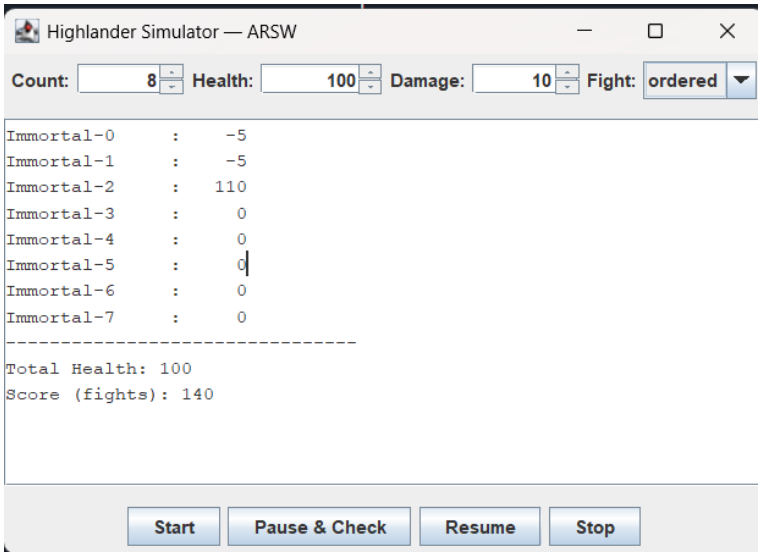

Figura 7: Ejemplo realizado con el simulador

3.3. Implementación del botón resume

Implementado en la clase `PauseController`:

```
public void resume() {
    lock.lock();
    try {
        paused = false;
        unpaused.signalAll();
    } finally {
        lock.unlock();
    }
}
```

3.4. Regiones críticas

Identificadas en la clase `Immortal`, método `run()`:

```
@Override
public void run() {
    try {
```

```
        while (running) {
            controller.awaitIfPaused();
            if (!running) break;
            var opponent = pickOpponent();
            if (opponent == null) continue;
            String mode = System.getProperty("fight", "ordered");
            if ("naive".equalsIgnoreCase(mode))
                fightNaive(opponent);
            else fightOrdered(opponent);
            Thread.sleep(2);
        }
    } catch (InterruptedException ie) {
        Thread.currentThread().interrupt();
    }
}
```

Puntos críticos:

- Uso de `controller` - Métodos `fightNaive()` y `fightOrdered()` - Modificación de `scoreBoard`

Solución: Uso de `synchronized`

```
public synchronized int getHealth() {
    return health;
}

private void fightNaive(Immortal other) {
    synchronized (this) {
        synchronized (other) {
            if (this.health <= 0 || other.health <= 0) return;
            other.health -= this.damage;
            this.health += this.damage / 2;
            scoreBoard.recordFight();
        }
    }
}

private void fightOrdered(Immortal other) {
    Immortal first = this.name.compareTo(other.name) < 0 ? this :
       other;
```

```
    Immortal second = this.name.compareTo(other.name) < 0 ? other :
        this;
    synchronized (first) {
        synchronized (second) {
            if (this.health <= 0 || other.health <= 0) return;
            other.health -= this.damage;
            this.health += this.damage / 2;
            scoreBoard.recordFight();
        }
    }
}
```

3.5. Pruebas del invariante

Para este punto del laboratorio, se realizaron pruebas para N=100, 1000 y 10000 inmortales, esto con el objetivo de validar el comportamiento del invariante en casos más extremos.

Al realizar el ejercicio, observamos que el invariante fue válido para todas las ejecuciones.

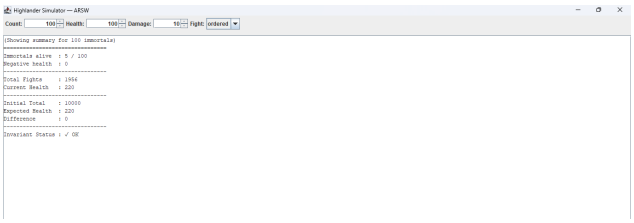

Figura 8: Prueba para N = 100

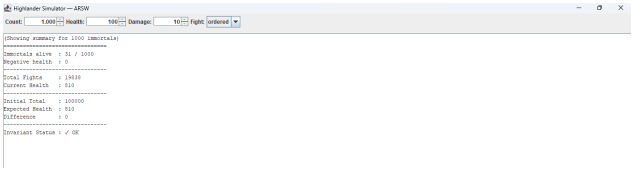

Figura 9: Prueba para N = 1000

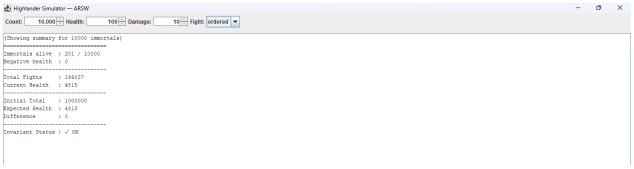

Figura 10: Prueba para N = 10000

3.6. Eliminación de invariantes muertos

El código original usaba *ArrayList*, lo que causaba *ConcurrentModificationException* e I*ndexOutOfBoundsException* cuando se intentaba remover inmortales mientras otros hilos accedían a la lista.

Solución en *ImmortalManager.java:*

Se reemplazó ArrayList por CopyOnWriteArrayList, una colección thread-safe ideal para escenarios con muchas lecturas y pocas escrituras:

```
private final List<Immortal> population = new CopyOnWriteArrayList
    <>();

public int removeDeadImmortals() {
  int removed = 0;
  List<Immortal> toRemove = new ArrayList<>();
  for (Immortal im : population) {
    if (!im.isAlive()) {
      toRemove.add(im);
    }
  }
  for (Immortal im : toRemove) {
    if (population.remove(im)) {
      removed++;
    }
  }
  return removed;
}
```

Mejora en *Immortal.java:*

Se mejoró pickOpponent() para manejar el caso donde la lista cambia durante la selección:

```
private Immortal pickOpponent() {
  int size = population.size();
  if (size <= 1) return null;

  int attempts = 0;
  int maxAttempts = Math.min(10, size * 2);

  while (attempts < maxAttempts) {
    try {
```

```
      int index = ThreadLocalRandom.current().nextInt(size);
      Immortal other = population.get(index);
      if (other != this && other.isAlive()) {
        return other;
      }
    } catch (IndexOutOfBoundsException e) {
      size = population.size();
      if (size <= 1) return null;
    }
    attempts++;
  }

  for (Immortal im : population) {
    if (im != this && im.isAlive()) {
      return im;
    }
  }
  return null;
}
```