FILÓSOFOS Y REINAS

Raúl Martín Torrabadella Mendoza y Rafael Auxilia Esquina

ÍNDICE

PROBLEMA DE LA CENA DE LOS FILÓSOFOS

Imagina a **cinco filósofos** sentados alrededor de una mesa, donde cada uno tiene un plato de comida y entre cada par de filósofos hay un tenedor. Cada filósofo puede hacer una de dos cosas:

- Pensar
- Comer (pero necesita dos tenedores para comer: uno a su izquierda y uno a su derecha)

El *Problema de la cena de los filósofos* surge cuando se deben coordinar las acciones de los filósofos de tal manera que:

- Ningún filósofo se quede esperando un tenedor para siempre.
- Los filósofos no llegan a un estado en el que todos estén esperando los tenedores (lo que se conoce como "deadlock").
- A la vez, debe evitarse que un filósofo coma indefinidamente mientras los demás esperan.

Este problema es útil para ilustrar conceptos sobre la competencia de recursos en sistemas computacionales, como la sincronización de hilos y el control de acceso a recursos compartidos.

En resumen, los desafíos planteados por el problema incluyen:

1. **Deadlock**: Si todos los filósofos toman un tenedor al mismo tiempo, pueden quedarse esperando que otro les pase el segundo, lo que genera un ciclo de espera infinita.
2. **Starvation**: Algunos filósofos pueden no tener la oportunidad de comer si otros siempre están tomando los tenedores primero.
3. **Concurrente acceso a recursos**: Hay que asegurarse de que los filósofos puedan acceder a los tenedores de manera justa sin causar conflictos.

Este problema se utiliza a menudo en teoría de sistemas operativos para enseñar sobre la importancia de la gestión de recursos y cómo evitar errores como los mencionados (deadlock y starvation) cuando varios procesos compiten por los mismos recursos.

Solución

Nosotros hemos decidido optar por una solución basada en hilos, usando las clases *Synchronized*, *wait*, *sleep* entre otras, las clases y métodos principales son:

Clase Mesa: Controla el acceso a los tenedores con un array de booleanos y usa *wait* para evitar que un filósofo tome un tenedor si está ocupado. Cuando un filósofo suelta los tenedores, notifyAll() despierta a los otros hilos.

```java
public static class Mesa {
    private boolean[] tenedores;

    public Mesa(int numTenedores) { this.tenedores = new boolean[numTenedores]; }

    public int tenedorIzquierda(int i) { return i; }

    public int tenedorDerecha(int i) {
        if (i == 0) {
            return this.tenedores.length - 1;
        } else {
            return i - 1;
        }
    }

    public synchronized void cogerTenedores(int comensal) {
        while (tenedores[tenedorIzquierda(comensal)] || tenedores[tenedorDerecha(comensal)]) {
            try {
                wait();
            } catch (InterruptedException ex) {
                Logger.getLogger(Mesa.class.getName()).log(Level.SEVERE, msg: null, ex);
            }
        }

        tenedores[tenedorIzquierda(comensal)] = true;
        tenedores[tenedorDerecha(comensal)] = true;
    }

    public synchronized void dejarTenedores(int comensal) {
        tenedores[tenedorIzquierda(comensal)] = false;
        tenedores[tenedorDerecha(comensal)] = false;
        notifyAll();
    }
}
```

Clase Filósofo: Representa a cada filósofo como un hilo. Siguen un ciclo de pensar, intentar coger tenedores, comer y liberarlos. Se usa un contador para asegurarse de que todos coman al menos una vez antes de finalizar.

```java
public static class Filosofo extends Thread {
    private Mesa mesa;               // Instancia de la clase Mesa
    private int comensal;            // Número del filósofo (1, 2, 3, ...)
    private int indiceComensal;      // Índice del filósofo en el array de tenedores
    private static int contadorComiendo = 0; // Contador de filósofos comiendo
    private static final Object lock = new Object(); // Objeto para sincronización
    private static boolean todosHanComido = false; // Estado de finalización

    public Filosofo(Mesa m, int comensal) {
        this.mesa = m;               // Asignar la mesa
        this.comensal = comensal;    // Asignar el número del filósofo
        this.indiceComensal = comensal - 1; // Ajustar el índice para el array
    }
```

Ejecución: El método ejecutar(int numFilosofo) crea la mesa y los filósofos, inicia sus hilos y espera su finalización con join(). Al ejecutarse se ve así, te recomendamos que veas el código detenidamente.

```java
// Método para ejecutar el programa
public static void ejecutar(int numFilosofo) {
    Mesa mesa = new Mesa(numFilosofo); // Crear la mesa con el número de filósofos
    Filosofo[] filosofos = new Filosofo[numFilosofo];

    for (int i = 1; i <= numFilosofo; i++) {
        filosofos[i - 1] = new Filosofo(mesa, i); // Crear filósofos
        filosofos[i - 1].start(); // Iniciar el hilo del filósofo
    }

    // Esperar a que todos los filósofos terminen
    for (Filosofo filosofo : filosofos) {
        try {
            filosofo.join(); // Esperar a que cada filósofo termine
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    System.out.println("El programa ha finalizado.");
}
```

```
Seleccione una opción:
1. Problema de los Filósofos
2. Problema de las N-Reinas
0. Salir
Opción: 1
Ingrese el número de filósofos: 5
Filósofo 2 está pensando.
Filósofo 4 está pensando.
Filósofo 5 está pensando.
Filósofo 3 está pensando.
Filósofo 1 está pensando.
Filósofo 5 está comiendo.
Filósofo 2 está comiendo.
Filósofo 5 deja de comer, tenedores libres: 5, 4
Filósofo 2 deja de comer, tenedores libres: 2, 1
Filósofo 3 está comiendo.
Filósofo 1 está comiendo.
Filósofo 3 deja de comer, tenedores libres: 3, 2
Filósofo 4 está comiendo.
Filósofo 4 deja de comer, tenedores libres: 4, 3
Filósofo 1 deja de comer, tenedores libres: 1, 5
Todos los filósofos han terminado de comer.
El programa ha finalizado.
```

PROBLEMA DE LAS N-REINAS

Imagina un tablero de ajedrez de **N x N** casillas. El objetivo es colocar **N reinas** en dicho tablero de tal manera que cada reina ocupe una fila y cumpla las siguientes condiciones:

- No haya dos reinas en la misma columna.
- No haya dos reinas en la misma diagonal.

Los desafíos planteados por el problema incluyen:

- **Explorar todas las posibles configuraciones**: Es necesario probar diferentes disposiciones de las reinas hasta encontrar una solución válida. La cantidad de posibles combinaciones aumenta exponencialmente con N.
- **Backtracking**: Al probar diferentes configuraciones, algunas serán inválidas (por ejemplo, si dos reinas están en la misma columna o diagonal). El algoritmo de **retroceso** se utiliza para deshacer las colocaciones incorrectas y seguir buscando nuevas soluciones.
- **Complejidad computacional**: La solución puede requerir una gran cantidad de tiempo de cómputo para valores grandes de N, ya que la complejidad crece rápidamente.

Este problema se utiliza a menudo en teoría de algoritmos para enseñar sobre la búsqueda de soluciones, la optimización y la complejidad computacional en problemas de programación.

Solución

Nuestra solución al problema de las N-Reinas implementa una búsqueda recursiva con backtracking para encontrar soluciones en un tablero de n × n:

Estructura: Usa un array tablero[] para almacenar la posición de cada reina en su respectiva fila.

```
private int[] tablero;
```

Verificación de seguridad: El método *esSeguro(int fila, int col)* comprueba si una reina puede colocarse sin ser atacada por otra en la misma columna o diagonal.

```
private boolean esSeguro(int fila, int col) {
    for (int i = 0; i < fila; i++) {
        if (tablero[i] == col ||
                tablero[i] - i == col - fila ||
                tablero[i] + i == col + fila) {
            return false;
        }
    }
    return true;
}
```

Backtracking: *resolverNReinas(int fila)* coloca reinas fila por fila, retrocediendo (backtracking) cuando una posición no es válida. Si una solución es válida, se imprime y se incrementa el contador de soluciones.

```
private void resolverNReinas(int fila) {
    if (fila == n) {
        imprimirSolucion();
        contadorSoluciones++; // Incrementar contador de soluciones
        if (contadorSoluciones >= 2) { // Si se encontraron 2 soluciones, salir
            System.exit( status: 0);
        }
        return;
    }
```

Al ejecutar se ve así dándote 2 soluciones:

```
Ingrese el número de reinas: 4
Solución posible:
. . Q .  4
Q . . .  3
. . . Q  2
. Q . .  1
  A B C D
```

```
Solución posible:
. Q . .  4
. . . Q  3
Q . . .  2
. . Q .  1
  A B C D
```

CONCLUSIONES

El **Problema de la cena de los filósofos** y el **Problema de las N-reinas** representan dos desafíos fundamentales en la computación y las matemáticas, abordando conceptos clave como la sincronización de procesos, la concurrencia y la búsqueda de soluciones óptimas.

Mientras que el primero ilustra los problemas de acceso a recursos compartidos y la necesidad de evitar situaciones como el deadlock y la starvation.

El segundo ejemplifica la complejidad de la exploración combinatoria y la importancia de técnicas como el backtracking.

Ambos enfoques nos permitieron trabajar con conceptos clave como la **recursividad** y la **sincronización de hilos**, adaptando cada solución al tipo de problema que enfrentamos.

BIBLIOGRAFÍA

1. **Problema de la cena de los filósofos**
   Wikipedia. (s.f.). *Problema de la cena de los filósofos*. Wikipedia, la enciclopedia libre. Recuperado el 11 de febrero de 2025, de https://es.wikipedia.org/wiki/Problema_de_la_cena_de_los_fil%C3%B3sofos

2. **Problema de las N-reinas (ocho reinas)**
   Wikipedia. (s.f.). *Problema de las ocho reinas*. Wikipedia, la enciclopedia libre. Recuperado el 11 de febrero de 2025, de https://es.wikipedia.org/wiki/Problema_de_las_ocho_reinas