ILERNA DAM 02

REINAS Y FILÓSOFOS

Alumnos:
Jose Echevarria
Pablo Ortiz

25/02/2025

ÍNDICE

Problema de las reinas

El problema de las N-Reinas es un desafío clásico en matemáticas y computación que consiste en colocar n reinas en un tablero de ajedrez de n x n de tal manera que ninguna de ellas pueda atacarse entre sí. Para lograr esto, se deben cumplir las siguientes condiciones:

1. No puede haber dos reinas en la misma fila.
2. No puede haber dos reinas en la misma columna.
3. No puede haber dos reinas en la misma diagonal, ni en dirección ascendente ni descendente.

Este problema tiene dos variantes principales:

- Búsqueda de una única solución válida para un valor dado de n.
- Cálculo de todas las soluciones posibles para un n determinado, lo que lo convierte en un problema más complejo computacionalmente.

El problema fue propuesto por primera vez en 1848 para n = 8 en un artículo anónimo, posteriormente atribuido al matemático Max Bezzel. Años más tarde, en 1850, Franz Nauck amplió el problema a un tamaño arbitrario de n. Carl Friedrich Gauss también estudió el problema, estimando que existían 72 soluciones para n = 8, aunque más tarde se comprobó que el número correcto es 92 (descubiertas por Glaisher en 1874).

Desde su planteamiento, este problema ha sido objeto de estudio en múltiples disciplinas y ha servido para evaluar y desarrollar nuevos algoritmos de búsqueda y optimización.

Uno de los aspectos más desafiantes del problema es que el número de soluciones crece de forma exponencial a medida que aumenta el valor de n. Por ejemplo, para n = 4, existen 2 soluciones, mientras que para n = 20, hay 39.029.188.884 soluciones. Esta rápida explosión combinatoria hace que encontrar todas las soluciones para valores grandes de n sea un desafío computacional considerable.

Solución Propuesta

```
private int n;
5 usages
private int[] tablero;

1 usage  Pablo Ortiz Muñoz
public NReinas(int n) {
    this.n = n;
    this.tablero = new int[n];
}
```

Se inician las variables y atributos

```
1 usage  Pablo Ortiz Muñoz *
private boolean esValido(int fila, int col) {
    for (int i = 0; i < fila; i++) {
        if (tablero[i] == col || Math.abs(tablero[i]
                - col) == Math.abs(i - fila)) {
            return false;
        }
    }
    return true;
}
```

Se crea un bucle para determinar si es posible dejar un a reina en una posición del tablero

```
2 usages   Pablo Ortiz Muñoz
private void resolver(int fila) {
    if (fila == n) {
        mostrarTablero();
        return;
    }
    for (int col = 0; col < n; col++) {
        if (esValido(fila, col)) {
            tablero[fila] = col;
            resolver( fila: fila + 1);
        }
    }
}
```

Este es un método recursivo para colocar las piezas el el orden correcto

```
1 usage   Pablo Ortiz Muñoz
private void mostrarTablero() {
    System.out.println("Solución encontrada:");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (tablero[i] == j) {
                System.out.print("Q ");
            } else {
                System.out.print(". ");
            }
        }
        System.out.println();
    }
    System.out.println();
}
```

Este bucle muestra por pantalla un tablero con las reinas

Problema de filósofos

El problema de los filósofos es un clásico en programación concurrente y sirve para ilustrar los desafíos de la sincronización y la posibilidad de que se produzcan interbloqueos (deadlocks). En este problema, varios filósofos (representados por procesos o hilos) comparten recursos limitados (los cubiertos) y deben coordinarse para evitar conflictos.

Descripción del problema:

- Hay **N filósofos** sentados alrededor de una mesa.
- Cada filósofo tiene un **cubierto a su izquierda** y un **cubierto a su derecha**.
- Para comer, un filósofo necesita **agarrar ambos cubiertos**.
- Si un filósofo no puede agarrar ambos cubiertos, debe esperar.
- Después de comer, el filósofo suelta los cubiertos y comienza a pensar.

El principal desafío es evitar que los filósofos se queden en un **interbloqueo**, donde cada filósofo agarra un cubierto y espera indefinidamente por el otro, sin soltar el que ya tiene.
La programación concurrente en Java proporciona facilidades para gestionar múltiples hilos, siendo un ejemplo clásico el problema de la Cena de los Filósofos. En este problema, los filósofos representan procesos y los tenedores, recursos compartidos de uso exclusivo.

Para poder resolver el problema de los filósofos, necesario cumplir los siguientes puntos:

- Exclusión mutua: Un proceso en su sección crítica impide que otros accedan a la suya.
- Progreso: Si nadie está en su sección crítica, los procesos que desean entrar deben poder hacerlo sin espera indefinida.
- Espera limitada: Un proceso no debe ser bloqueado indefinidamente.

El problema del deadlock

Una implementación incorrecta puede llevar a bloqueo indefinido (deadlock) si todos los filósofos toman un tenedor y esperan el otro. Para evitarlo, se debe romper al menos una de estas condiciones:

1. Exclusión mutua: Permitir compartir algunos recursos.
2. Retención y espera: Evitar que un proceso retenga un recurso mientras espera otro.
3. No apropiación: Permitir la reasignación de recursos.
4. Espera circular: Romper la cadena de espera entre procesos.

Solución Propuesta

```
private int id;
3 usages
private Semaphore izq, der;
2 usages
private volatile boolean seguirEjecutando = true; // Variable de control

1 usage  Pablo Ortiz Muñoz
public Filosofo(int id, Semaphore izq, Semaphore der) {
    this.id = id;
    this.izq = izq;
    this.der = der;
}
```

Declaramos atributos en la clase “Filósofo” las id de los filósofos y los semáforos que representan los cubiertos

```
public void run() {
    try {
        while (seguirEjecutando) {
            System.out.println("Filósofo " + id + " está pensando.");
            Thread.sleep((int) (Math.random() * 1000));
            //intenta tomar los cubiertos
            izq.acquire();
            der.acquire();
            System.out.println("Filósofo " + id + " está comiendo.");
            Thread.sleep((int) (Math.random() * 1000));
            //liberar cubiertos
            izq.release();
            der.release();
        }
        System.out.println("Filósofo " + id + " ha terminado.");
    } catch (InterruptedException e) {
        System.out.println("Filósofo " + id + " interrumpido.");
    }
```

En esta clase se simula los filósofos comiendo y esperando por los cubiertos

```java
public void detener() {
    seguirEjecutando = false;
    this.interrupt();
}
```

Esta clase es para ayudar que se detenga le bucle

```java
public static void iniciarFilosofos() {

    int numFilosofos = 5;
    Semaphore[] tenedores = new Semaphore[numFilosofos];
    for (int i = 0; i < numFilosofos; i++) {
        tenedores[i] = new Semaphore( permits: 1);
    }

    Filosofo[] filosofos = new Filosofo[numFilosofos];
    for (int i = 0; i < numFilosofos; i++) {
        filosofos[i] = new Filosofo(i, tenedores[i], tenedores[(i + 1) % numFilosofos]);
        filosofos[i].start();
    }

    // Espera 5 segundos y detiene a los filósofos
    try {
        Thread.sleep( millis: 5000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    //detener folosofos
    for (Filosofo f : filosofos) {
        f.detener();
    }

    // Esperar a que todos los hilos terminen
    for (Filosofo f : filosofos) {
        try {
            f.join(); // Asegura que el hilo haya terminado antes de continuar
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

En la clase “Filósofos Cena”, hay una única clase que inicia a los filósofos y cubiertos, y sea segura de cerrar todos los procesos para que el bucle no sea infinito

Conclusiones

La programación concurrente requiere una correcta sincronización para evitar problemas como condiciones de carrera, deadlocks y hambre de recursos. En el problema de la Cena de los Filósofos, cada filósofo representa un proceso y los tenedores son recursos compartidos. Si la sincronización no se maneja adecuadamente, algunos filósofos pueden quedarse sin comer indefinidamente, lo que refleja situaciones reales en sistemas operativos y bases de datos donde procesos pueden quedar bloqueados esperando recursos.

Para garantizar el correcto funcionamiento de un sistema concurrente, es fundamental cumplir con ciertas reglas, como la exclusión mutua, el progreso y la espera limitada. Sin estas condiciones, los procesos podrían acceder simultáneamente a recursos compartidos de manera descontrolada o quedar atrapados en bloqueos indefinidos. El problema del deadlock surge cuando los procesos retienen ciertos recursos mientras esperan otros, generando una cadena de espera circular que impide su avance.

Existen diversas estrategias para evitar bloqueos y mejorar la eficiencia en sistemas concurrentes. Una de ellas es la ordenación de recursos, que consiste en asignarlos en un orden predefinido para evitar esperas circulares. Otra técnica es la introducción de tiempos de espera aleatorios, donde los procesos que no pueden acceder a un recurso lo liberan temporalmente y lo intentan de nuevo más tarde, reduciendo la probabilidad de un deadlock. Además, algunos sistemas implementan algoritmos de detección y recuperación, que identifican bloqueos y liberan recursos para desbloquear los procesos afectados.

El problema de la Cena de los Filósofos tiene aplicaciones en diversos ámbitos, como la gestión de concurrencia en bases de datos, sistemas operativos y aplicaciones de alto rendimiento. Su estudio permite diseñar sistemas más eficientes y robustos, asegurando que los recursos se utilicen de manera equitativa y evitando bloqueos que afecten el rendimiento general. En definitiva, entender y aplicar correctamente los principios de sincronización y gestión de recursos en entornos concurrentes es clave para el desarrollo de software confiable y eficiente.

Bibliografía

Spading, A., & Itaim, P. (s.f.). *El Problema de las N-Reinas*. Anais.

Sistemas Operativos, Interbloqueo 5: El problema de los filósofos. (s.f.). [Video]. YouTube. https://www.youtube.com/watch?v=yFoq8mTL9RE

OpenAI. (2023). ChatGPT.