MEDAC
Instituto Oficial de Formación Profesional

El Problema de la Cena de los Filósofos

Gabriel Sánchez Heredia

Índice

1. Introducción

1.1. Descripción del problema

El problema de la cena de los filósofos es un problema clásico de concurrencia propuesto por Edsger Dijkstra en 1965. Representa los desafíos de sincronización en sistemas donde múltiples procesos compiten por recursos compartidos limitados.

En este escenario, cinco filósofos están sentados alrededor de una mesa circular. Cada filósofo alterna entre dos actividades: pensar y comer. Para comer, cada filósofo necesita dos palillos: el de su izquierda y el de su derecha. Sin embargo, sólo hay cinco palillos en total, uno entre cada par de filósofos adyacentes.

1.2. Objetivo de la implementación

El objetivo de esta implementación es resolver el problema utilizando semáforos en Java para:

- Garantizar la exclusión mútua (dos filósofos no pueden usar el mismo palillo simultáneamente).
- Evitar el interbloqueo (deadlock).
- Prevenir la inanición (starvation) de cualquier filósofo.

2. Análisis del Problema

2.1. Componentes del sistema

- **Filósofos (5 hilos):** Cada filósofo es un hilo independiente que ejecuta un ciclo de pensar-comer.
- **Palillos (5 recursos):** Recursos compartidos que deben ser adquiridos de forma exclusiva.
- **Mesa circular:** Disposición que crea dependencias cíclicas entre los filósofos.

2.2. Desafíos de concurrencia

- **Interbloqueo (Deadlock):** Puede ocurrir si todos los filósofos toman su palillo izquierdo simultáneamente. Cada uno esperará indefinidamente por el palillo derecho, que está siendo sostenido por su vecino.
- **Inanición (Starvation):** Un filósofo podría nunca conseguir ambos palillos si sus vecinos comen continuamente.
- **Condiciones de carrera:** Acceso simultáneo a los mismos recursos sin sincronización adecuada.

3. Diseño de la Solución

3.1. Diagrama de clases

La solución ha sido refactorizada a una Arquitectura de Tres Capas que mejora la Separación de Responsabilidades, introduciendo la clase Mesa como el gestor de recursos central.

- **CenaFilosofos:** Actúa como el orquestador de nivel superior, solo inicializa la clase Mesa y ejecuta la simulación.
- **Mesa:** Gestiona el array de Semaphore (palillos), implementa la lógica anti-deadlock, y se encarga de la orquestación (creación, inicio y espera de los hilos Filosofo).
- **Filosofo:** Es el hilo de trabajo. Delega todas las acciones de adquisición y liberación de recursos al objeto Mesa.

3.2. Uso de Semáforos

Los semáforos binarios (inicializados con 1 permiso) se utilizan para representar cada palillo:

- **semaphore.acquire():** Intenta tomar el palillo. Si está ocupado, el hilo espera.
- **semaphore.release():** Libera el palillo, permitiendo que otro filósofo lo tome.

Cada semáforo garantiza que sólo un filósofo puede usar un palillo a la vez (exclusión mutua).

4. Implementación

4.1. Clase filósofo

4.1.1. Método run()

Filosofo.java

```
public void run() {
    try {
        while (vecesComido < MAX_COMIDAS) {
            pensar();       // Simula actividad de pensamiento
            comer();        // Intenta coger los palillos y comer
            vecesComido++;
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        System.err.println("Filosofo " + id + " fue interrumpido.");
    }
}
```

Este ciclo representa el comportamiento natural del filósofo: alternar entre pensar y comer.

4.1.2. Método pensar()

Filosofo.java

```
private void pensar() throws InterruptedException {  1 usage
    System.out.println("Filosofo " + id + " esta pensando");
    Thread.sleep((long) (Math.random() * 1000));
}
```

Simula un tiempo aleatorio de pensamiento (0-1 segundo).

4.1.3. Método comer() - Sincronización con Semáforos

Filosofo.java

```
private void comer() throws InterruptedException {  1 usage  ga
    System.out.println("Filosofo " + id + " esta hambriento");

    // Llama al metodo de la Mesa para coger los palillos
    mesa.cogerPalillos(id, palilloIzqIndex, palilloDerIndex);

    // Si llega aquí, tiene ambos palillos
    System.out.println("Filosofo " + id + " esta comiendo");

    // Simula el tiempo que tarda en comer
    Thread.sleep((long) (Math.random() * 1500));

    // Llama al metodo de la Mesa para liberar los palillos
    soltarPalillos();
}
```

Clave de la solución: Los filósofos pares e impares cogen los palillos en orden diferente

4.1.4. Método soltarPalillos()

Filosofo.java

```
private void soltarPalillos() {  1 usage  gabiisnchez
    mesa.soltarPalillos(palilloIzqIndex, palilloDerIndex);

    // Mensaje con formato específico solicitado
    System.out.println("Filosofo " + id + " ha terminado de comer, palillos libres: "
            + idPalilloIzqVisual + ", " + idPalilloDerVisual);
}
```

Libera ambos palillos para que otros filósofos puedan usarlos.

4.2. Clase Mesa

4.2.1. Preparación de la mesa

Mesa.java

```java
public Mesa(int numFilosofos) {  13 usages  gabiisnchez
    this.numFilosofos = numFilosofos;
    this.palillos = new Semaphore[numFilosofos];

    // Inicializa cada palillo como un semáforo binario (permiso = 1)
    for (int i = 0; i < numFilosofos; i++) {
        this.palillos[i] = new Semaphore( permits: 1);
    }
}
```

4.2.2. Ejecución de la Simulación

Mesa.java

```java
public void ejecutarSimulacion() {  1 usage  gabiisnchez
    Thread[] filosofos = new Thread[numFilosofos];

    System.out.println("\nCena de los filosofos iniciada.\n");

    // Creación e inicio de los hilos
    for (int i = 0; i < numFilosofos; i++) {

        // Obtenemos los índices de los palillos.
        int[] indices = getIndicesPalillos(i);
        int palilloIzqIndex = indices[0];
        int palilloDerIndex = indices[1];

        // Creamos el objeto Runnable (Filosofo)
        Filosofo filosofo = new Filosofo(i,  mesa: this, palilloIzqIndex, palilloDerIndex);

        // Creamos el Hilo y le asignamos el Runnable
        filosofos[i] = new Thread(filosofo,  name: "Filosofo-" + i);

        filosofos[i].start();
    }

    // Esperar a que todos los filósofos terminen (join)
    try {
        for (Thread filosofo : filosofos) {
            filosofo.join();
        }
    } catch (InterruptedException e) {
        System.err.println("Error al esperar a los filosofos: " + e.getMessage());
    }

    System.out.println("\nCena de los filosofos finalizada.");
}
```

4.3. Clase CenaFilósofos (Main)

CenaFilosofos.java

```java
public class CenaFilosofos {  gabiisnchez

    private static final int NUM_FILOSOFOS = 5;  1 usage

    public static void main(String[] args) {  gabiisnchez

        // 1. Instancia la clase Mesa con el número de filósofos.
        Mesa mesa = new Mesa(NUM_FILOSOFOS);

        // 2. Ejecuta toda la simulación con una sola llamada.
        mesa.ejecutarSimulacion();
    }
}
```

5. Prevención de Interbloqueo e Inanición

5.1. Prevención de Interbloqueo

Estrategia implementada

Orden asimétrico de adquisición de recursos

- **Filósofos pares (2 y 4):** Toman primero el palillo izquierdo, luego el derecho.
- **Filósofos impares (1, 3, 5):** Toman primero el palillo derecho, luego el izquierdo.

¿Por qué funciona?

Esta estrategia rompe una de las cuatro condiciones necesarias para el interbloqueo (espera circular). Al menos un filósofo siempre podrá adquirir ambos palillos, evitando que todos queden esperando indefinidamente.

Ejemplo:

- Si los filósofos 2 y 4 toman sus palillos izquierdos.
- Los filósofos 1, 3, 5 intentarán tomar primero el derecho.
- Como algunos palillos derechos están libres, el ciclo se rompe.

5.2. Prevención de Inanición

Mecanismos implementados:

1. **Fairness del semáforo:** Java garantiza que los hilos esperando por un semáforo eventualmente lo adquirirán
2. **Tiempo aleatorio de pensar/comer:** Evita patrones de sincronización que favorezcan siempre a los mismos filósofos
3. **Límite de comidas:** Cada filósofo come un número finito de veces, garantizando que todos tengan oportunidad

6. Resultados de la Ejecución

6.1. Capturas de pantalla

Inicio de la simulación

```
Cena de los filosofos iniciada.

Filosofo 1 esta pensando
Filosofo 2 esta pensando
Filosofo 5 esta pensando
Filosofo 4 esta pensando
Filosofo 3 esta pensando
```

Filósofos pensando y comiendo

```
Filosofo 4 esta comiendo
Filosofo 4 ha terminado de comer, palillos libres: 4, 3
Filosofo 4 esta pensando
Filosofo 3 esta hambriento
Filosofo 3 esta comiendo
Filosofo 2 esta hambriento
Filosofo 5 esta hambriento
Filosofo 4 esta hambriento
Filosofo 1 ha terminado de comer, palillos libres: 1, 5
Filosofo 1 esta pensando
Filosofo 5 esta comiendo
Filosofo 5 ha terminado de comer, palillos libres: 5, 4
Filosofo 5 esta pensando
Filosofo 5 esta hambriento
Filosofo 5 esta comiendo
```

Finalización exitosa

```
Cena de los filosofos finalizada.
```

6.2. Análisis de la salida

La salida del programa demuestra:

1. **Exclusión mutua correcta**: Nunca vemos dos filósofos tomando el mismo palillo simultáneamente.
2. **Ausencia de interbloqueo:** El programa continúa ejecutándose sin bloquearse, todos los filósofos eventualmente comen.
3. **Ausencia de inanición:** Todos los filósofos completan sus MAX_COMIDAS comidas.
4. **Progreso del sistema:** Los mensajes muestran transiciones fluidas entre estados (pensar → intentar comer → comer → soltar palillos).

Ejemplo de secuencia exitosa:

```
Filosofo 4 esta hambriento
Filosofo 4 esta comiendo
Filosofo 4 ha terminado de comer, palillos libres: 4, 3
```

7. Conclusiones

7.1. Lecciones aprendidas

1. **Importancia de la sincronización:** En sistemas concurrentes, la sincronización adecuada es crucial para evitar condiciones de carrera y garantizar la consistencia

2. **Semáforos como herramienta:** Los semáforos son una primitiva poderosa para controlar el acceso a recursos compartidos

3. **Diseño cuidadoso previene problemas:** Una estrategia bien pensada (orden asimétrico) puede prevenir elegantemente el interbloqueo sin necesidad de mecanismos complejos

4. **Testing de concurrencia:** Los problemas de concurrencia pueden no manifestarse siempre, requiriendo múltiples ejecuciones para verificar la corrección

7.2. Posibles mejoras

1. **Límite de tiempo:** Añadir timeouts en las adquisiciones de semáforos para detectar posibles problemas
2. **Métricas de rendimiento:** Registrar estadísticas como tiempo promedio esperando, tiempo comiendo, distribución de comidas
3. **Simulación configurable:** Permitir ajustar el número de filósofos, tiempos de pensar/comer y número de comidas mediante parámetros
4. I**nterfaz gráfica:** Crear una visualización que muestre el estado de cada filósofo y palillo en tiempo real