Reporte de Laboratorio
Diseño de Sincronización en PrimeFinder - - PARTE 1

Objetivo del Diseño
Implementar un sistema multihilo para buscar números primos con capacidad de pausar todos los hilos periódicamente, mostrar estadísticas y reanudar con sincronización adecuada, evitando condiciones de carrera y garantizando coherencia.

Elementos de Sincronización Implementados

1. **Lock (Monitor)**
   Objeto monitor único: Se crea un objeto `Object monitor = new Object()` en la clase "Contro".`
   Este mismo objeto se pasa a todos los hilos `PrimeFinderThread` mediante el constructor.
   Todas las secciones críticas se sincronizan sobre este mismo objeto:

   ```
   synchronized(monitor) {
       // código crítico
   }
   ```

   Ventaja: Garantiza que todos los hilos (incluyendo el controlador) comparten el mismo lock, evitando deadlocks por locks diferentes.

2. **Condiciones de Espera:**
   Se utilizan tres variables de condición principales:

   shouldPause(volatile boolean)
   - Indica cuando el sistema debe entrar en estado de pausa
   - Controlada por el hilo `Control` cada `t` milisegundos

   threadsPaused (int)
   - Contador de hilos que han confirmado su pausa
   - Incrementado por cada hilo cuando entra en estado de espera

   activeThreads(int)
   - Contador de hilos que aún están ejecutándose
   - Decrementado cuando un hilo termina su rango de búsqueda

Mecanismo de Sincronización

Flujo de Pausa

1. Control.wait(TMILISECONDS) → Espera timeout o notificación
2. Control: shouldPause = true
3. Control: monitor.notifyAll() → Despierta todos los hilos
4. Hilos: Detectan shouldPause == true
5. Hilos: threadsPaused++ y monitor.wait()
6. Control: Espera mientras threadsPaused < activeThreads
7. Control: Muestra estadísticas y espera ENTER
8. Control: shouldPause = false
9. Control: monitor.notifyAll() → Reanuda todos los hilos

Implementación en Código

En Control (hilo principal):

```
// Pausa programada
synchronized(monitor) {
    shouldPause = true;
    threadsPaused = 0;
    monitor.notifyAll();

    // Espera a que todos los hilos se pausen
    while (threadsPaused < activeThreads) {
        monitor.wait();
    }
}
```

En PrimeFinderThread (hilos trabajadores):

```
synchronized(monitor) {
    while (control.shouldPause() && !paused) {
        paused = true;
        control.threadPaused(); // Incrementa threadsPaused

        // Espera hasta que shouldPause sea false
        while (control.shouldPause()) {
```

```
                monitor.wait();
            }
            paused = false;
        }
}
```

Prevención de Lost Wakeups

Los "lost wakeups" ocurren cuando un hilo llama a “notify()”/”notifyAll()” antes de que otro hilo llame a “wait()”. El hilo que debería ser despertado entra en “wait()”después de la notificación, lo que resulta en que el hilo se queda esperando indefinidamente

Soluciones Implementadas

1. **Uso de “notifyAll()” en lugar de “notify()”**

   ```
   monitor.notifyAll(); // En lugar de monitor.notify()
   ```

   Esto garantiza que todos los hilos en espera sean notificados y evita que un hilo específico se pierda la notificación

2. **Patrón de verificación en bucle while**

   ```
   // PATRÓN CORRECTO (evita lost wakeups y spurious wakeups)
   while (condition) {
       monitor.wait();
   }

   // PATRÓN INCORRECTO (vulnerable a lost wakeups)
   if (condition) {
       monitor.wait();
   }
   ```

3. **Prevención de Spurious Wakeups**

   Se da un Spurious Wake up si un hilo despierta de un “wait()” sin haber sido llamado por “notify()”. Para solucionar esto debemos lograra que el mismo patrón de bucle “while” proteja contra spurious wakeups, con esto si un hilo despierta espuriamente, vuelve a verificar la condición

Evitando Busy-Waiting

El busy watimg se cuando un hilo verifica repetidamente una condición en un bucle, consumiendo CPU innecesariamente:

```
// EJEMPLO DE BUSY-WAITING (NO RECOMENDADO)
while (!condition) {
    // Consume CPU constantemente
}
```

Solución Implementada:
Uso exclusivo de `wait()` para esperas:

```
// CORRECTO: No hay busy-waiting
synchronized(monitor) {
    while (!condition) {
        monitor.wait();  // Libera CPU mientras espera
    }
}`
```

Diseño de Variables de Estado
Variables "volatile"

```
private volatile boolean shouldPause = false;
private volatile boolean finished = false;
```

Decidimos usar este tipo de variables para garantizar la visibilidad entre hilos, sin esto un hilo podría no ver que cambios ha realizado otro y por lo tanto generar un problema de integridad de los datos

Contadores Protegidos

```
private int threadsPaused = 0;
private int activeThreads = 0;
```

Estos contadores solo pueden ser accedidos en bloques "syncronized"dado que las operaciones como ++ no son atómicas.

Decisiones de Diseño

Para abordar estos desafíos, se tomaron varias decisiones arquitectónicas fundamentales. Primero, se implementó un monitor único compartido por todos los hilos, simplificando la sincronización y eliminando posibles deadlocks. Segundo, se adoptó el patrón de verificación en bucle while para todas las condiciones de espera, protegiendo tanto contra lost wakeups como contra spurious wakeups (despertares espurios del sistema). Tercero, se prefirió el uso de

notifyAll() sobre notify() para garantizar que todas las notificaciones llegarán a su destino en un sistema con múltiples condiciones de espera. Cuarto, se utilizaron variables volatile para los flags de control entre hilos, asegurando visibilidad inmediata de los cambios. Quinto, todos los contadores como threadsPaused y activeThreads se accedieron exclusivamente dentro de bloques sincronizados, ya que operaciones como incremento y decremento no son atómicas en Java.

Ventajas

El diseño resultante presenta varias ventajas significativas. En primer lugar, elimina completamente el busy-waiting, utilizando solo mecanismos de espera pasiva que liberan la CPU cuando los hilos no tienen trabajo que hacer. En segundo lugar, evitar deadlocks mediante un flujo de sincronización claro y predecible donde todos los hilos comparten el mismo monitor. En tercer lugar, garantiza notificaciones confiables sin lost wakeups gracias al patrón de verificación en bucle. Además, el sistema es escalable, funcionando correctamente con cualquier número de hilos de trabajo sin necesidad de modificar la lógica de sincronización. Finalmente, es robusto en el manejo de casos límite, como cuando los hilos terminan su ejecución durante períodos de pausa.

Conclusión

El diseño implementado demuestra un uso correcto y efectivo de los mecanismos de sincronización provistos por Java. Mediante la implementación de un solo lock compartido para toda la comunicación entre hilos, se estableció una base sólida para la coordinación. Las variables de condición claramente definidas (shouldPause, threadsPaused) permitieron expresar de manera intuitiva los estados del sistema. La protección completa contra lost wakeups se logró mediante la aplicación consistente del patrón estándar de bucle while alrededor de todas las llamadas a wait(). La ausencia total de busy-waiting se consiguió utilizando exclusivamente los mecanismos wait()/notify() para la coordinación entre hilos. Por último, el manejo robusto de casos límite, como la finalización de hilos durante pausas, aseguró que el sistema fuera resiliente a condiciones inesperadas. En conjunto, este diseño garantiza que el sistema sea eficiente en el uso de recursos, correcto en su funcionamiento y libre de condiciones de carrera, cumpliendo íntegramente con todos los requisitos de sincronización solicitados y sirviendo como ejemplo de buenas prácticas en programación concurrente en Java.

PARTE 2 - - SnakeRace Concurrent

1. **Análisis de concurrencia**
   Explica cómo el código usa hilos para dar autonomía a cada serpiente.

En este laboratorio, la concurrencia se utiliza para darle autonomía a cada serpiente, permitiendo que todas se muevan y actúen al mismo tiempo sin depender unas de otras. Esto se logra gracias al uso de hilos, donde cada serpiente se ejecuta de manera independiente.

La autonomía se construye a partir de dos clases principales: SnakeRunner y SnakeApp, cada una con un rol claro dentro del sistema.

Por un lado, SnakeRunner, que implementa la interfaz Runnable, define qué hace cada serpiente de forma autónoma. En esta clase se encuentra la lógica de comportamiento individual: cada serpiente se mueve continuamente, puede girar de forma aleatoria, reacciona ante colisiones y activa el modo turbo cuando es necesario. Todo esto ocurre dentro de un bucle while que se ejecuta en el método run(), el cual es ejecutado por un hilo propio. De esta manera, cada serpiente toma sus decisiones sin necesidad de coordinarse con las demás.

Por otro lado, SnakeApp es la clase encargada de crear y lanzar los hilos, es decir, es donde realmente se otorga la autonomía a cada serpiente. Primero, se crean múltiples instancias de serpientes (líneas 28–34), donde el número de serpientes se define dinámicamente y cada una se inicializa con su propia posición y dirección.

Posteriormente, en las líneas 49–50, se crea un ejecutor con Executors.newVirtualThreadPerTaskExecutor(), el cual permite usar hilos virtuales (disponibles en Java 21). A continuación, cada serpiente es enviada al ejecutor mediante exec.submit(new SnakeRunner(s, board)), lo que provoca que cada serpiente se ejecute en su propio hilo virtual.

Gracias a este enfoque, cada hilo ejecuta de forma independiente el método run() de su respectivo SnakeRunner. No existe sincronización entre serpientes, por lo que cada una mantiene su propio ciclo de vida y actúa de manera totalmente independiente. Esto permite que todas se ejecuten de forma concurrente, es decir, al mismo tiempo.

En conclusión, la autonomía de las serpientes se logra gracias a la combinación de ambas clases: SnakeRunner, que define el comportamiento autónomo de cada serpiente, y SnakeApp, que se encarga de crear y asignar un hilo virtual a cada una. Mencionar ambas es fundamental para explicar correctamente cómo se implementa la concurrencia en este laboratorio.

2. Data Races Encontradas y su Solución

Ubicación	Data Race potencial	Solución implementada
Snake.direction	Múltiples hilos (interfaz gráfica con entradas WASD y SnakeRunner) leen y escriben la dirección de la serpiente de forma simultánea.	Declaración de la variable como volatile, garantizando visibilidad inmediata de los cambios entre hilos.
Snake.body (ArrayDeque)	El hilo de la interfaz gráfica accede al cuerpo de la serpiente para dibujarla mientras el hilo SnakeRunner lo modifica mediante el método advance().	Métodos marcados como synchronized (head(), snapshot(), advance()), asegurando exclusión mutua durante el acceso.
Board.mice, obstacles, turbo, teleports	Varias serpientes acceden concurrentemente a las colecciones durante la ejecución de step(), mientras la interfaz gráfica las lee para renderizar el tablero.	Uso de bloques synchronized(this) en step() y métodos sincronizados que retornan copias defensivas de las colecciones.
GameState	Diferentes hilos (UI, SnakeRunner y GameClock) consultan y modifican el estado del juego de manera concurrente.	Implementación de AtomicReference<GameState> con operaciones atómicas como compareAndSet() para garantizar consistencia.

Codigo implementado

2. Colecciones Mal Usadas y Cómo se Protegieron

Colección	Problema original	Protección aplicada
Set<Position> mice	Modificación concurrente (remove() / add() en step()) mientras la interfaz gráfica lee la colección para dibujar los elementos.	Uso de bloques synchronized y retorno de una copia defensiva mediante new HashSet<>(mice).
Set<Position> obstacles	Acceso concurrente similar al de mice, con modificaciones durante la ejecución del juego y lecturas simultáneas desde la UI.	Sincronización explícita y retorno de una copia defensiva de la colección.
Set<Position> turbo	Modificación concurrente por la lógica del juego mientras la UI accede a los datos para renderizar.	Protección mediante synchronized y copias defensivas para evitar interferencias entre hilos.
Map<Position, Position> teleports	Lecturas concurrentes mientras otras partes del sistema pueden modificar la estructura.	Métodos sincronizados y retorno de una copia defensiva con new HashMap<>(teleports).
Deque<Position> body (Snake)	Operaciones como addFirst() y removeLast() pueden colisionar con lecturas realizadas por la interfaz gráfica.	Todos los accesos encapsulados dentro de métodos synchronized, garantizando exclusión mutua.

Getters con copias defensivas:

```
public synchronized Set<Position> mice() { return new
HashSet<>(mice); }
public synchronized Set<Position> obstacles() { return new
HashSet<>(obstacles); }
public synchronized Set<Position> turbo() { return new
HashSet<>(turbo); }
public synchronized Map<Position, Position> teleports() { return
new HashMap<>(teleports); }
```

3. Esperas Activas Eliminadas y Mecanismo Utilizado

Ubicación	Espera activa original	Mecanismo de reemplazo
SnakeRunner.run() (pausa)	Uso de *busy-wait* mediante while (gameState == PAUSED), provocando consumo innecesario de CPU.	Reemplazo por CyclicBarrier.await() junto con Thread.sleep(50), implementando un *polling* ligero y controlado.
Control de velocidad de la serpiente	Bucle de ejecución continuo sin pausas entre movimientos, generando uso excesivo del procesador.	Inserción de Thread.sleep(baseSleepMs) entre iteraciones (80 ms en modo normal y 40 ms en modo turbo).
GameClock	Consulta manual y repetitiva del estado del juego para controlar el avance del tiempo.	Uso de ScheduledExecutorService.scheduleAtFixedRate() para generar *ticks* automáticos y periódicos.

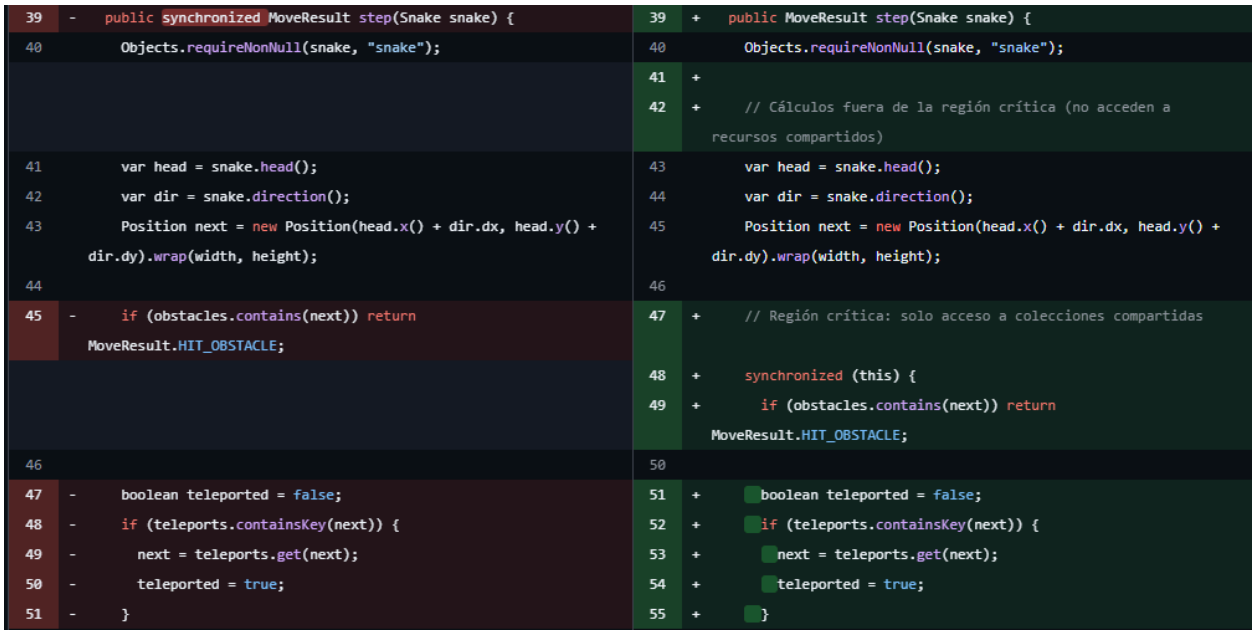
```
} else if (gameState.get() == GameState.PAUSED) {
    pauseBarrier.await();  // Sincronización de todas las
serpientes
    while (gameState.get() == GameState.PAUSED) {
        Thread.sleep(50);  // Polling ligero en lugar de
busy-wait
    }
}
```

CyclicBarrier justificación: Se utiliza para garantizar que todas las serpientes lleguen al punto de pausa antes de que cualquiera continúe, evitando estados inconsistentes visuales.

4. Regiones Críticas Definidas y Justificación de Alcance Mínimo

Clase	Región crítica	Alcance	Justificación
Board.step()	synchronized(this) { ... }	Únicamente la verificación de colisiones y la modificación de las colecciones compartidas.	Los cálculos de posición (head, dir, next.wrap()) se realizan fuera del bloque sincronizado para reducir la contención entre hilos.
Snake.head()	Método completo sincronizado	Lectura de un solo elemento	Se protege el acceso mínimo necesario para retornar de forma segura la cabeza de la serpiente.
Snake.snapshot()	Método completo sincronizado	Creación de una copia del cuerpo	La copia defensiva permite que la interfaz gráfica trabaje con los datos sin necesidad de mantener el bloqueo.
Snake.advance()	Método completo sincronizado	Modificación del cuerpo	Agrupa las operaciones addFirst() y removeLast() de manera atómica, evitando estados intermedios inconsistentes.
***Getters* de Board**	Método completo sincronizado	Creación de copias defensivas	El uso de copias defensivas libera al hilo llamador inmediatamente y reduce el tiempo de bloqueo.

Board.step() - Región crítica minimizada:



```
- public synchronized MoveResult step(Snake snake) {
    Objects.requireNonNull(snake, "snake");

    var head = snake.head();
    var dir = snake.direction();
    Position next = new Position(head.x() + dir.dx, head.y() + dir.dy).wrap(width, height);

-   if (obstacles.contains(next)) return MoveResult.HIT_OBSTACLE;

    boolean teleported = false;
-   if (teleports.containsKey(next)) {
-     next = teleports.get(next);
-     teleported = true;
-   }
```

```
+ public MoveResult step(Snake snake) {
    Objects.requireNonNull(snake, "snake");
+
+   // Cálculos fuera de la región crítica (no acceden a recursos compartidos)
    var head = snake.head();
    var dir = snake.direction();
    Position next = new Position(head.x() + dir.dx, head.y() + dir.dy).wrap(width, height);

+   // Región crítica: solo acceso a colecciones compartidas
+   synchronized (this) {
+     if (obstacles.contains(next)) return MoveResult.HIT_OBSTACLE;

    boolean teleported = false;
+   if (teleports.containsKey(next)) {
+     next = teleports.get(next);
+     teleported = true;
+   }
```

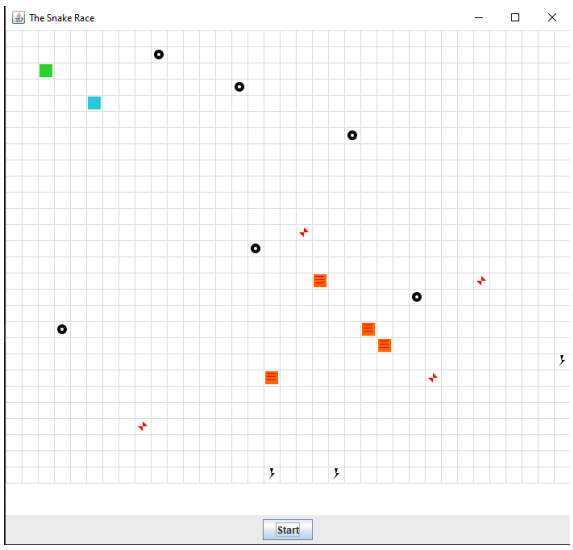
Resumen de Mecanismos de Sincronización Utilizados:

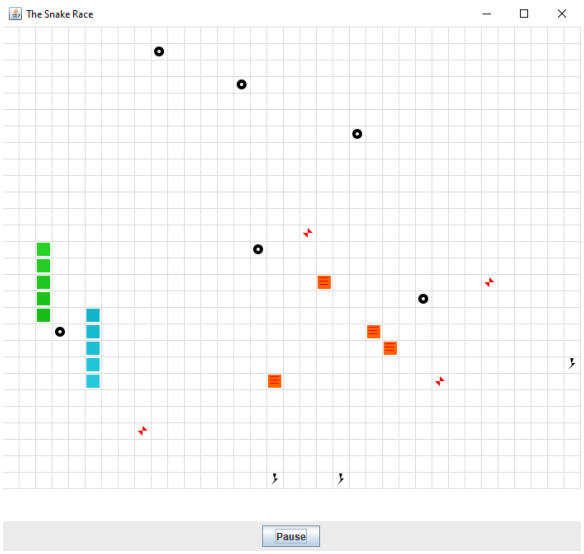
Mecanismo	Uso en el proyecto
volatile	Utilizado en Snake.direction para garantizar la visibilidad inmediata de los cambios de dirección entre hilos.
synchronized	Aplicado en métodos de las clases Snake y Board para asegurar acceso exclusivo a recursos compartidos y evitar condiciones de carrera.
AtomicReference<GameState>	Permite manejar el estado del juego de forma segura entre múltiples hilos mediante operaciones atómicas.
CyclicBarrier	Usado para sincronizar todas las serpientes cuando el juego entra en estado de pausa, asegurando que todas esperen hasta reanudarse.
Thread.sleep()	Implementado para controlar la velocidad de movimiento de las serpientes y para realizar *polling* ligero durante pausas.
Copias defensivas	Se retornan nuevas instancias de colecciones para evitar modificaciones externas y reducir conflictos entre hilos.

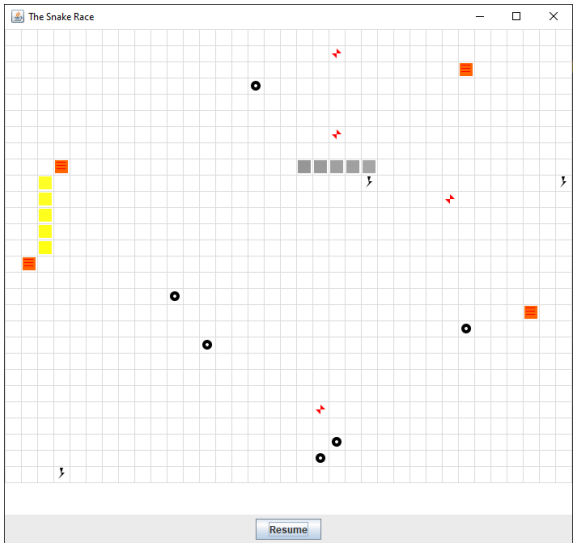
Hilos virtuales (Java 21)	Uso de Executors.newVirtualThreadPerTaskExecutor() para permitir el manejo eficiente de múltiples serpientes concurrentes con menor consumo de recursos.

3.UI con Iniciar / Pausar / Reanudar y estadísticas solicitadas al pausar.

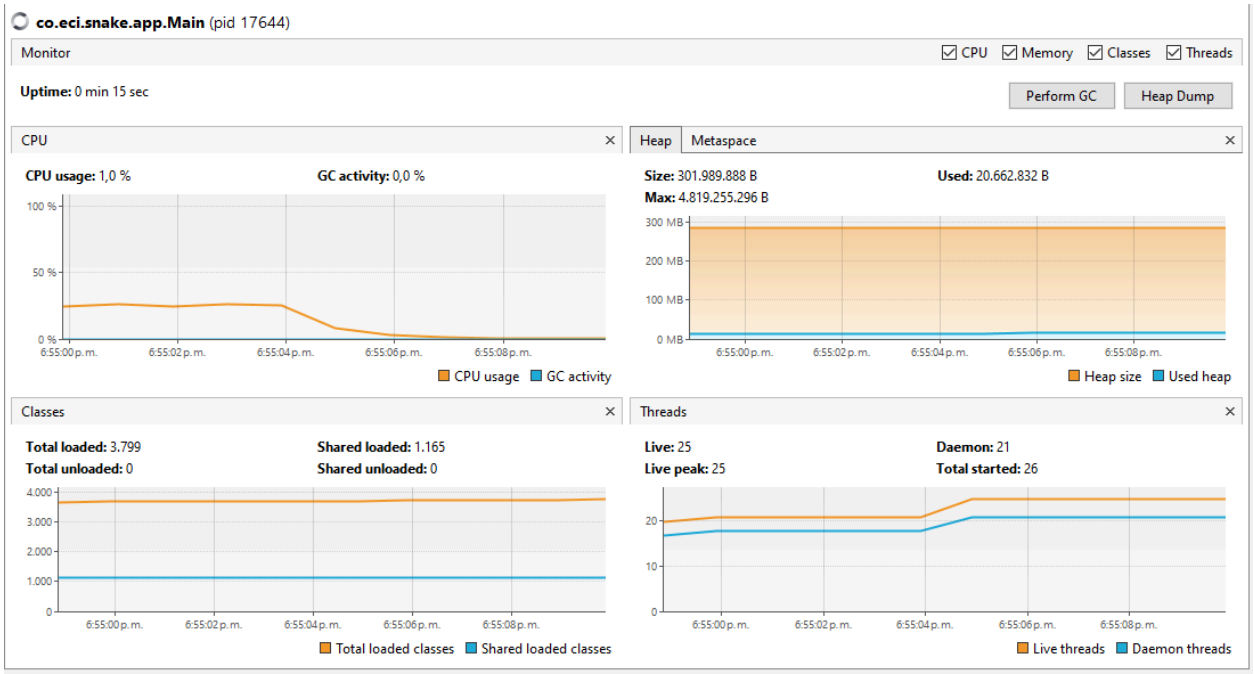
Para que el usuario tenga una mejor experiencia de usuarios se optó porque los botones cambiarán al momento de que se les de click. No se eligió colocar 3 botones con 3 funcionalidades distintas porque deja la posibilidad al jugador para que los clickee y estos no hagan nada.



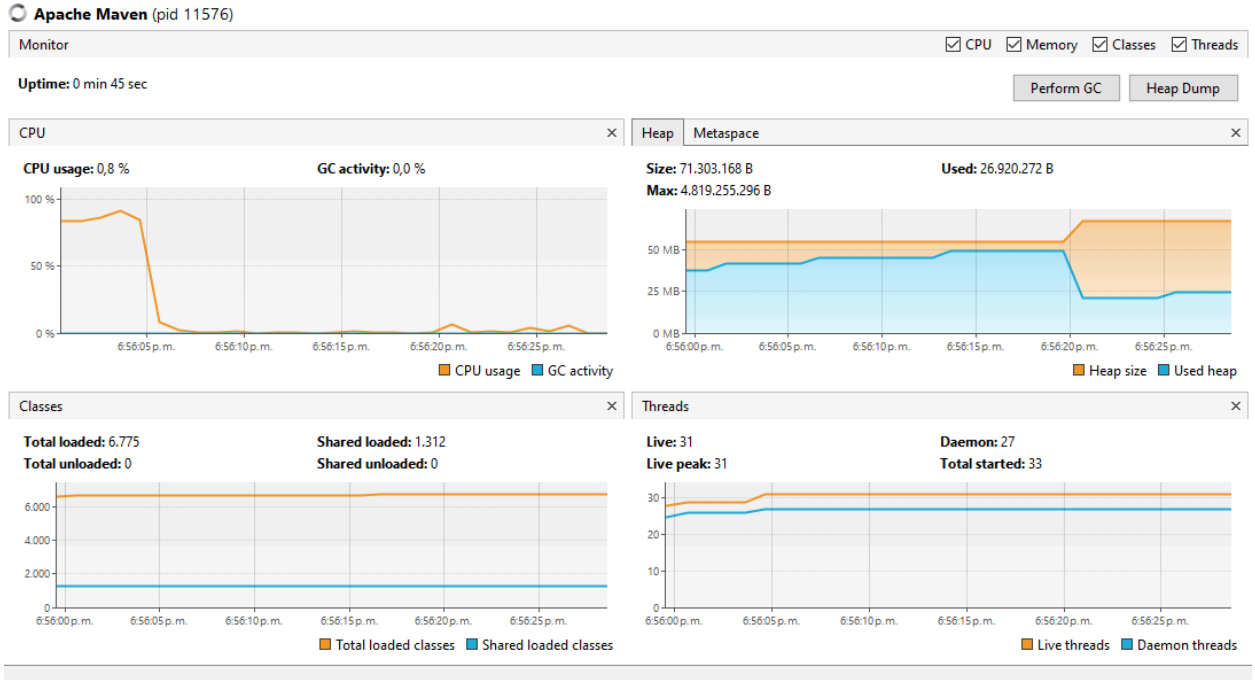




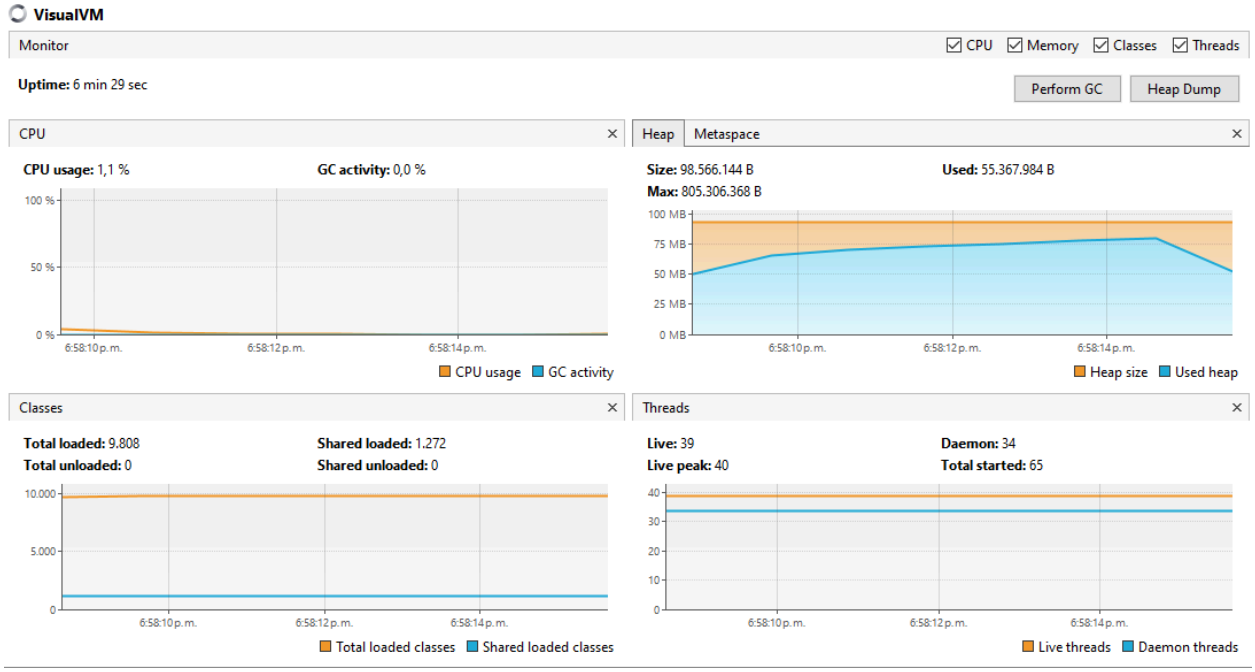
Con 2 serpientes activas:



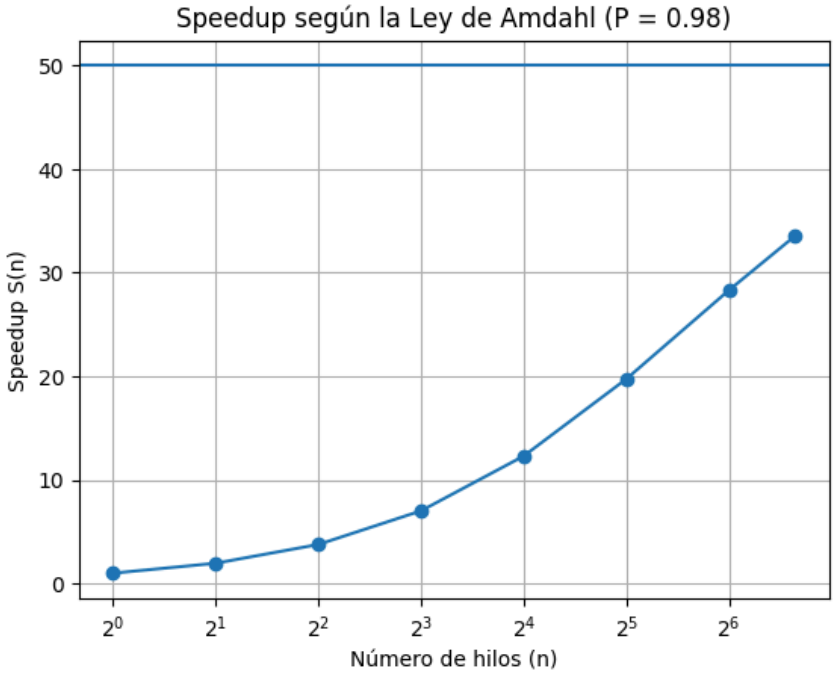
Con 20 serpientes activas:



Con 60 serpientes activas:



Analisis Ley de Amdahl:



Análisis: Con una fracción paralelizable del 98%, el proyecto Snake Race presenta un speedup teórico máximo de 50x. Sin embargo, la eficiencia decrece rápidamente: con 8 serpientes se alcanza el 88% de eficiencia, pero con 32 serpientes baja al 62%. Esto demuestra que las regiones críticas sincronizadas (Board.step() y Snake.advance()) junto con el CyclicBarrier constituyen el 2% secuencial que limita la escalabilidad según la Ley de Amdahl.