| | |
|---|---|
| | **Carátula para entrega de prácticas** |
| Facultad de Ingeniería | Laboratorio de docencia |

# Laboratorios de computación salas A y B

*Profesora:* Rocío Alejandra Aldeco Pérez

*Asignatura:* Programación Orientada a Objetos

*Grupo:* 6

*No de Práctica(s):* 12

*Integrante(s):* Ugalde Velasco Armando

*No. de Equipo de cómputo empleado:*

*No. de Lista o Brigada:*

*Semestre:* 2021-1

*Fecha de entrega:* 15 de enero de 2021

*Observaciones:*

CALIFICACIÓN: _____

**Práctica de Estudio 12: Hilos**
Programación Orientada a Objetos Grupo 6
Facultad de Ingeniería
Departamento de Computación

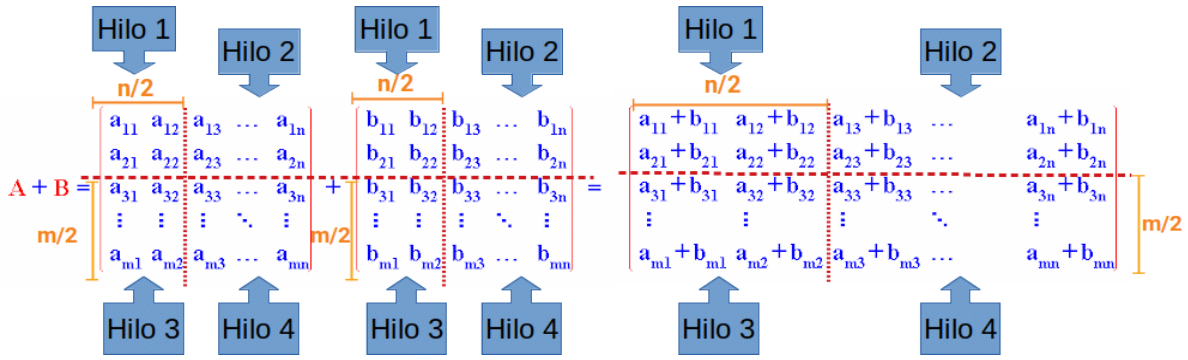
**Objetivo de la práctica:** Implementar el concepto de multitarea utilizando hilos en un lenguaje orientado a objetos.

Realiza las siguientes actividades después de leer y revisar en clase la **Práctica de Estudio 12: Hilos**.

1. Para esta práctica deberás realizar un programa que realiza la suma de dos matrices de la misma dimensión usando cuatro hilos.

2. Primero deberás de leer las matrices. Se proporcionará una fila por línea de entrada. Recuerda las matrices son cuadradas.

3. Posteriormente deberás asignar un cuarto del trabajo a cada hilo, de tal forma que al tener 4 hilos se distribuyan el trabajo como se muestra en la imagen




4. La suma la guardarás en una nueva matriz y la imprimirás con el mismo formato que fue leída.

5. Cuando estés seguro de que tu programa es correcto, súbelo a *Alphagrader*. Recuerda que si no pasas todos los test no obtendrás los puntos de ejecución.

6. Genera un documento **pdf** con la correspondiente carátula que explique el proceso de creación de este programa. Finalmente concluye. Recuerda subir este documento en PDF a Schoology junto con el código de esta práctica (los archivos .java).

Para cumplir con el objetivo planteado, se creó una clase que representara a una matriz: **Matrix**. Ésta, contendría la lógica necesaria para realizar la suma entre dos matrices utilizando **4** hilos y la estrategia planteada. A continuación, se muestran los atributos de la clase y sus constructores:

```java
Representa una matriz.
public class Matrix
{
    Número de hilos a utilizar al realizar la operación de suma.
    private static final int NUMBER_OF_THREADS = 4;

    Representación interna de ésta matriz por un arreglo bidimensional.
    private final int[][] representation;

    Construye una nueva matriz a partir de una representación en una lista bidimensional.
    Params: representation – Representación de la matriz.
    public Matrix(List<List<Integer>> representation)
    {
        int rows = representation.size();
        int cols = representation.get(0).size();
        this.representation = new int[rows][cols];
        for (int i = 0; i < rows; i++)
        {
            this.representation[i] = representation.get(i).stream().mapToInt(Integer::intValue).toArray();
        }
    }

    Construye una matriz vacía, con las dimensiones especificadas.
    Params: rows – Número de filas.
            cols – Número de columnas.
    private Matrix(int rows, int cols) { this.representation = new int[rows][cols]; }
```

***Atributos y constructores de la clase Matrix***

Como se puede observar, la clase cuenta con dos atributos: una constante estática y privada que contiene el número de hilos a utilizar **(4)**, y otro atributo privado que contiene la representación interna de la matriz, utilizando un arreglo bidimensional. El constructor público permite obtener una instancia de la clase al proporcionar la representación de una matriz por medio de una lista bidimensional. En cambio, el constructor privado permite obtener una instancia de la clase representando una matriz vacía, con las dimensiones especificadas.

Posteriormente, para implementar la funcionalidad de **suma**, se crearon algunos métodos estáticos auxiliares:

```java
/**
 * Suma los elementos presentes en la región indicada de dos matrices representadas por arreglos
 * bidimensionales, y coloca el resultado en las posiciones respectivas en la matriz result.
 * Params: matrix1 – Matriz a sumar.
 *         matrix2 – Matriz a sumar.
 *         result – Matriz donde se colocan los resultados.
 *         startX – Posición de la primera fila a sumar, con índices basados en 0.
 *         endX – Posición de la última fila a sumar, con índices basados en 0.
 *         startY – Posición de la primera columna a sumar, con índices basados en 0.
 *         endY – Posición de la última columna a sumar, con índices basados en 0.
 */
private static void sumQuadrant(int[][] matrix1, int[][] matrix2, int[][] result, int startX, int endX, int startY,
                                int endY)
{
    for (int i = startX; i ⩽ endX; i++)
    {
        for (int j = startY; j ⩽ endY; j++)
        {
            result[i][j] = matrix1[i][j] + matrix2[i][j];
        }
    }
}

/**
 * Computa la suma entre el cuadrante superior izquierdo de dos matrices. Coloca los resultados en las
 * posiciones respectivas en la matriz result.
 * Params: matrix1 – Matriz a sumar.
 *         matrix2 – Matriz a sumar.
 *         result – Matriz donde se colocan los resultados.
 */
private static void sumFirstQuadrant(Matrix matrix1, Matrix matrix2, Matrix result)
{
    sumQuadrant(matrix1.getInternalRepresentation(),
            matrix2.getInternalRepresentation(),
            result.getInternalRepresentation(),
             startX: 0,
             endX: matrix1.getNumberOfRows() / 2,
             startY: 0,
             endY: matrix1.getNumberOfColumns() / 2);
}
```

***Método utilizado para sumar cuadrantes***

El método **sumQuadrant** suma los elementos presentes en la región indicada de dos matrices representadas por arreglos bidimensionales, y coloca el resultado en las posiciones respectivas en la matriz **result**. Luego, el método **sumFirstQuadrant** computa la suma entre el cuadrante superior izquierdo de dos matrices, utilizando el método **sumQuadrant** y especificando los índices pertinentes. Para calcular los tres cuadrantes restantes, se implementaron los métodos respectivos utilizando una lógica equivalente al del método anterior.

A continuación, se muestra la implementación del método **sum**, que suma la matriz de esa instancia a la matriz **other**, y retorna una nueva matriz con el resultado:

```java
public Matrix sum(Matrix other)
{
    if (this.getNumberOfRows() ≠ other.getNumberOfRows() || this.getNumberOfColumns() ≠ other
            .getNumberOfColumns())
    {
        throw new IllegalArgumentException("Matrices should contain the same number of rows and columns");
    }

    Matrix result = new Matrix(this.getNumberOfRows(), this.getNumberOfColumns());
    Thread[] threads = new Thread[NUMBER_OF_THREADS];

    threads[0] = new Thread(() → Matrix.sumFirstQuadrant( matrix1: this, other, result));
    threads[1] = new Thread(() → Matrix.sumSecondQuadrant( matrix1: this, other, result));
    threads[2] = new Thread(() → Matrix.sumThirdQuadrant( matrix1: this, other, result));
    threads[3] = new Thread(() → Matrix.sumFourthQuadrant( matrix1: this, other, result));

    for (int i = 0; i < NUMBER_OF_THREADS; i++)
    {
        threads[i].start();
    }

    for (int i = 0; i < NUMBER_OF_THREADS; i++)
    {
        try
        {
            threads[i].join();
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
    return result;
}
```

***Método sum***

En este método, como se puede observar, primero se comprueba que las matrices tengan las mismas dimensiones, de lo contrario, se tira una excepción. Posteriormente, se crea una matriz vacía para almacenar el resultado y se crean e inicializan los hilos que computan los resultados para cada cuadrante. Finalmente, se espera a que todos los hilos terminen su ejecución al utilizar el método **join**, y, una vez que esto último ocurre, se retorna la matriz resultado.

Además, se implementaron algunos métodos auxiliares:

```java
public static Matrix parseMatrix(Scanner scanner)
{
    List<List<Integer>> matrixRepresentation = new ArrayList<>();
    while (scanner.hasNextLine())
    {
        String currentLine = scanner.nextLine();
        if (currentLine.equals("")) break;
        String[] numbers = currentLine.split( regex: " ");
        List<Integer> row = Arrays.stream(numbers)
                .filter(Predicate.not(String::isBlank))
                .map(Integer::parseInt)
                .collect(Collectors.toList());
        matrixRepresentation.add(row);
    }
    return new Matrix(matrixRepresentation);
}
```

***Método parseMatrix***

Este método analiza sintácticamente la entrada presente en ***scanner***, y, si ésta tiene una representación válida de una matriz, retorna una instancia de **Matrix** que la representa.

```java
Obtiene la representación de esta matriz como cadena. Cada fila de la matriz se representa como una
línea, y los elementos se encuentran separados por espacios.
Returns: La representación de esta matriz como cadena.
@Override
public String toString()
{
    StringBuilder stringBuilder = new StringBuilder();
    for (int i = 0; i < this.getNumberOfRows(); i++)
    {
        for (int j = 0; j < this.getNumberOfColumns(); j++)
        {
            stringBuilder.append(this.representation[i][j]).append(" ");
        }
        stringBuilder.replace( start: stringBuilder.length() - 1, Integer.MAX_VALUE, str: "\n");
    }
    return stringBuilder.toString();
}
```

***Método toString***

Además, se sobrescribió el método **toString**, para obtener la representación en cadena de la Matriz con el formato deseado.

Finalmente, en el método principal, se crean las matrices correspondientes, se calcula su suma, y se imprime la representación del resultado, utilizando los métodos ya mostrados:

```java
public static void main(String[] args)
{
    Scanner scanner = new Scanner(System.in);

    Matrix matrixA = Matrix.parseMatrix(scanner);
    Matrix matrixB = Matrix.parseMatrix(scanner);

    Matrix result = matrixA.sum(matrixB);
    System.out.println(result);
}
```

***Método principal***

## CONCLUSIONES

La **concurrencia** es un concepto fundamental en las Ciencias de la Computación, ya que nos permite, desde la vista del programador, ejecutar dos o más tareas de manera simultánea. Es importante recalcar que este concepto no es equivalente a **paralelismo**, donde las tareas realmente se realizan al mismo tiempo, en dos o más unidades de procesamiento distintas. Sin embargo, si se ejecuta un programa de forma concurrente en varios procesadores, es posible ejecutar tareas de forma paralela.

Una abstracción de tareas concurrentes provista por los sistemas operativos son los hilos, los cuales individualmente consisten en un flujo de ejecución de cierto programa. En Java, es posible utilizar la abstracción antes mencionada de diversas formas. Como se comentó, para utilizarlos es posible extender la clase **Thread**, o bien implementar la interfaz **Runnable**.