

INFORME DE LABORATORIO 03

Concurrencia y Sincronización

Asignatura: Arquitecturas de Software (ARSW)

Institución: Escuela Colombiana de Ingeniería Julio Garavito

1. Parte I: Productor / Consumidor

1.1 Diagnóstico de Consumo de CPU

Al ejecutar el programa original con la implementación BusySpinQueue, se observó un alto consumo de CPU (cercano al 100% en el núcleo asignado).

Causa Identificada

La clase BusySpinQueue utilizaba una estrategia de **Espera Activa (Busy Wait)**:

```
while (condition) {  
    Thread.onSpinWait();  
}
```

Esto impide que el hilo entre en estado WAITING o BLOCKED, por lo que el planificador del sistema operativo le sigue asignando tiempo de CPU innecesariamente.

1.2 Optimización Implementada

Se migró la implementación a BoundedBuffer utilizando el mecanismo de monitores de Java (synchronized, wait(), notifyAll()).

Cambios Clave

- **Sincronización:**
Se restringió el acceso a la cola crítica usando synchronized(this).
- **Espera Eficiente:**
Se reemplazó el bucle de espera activa por wait().
Esto libera el lock y suspende el hilo, reduciendo el consumo de CPU a valores mínimos hasta que sea notificado.
- **Notificación:**
Se utiliza notifyAll() para despertar a los hilos (productores o consumidores) cuando el estado de la cola cambia.

Evidencia de Mejora

(Incluir aquí captura de pantalla de VisualVM comparando Busy Wait vs Wait/Notify. Ver README.)

2. Parte III: Highlander Simulator (Inmortales)

2.1 Invariante y Consistencia

Invariante Teórico

La suma de la salud de todos los inmortales ($N \times \text{SaludInicial}$) debe permanecer constante durante toda la simulación.

Problemas y Correcciones

1. Lógica de Combate

El código original restaba M al atacado pero sumaba M/2 al atacante, causando una pérdida neta de energía.

Corrección:

Se ajustó para sumar la misma cantidad que se resta:

```
this.health += damage;
```

Pérdida por Muerte

Cuando un inmortal moría y era eliminado de la lista, su salud desaparecía de la suma total.

Corrección:

Se implementó la variable deadHealthSum en ImmortalManager para acumular la salud de los inmortales caídos.

La validación ahora es:

```
SaludTotal = Suma(Vivos) + deadHealthSum
```

Evidencia de Validación

Al usar “**Pause & Check**” con N=100 y N=1000, el sistema reporta consistentemente:

Invariant OK: true

2.2 Control de Pausa y Dead Threads**Problema**

La simulación se bloqueaba (timeout) al intentar pausar si ya habían muerto inmortales.

Causa

El PauseController esperaba que todos los N hilos originales reportaran su pausa.

Los hilos muertos ya habían finalizado su método run() y no podían reportarse.

Solución

Se añadió un bloque finally en el método run() de Immortal para notificar su salida:

```
finally {
    controller.removeThread(); // Descuenta este hilo del total esperado
}
```

Esto permite que la pausa funcione correctamente con una población decreciente.

2.3 Regiones Críticas y Deadlocks

Se identificó que el método de pelea accede a dos recursos compartidos (this y other).

Estrategia Anti-Deadlock Adoptada: Orden Total

Para evitar el bloqueo mutuo (Deadlock), donde A espera a B y B espera a A, se impuso un orden de adquisición de locks basado en el ID único del inmortal:

```
Immortal first = id1 < id2 ? im1 : im2;
```

```
Immortal second = id1 < id2 ? im2 : im1;
```

```
synchronized(first) {
    synchronized(second) {
        // Lógica de combate crítica
    }
}
```

Esto rompe la condición de **espera circular**, necesaria para que ocurra un deadlock.

Estrategia Alternativa: TryLock

Se implementó también una variante (modo trylock) usando ReentrantLock.

Si un hilo no puede adquirir el lock del oponente inmediatamente, libera su propio lock y reintenta más tarde, evitando quedarse bloqueado indefinidamente.

Evidencia de Deadlock (Modo Naive)

Si se captura un deadlock con jstack, puede observarse una salida como la siguiente:
Found one Java-level deadlock:

"Immortal-1":

waiting to lock monitor 0x000... (object 0x000... "Immortal-2")
which is held by "Immortal-2"

"Immortal-2":

waiting to lock monitor 0x000... (object 0x000... "Immortal-1")
which is held by "Immortal-1"

2.4 Eliminación de Muertos (Sin Bloqueo Global)

Para remover inmortales muertos sin detener la simulación ni usar sincronización pesada:

1. **Cola Concurrente:**
Los inmortales muertos se añaden a una ConcurrentLinkedQueue (thread-safe, non-blocking).
2. **Hilo de Limpieza:**
Un hilo dedicado (Cleanup-Thread) procesa esta cola periódicamente.
3. **Lista Segura:**
La población principal usa CopyOnWriteArrayList, permitiendo que el hilo de limpieza elimine elementos mientras otros hilos iteran sobre ella (para pintar la UI o calcular totales) sin lanzar ConcurrentModificationException.

3. Conclusiones

La práctica demostró la importancia de:

1. Evitar la espera activa para un uso racional de los recursos.
2. Gestionar correctamente la sincronización de múltiples recursos para evitar deadlocks.
3. Mantener la consistencia de los datos (invariantes) incluso en escenarios de alta concurrencia y fallos (muerte de hilos).