

# Prácticas Concurrencia y Distribución (18/19)

Arno Formella, Anália García Lourenço, Hugo López Fernández, David Olivieri

semana 11 febrero – 15 febrero

## 2. Semana 3 (11/02–15/02): Gestión de hilos y tareas concurrentes independientes

**Objetivos:** Gestión de hilos y tareas concurrentes independientes

### 1. (P4: para entregar en grupo de práctica): Variables locales del hilo y Interrupts

Este problema continúa explorando dos temas importantes en la administración de hilos: a) el alcance de las variables locales y b) la interrupción de hilos. Para hacer esto, sigue estos pasos:

- a) **Configuración del problema:** crea una clase de hilo vacío (por el momento) (`MiThread`) que *extends* `Thread` (o *implements* `Runnable`). También crea una clase *principal* (por ejemplo, `MiProblema`) donde dentro su método *main* crea una *array* (o `ArrayList`) del objeto `MiThread`.
- b) **Variables local del hilo:** En la clase `MiThread`, crea una variable privada de tipo `Integer` (con nombre como `miSuma`). ¿Qué sucede con respecto a este atributo privado cuando hay múltiples instancias de hilos ejecutándose? A continuación, sobrescribe el método `run()` para sumar los números de 0 a algún número `N` y guardarlo dentro de `miSuma`. Una estructura muy esquemática para la clase del hilo sería:

```
class MiThread {
    private Integer miSuma;
    //...
    @Override
    public void run() {
        //for loop () {
            //imprime "started", threadID, miSuma
            // dormir
            // incremente miSuma
        }
        //imprime "finished", threadID, miSuma
    }
}
```

Explicar el resultado ¿Se comporta como se esperaba?

- c) **Variables locales en un subproceso:** en el API de Java, `ThreadLocal<>` es un método que se puede usar para mantener variables locales dentro de hilos. Reescribe la clase anterior creando una variable local con el mecanismo `ThreadLocal<Integer>` (consulta la documentación de Java). Demostrar que con este mecanismo los hilos solo suman sus propias variables locales. Ten en cuenta que al utilizar `ThreadLocal<>`, tendrás que usar métodos como `get()` y `set()`.

- d) **Interrumpir un hilo desde Main:** cambia la tarea del hilo de la parte 2 para calcular la constante  $\pi$  (PI). Para ello, hay muchas fórmulas iterativas, pero una muy sencilla de implementar es la siguiente:

```
for (int i = 3; i < 100000; i += 2)
{
    if (negative)
        pi -= (1.0 / i);
    else
        pi += (1.0 / i);
    negative = !negative;
}
pi += 1.0;
pi *= 4.0;
```

A continuación, escribe el código apropiado en el programa principal para que interrumpa los hilos después de que el main duerma por un tiempo aleatorio. ¿Qué comportamiento observas? ¿Se interrumpe inmediatamente el hilo? ¿Cómo podrías cuantificar tu respuesta?

2. (P3: para entregar dentro de una semana): Estudio de dividir tareas concurrentes: dividir tareas en una matriz

Este problema estudia hilos concurrentes independientes. En particular, dada una matriz grande, cada hilo debe realizar un cálculo en una subregión de esta matriz. Un cálculo simple es el de un filtro numérico (típico en el procesamiento de imágenes), que reemplaza cada elemento con su promedio obtenido a partir de los valores de sus elementos vecinos inmediatos. El problema explora el rendimiento en función del número de hilos no interactivos y no sincronizados. Para hacer esto, sigue estos pasos:

- a) **Configuración del problema:** crea un class de hilo vacío (por el momento) (MiThread) que *extends* Thread (o *implements* Runnable). También crea una class *main* (por ejemplo, MiProblema) con el método principal que crea una *array* (o ArrayList) del objeto MiThread. Finalmente, crea una clase MiMatriz que representa un objeto de matriz de 2 dimensiones e implementa una estrategia para asignar bloques de la matriz a diferentes hilos.
- b) **Desarrollar la clase MiThread:** la tarea que se realizará se llama filtro mediano. Dada una matriz, el filtro mediano reemplaza cada elemento de la matriz  $(i, j)$  con el promedio calculado con los ocho vecinos y el mismo (ten cuidado cuando te encuentras en los bordes de la matriz). La ecuación para este filtro,  $J$ , es:

$$J(i, j) = \frac{1}{(2f + 1)^2} \sum_{k=-f}^f \sum_{l=-f}^f M(i + k, j + l)$$

donde  $M$  es la imagen original y  $f$  es el tamaño del filtro. Tratamiento de bordes de matriz: si  $M(i + k, j + l)$  resulta en un elemento fuera de la matriz; para tratar estos casos, puedes reflejar las coordenadas en el borde para simplificar el cálculo. Por ejemplo, puedes asumir que  $M(-2, -1) = M(2, 1)$ , e igual a lo largo de los demás bordes.

- c) **Desarrollar la clase MiMatriz:** esta clase representa el objeto matriz. También es responsable de distribuir los hilos de filtro por toda la matriz. La clase debe implementar un método con una estrategia particular para distribuir la lista de hilos. Una estrategia debería incluir la división de matriz por filas, columnas, bloques o cualquier combinación de los mismos.

- d)* **Desarrollar la clase principal:** esta clase principal es simplemente responsable de configurar el problema y crear el objeto `MiMatriz` que ejecutará el filtro dentro de un subregión. Tu código debería ser capaz de crear matrices de diferentes tamaños con elementos constantes o aleatorios.
- e)* **Estudio del comportamiento:** ejecuta tu programa para crear diferentes gráficos (plots) de matriz y número de hilos. A partir de los resultados de estos plots, ¿notas una diferencia en el rendimiento entre las diferentes estrategias? Si es así, ¿cuál podría ser la causa?