22/08/2025

Trabajo Práctico 8 – Threads

Ejercicio Nº 1

Crear una clase que derive de Thread o implemente la interface Runnable. Su constructor debe recibir el nombre del nuevo thread y en el método run() debe escribir en pantalla el nombre del thread mil veces a intervalos de entre 0 y 1 segundo. Luego debe imprimir HECHO!.
Construya una clase que ejecute al menos dos Threads con nombres diferentes en forma simultanea.

Ver: `ThreadSimple.java y ThreadSimpleTest.java`

Realizar el mismo ejercicio implementando la interface Runnable.
Ver: `TareaSimple.java y RunnableTest.java`

¿Por qué `implements Runnable` suele ser preferible a `extends Thread`?

Ejercicio N° 2

Realice una aplicación para la multiplicación de matrices utilizando Threads

Desafíos y Conceptos Clave:

1. División del Problema: ¿Cómo dividir el trabajo? La sugerencia de "cada multiplicación de vectores" es clave. Para calcular C[i][j], se necesita multiplicar la fila i de la matriz A por la columna j de la matriz B. Esta operación es una tarea perfecta para un hilo.
2. Paso de Parámetros: Cada hilo necesitará saber qué fila y qué columna debe procesar, y necesitará acceso a las matrices originales. Esto se soluciona pasando los datos necesarios a través del constructor de la clase Runnable o Thread.
3. Sincronización y Agregación de Resultados: Este es el punto más importante. Una vez que todos los hilos calculan sus valores individuales, ¿cómo sabe el programa principal que todos han terminado para poder imprimir la matriz resultante? Aquí es donde se debe introducir el método thread.join().

Propuesta de Estructura para la Solución:

1. Clase TareaMultiplicacion (implements Runnable):
    - Constructor: Debe recibir las dos matrices de entrada (A y B), la matriz de resultado (C), y el índice de la fila (fila) y columna (columna) que debe calcular.
    - Método run(): Contendrá el bucle que calcula el producto punto de la fila i de A y la columna j de B. El resultado se guarda directamente en resultado[fila][columna].
2. Clase MultiplicadorMatrices:
    - Método main() o un método multiplicar():
        - Define las matrices A, B y C (la de resultados, inicializada en ceros).
        - Crea una lista para guardar todos los hilos que se van a lanzar: List<Thread> hilos = new ArrayList<>();
        - Usa dos bucles for anidados para recorrer cada celda de la matriz de resultado C.
        - Dentro del bucle, por cada celda (i, j):
            - Crea una instancia de TareaMultiplicacion(A, B, C, i, j).
            - Crea un nuevo Thread con esa tarea.
            - Inicia el hilo con start().

- Añade el hilo a la lista hilos.

- **Sincronización**: Después de los bucles de creación, se necesita otro bucle para esperar a que todos terminen:

```
for (Thread hilo : hilos) {
    try {
        hilo.join(); // El hilo principal espera a que este hilo termine.
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

- **Finalización**: Una vez que el bucle join termina, se tiene la garantía de que todos los cálculos están hechos y se puede imprimir la matriz resultante.

Ejercicio N° 3

Optimización con un Pool de Hilos

En el ejercicio anterior, creamos un nuevo hilo por cada celda de la matriz resultante. Si bien esto funciona para matrices pequeñas, para matrices grandes (ej: 1000x1000) crear un millón de hilos es extremadamente ineficiente y podría colapsar el sistema. La creación y destrucción de hilos tiene un costo computacional significativo.

Una solución moderna y robusta es utilizar un Pool de Hilos. Un pool es una colección de hilos "trabajadores" preexistentes que esperan para ejecutar tareas. Esto evita el costo de crear un hilo nuevo por cada tarea y permite un control más fino sobre cuántos hilos se ejecutan simultáneamente.

En Java, la forma estándar de gestionar pools de hilos es a través del framework ExecutorService.

Consigna

Modifique la solución del Ejercicio N° 2 para que, en lugar de crear un new Thread() por cada cálculo, se utilice un ExecutorService con un número fijo de hilos (por ejemplo, el número de procesadores disponibles en la máquina) para ejecutar todas las tareas de multiplicación de vectores.

1. Crear el ExecutorService: Utilice Executors.newFixedThreadPool() para crear un pool con un tamaño adecuado. Puede obtener el número de procesadores con Runtime.getRuntime().availableProcessors().
2. Enviar las Tareas: En lugar de new Thread(tarea).start(), envíe cada objeto Runnable (la TareaMultiplicacion) al pool utilizando el método executor.submit(tarea).
3. Manejar el Apagado: Después de enviar todas las tareas, es crucial apagar el ExecutorService de forma ordenada.
    - Primero, llame a executor.shutdown(). Esto evita que se acepten nuevas tareas, pero permite que las ya enviadas terminen.
    - Luego, utilice executor.awaitTermination() dentro de un bloque try-catch para que el hilo principal espere a que todas las tareas en el pool finalicen antes de continuar y mostrar el resultado.

1. Rendimiento: ¿Qué ventaja de rendimiento ofrece un ExecutorService frente a la creación manual de hilos para este problema?
2. Recursos: ¿Qué ocurriría si intentaras resolver una multiplicación de matrices de 2000x2000 con el método del Ejercicio 2? ¿Y con el del Ejercicio 3?
3. Tipos de Pools: Investiga la diferencia entre newFixedThreadPool y newCachedThreadPool. ¿Cuándo podrías preferir uno sobre el otro?