

Taller de Programación

Guía 6

Estudiosos:

Juan David Rincon Muñoz

Julian Eduardo Lozano Rios

Universidad Manuela Beltran

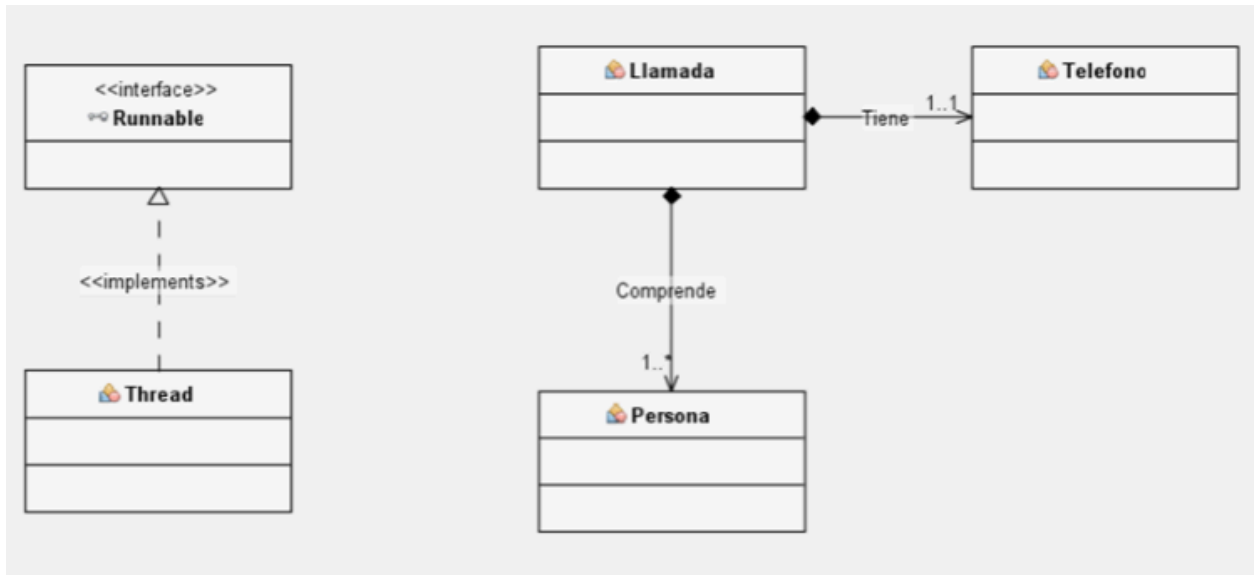
Ingeniería de software

Olga Lucia Roa

24 de octubre de 2024

Sesión 1

1. Analice y comprenda el siguiente diagrama de clases; cree un programa en java que permita simular un call Center haciendo uso de Hilos.



```
1 package call_center;
2 public class CallCenter {
3
4     public static void main(String[] args) {
5
6         Telefono telefon1 = new Telefono("463-1234");
7         Telefono telefon2 = new Telefono("298-5678");
8         Telefono telefon3 = new Telefono("375-9000");
9
10
11         Llamada llamada1 = new Llamada(telefon1);
12         Llamada llamada2 = new Llamada(telefon2);
13         Llamada llamada3 = new Llamada(telefon3);
14
15         Persona persona1 = new Persona("Juan", llamada1);
16         Persona persona2 = new Persona("Ana", llamada2);
17         Persona persona3 = new Persona("Luis", llamada3);
18
19         Thread hilo1 = new Thread(persona1);
20         Thread hilo2 = new Thread(persona2);
21         Thread hilo3 = new Thread(persona3);
22
23         hilo1.start();
24         hilo2.start();
25         hilo3.start();
26
27     }
```

The screenshot shows the IDE environment with the **CallCenter.java** file open. The code implements the **main** method, creating three **Telefono** objects, three **Llamada** objects (each associated with a **Telefono**), and three **Persona** objects (each associated with a **Llamada**). Three threads (**hilo1**, **hilo2**, **hilo3**) are created, each associated with a **Persona** object, and then started to simulate the call center process.

```
CallCenter.java x Telefono.java x Personajava x Llamada.java x
Source History
1 package call_center;
2
3
4 public class Llamada {
5     private Telefono telefono;
6
7     public Llamada(Telefono telefono) {
8         this.telefono = telefono;
9     }
10
11     public void realizarLlamada(String nombrePersona) {
12         System.out.println(nombrePersona + " esta llamando al numero " + telefono.getNumero ());
13         try {
14             // Simular el tiempo de la llamada
15             Thread.sleep(3000);
16         } catch (InterruptedException e) {
17             System.out.println("La llamada se corto.");
18         }
19         System.out.println("Llamada terminada por " + nombrePersona);
20     }
21 }
```

```
CallCenter.java x Telefono.java x Personajava x Llamada.java x
Source History
1 package call_center;
2
3 public class Telefono {
4     private String numero;
5
6     // Constructor
7     public Telefono(String numero) {
8         this.numero = numero;
9     }
10
11     // Método getNumero para obtener el número del teléfono
12     public String getNumero() {
13         return this.numero;
14     }
15 }
16
```

```
CallCenter.java x Telefono.java x Personajava x Llamada.java x
Source History
1 package call_center;
2
3 public class Persona implements Runnable {
4     private String nombre;
5     private Llamada llamada;
6
7     public Persona(String nombre, Llamada llamada) {
8         this.nombre = nombre;
9         this.llamada = llamada;
10     }
11
12     @Override
13     public void run() {
14         llamada.realizarLlamada(nombre);
15     }
16 }
17
```

```
Output - CallCenter (run)

run:
Juan esta llamando al numero 463-1234
Luis esta llamando al numero 375-9000
Ana esta llamando al numero 298-5678
Llamada terminada por Ana
Llamada terminada por Juan
Llamada terminada por Luis
BUILD SUCCESSFUL (total time: 3 seconds)
```

2. Desarrolle el siguiente ejercicio: "simular el proceso de cobro de un supermercado; es decir, unos clientes van con un carro lleno de productos y una cajera les cobra los productos, pasándose uno a uno por el escáner de la caja registradora. En este caso la cajera debe de procesar la compra cliente a cliente

```
public static void main(String[] args) {
    // Crear productos
    Producto producto1 = new Producto( nombre: "Manzanas", precio: 2.50);
    Producto producto2 = new Producto( nombre: "Pan", precio: 1.25);
    Producto producto3 = new Producto( nombre: "Leche", precio: 3.00);
    Producto producto4 = new Producto( nombre: "Arroz", precio: 1.75);

    // Crear cliente y agregar productos al carrito
    Cliente cliente1 = new Cliente( nombre: "Juan");
    cliente1.agregarProducto(producto1);
    cliente1.agregarProducto(producto2);
    cliente1.agregarProducto(producto3);

    Cliente cliente2 = new Cliente( nombre: "Maria");
    cliente2.agregarProducto(producto3);
    cliente2.agregarProducto(producto4);

    // Crear cajera
    Cajera cajera = new Cajera( nombre: "Laura");

    // Procesar compras
    cajera.procesarCompra(cliente1);
    cajera.procesarCompra(cliente2);
}
```

```

public class Producto { 12 usages new *

    private String nombre; 3 usages
    private double precio; 3 usages

    public Producto(String nombre, double precio) {
        this.nombre = nombre;
        this.precio = precio;
    }

    public String getNombre() { no usages new *
        return nombre;
    }

    public double getPrecio() { 1 usage new *
        return precio;
    }

    @Override new *
    public String toString() {
        return nombre + ": $" + precio;
    }
}

```

```

import java.util.ArrayList;
import java.util.List;

public class Cliente { 5 usages new *

    private String nombre; 2 usages
    private List<Producto> carrito; 3 usages

    public Cliente(String nombre) { 2 usages new *
        this.nombre = nombre;
        this.carrito = new ArrayList<>();
    }

    public void agregarProducto(Producto producto) {
        carrito.add(producto);
    }

    public List<Producto> getCarrito() { 1 usage new *
        return carrito;
    }

    public String getNombre() { 2 usages new *
        return nombre;
    }
}

```

```

public class Cajera { 2 usages new *

    private String nombre; 2 usages

    public Cajera(String nombre) { 1 usage new *
        this.nombre = nombre;
    }

    public void procesarCompra(Cliente cliente) { 2 usages new *
        System.out.println("Cajera " + nombre + " procesando la compra del cliente " + cliente.getNombre());
        double total = 0;
        int numeroProducto = 1;

        for (Producto producto : cliente.getCarrito()) {
            System.out.println("Producto " + numeroProducto + ": " + producto);
            total += producto.getPrecio();
            numeroProducto++;
            esperar(1000); // Simula el tiempo que tarda en pasar cada producto (1 segundo)
        }

        System.out.println("Total a pagar por " + cliente.getNombre() + ": $" + total);
        System.out.println("Compra procesada.\n");
    }

    // Simulación de tiempo de espera
    private void esperar(int milisegundos) { 1 usage new *
        try {
            Thread.sleep(milisegundos);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

```

```

Cajera Laura procesando la compra del cliente Juan
Producto 1: Manzanas: $2.5
Producto 2: Pan: $1.25
Producto 3: Leche: $3.0
Total a pagar por Juan: $6.75
Compra procesada.

```

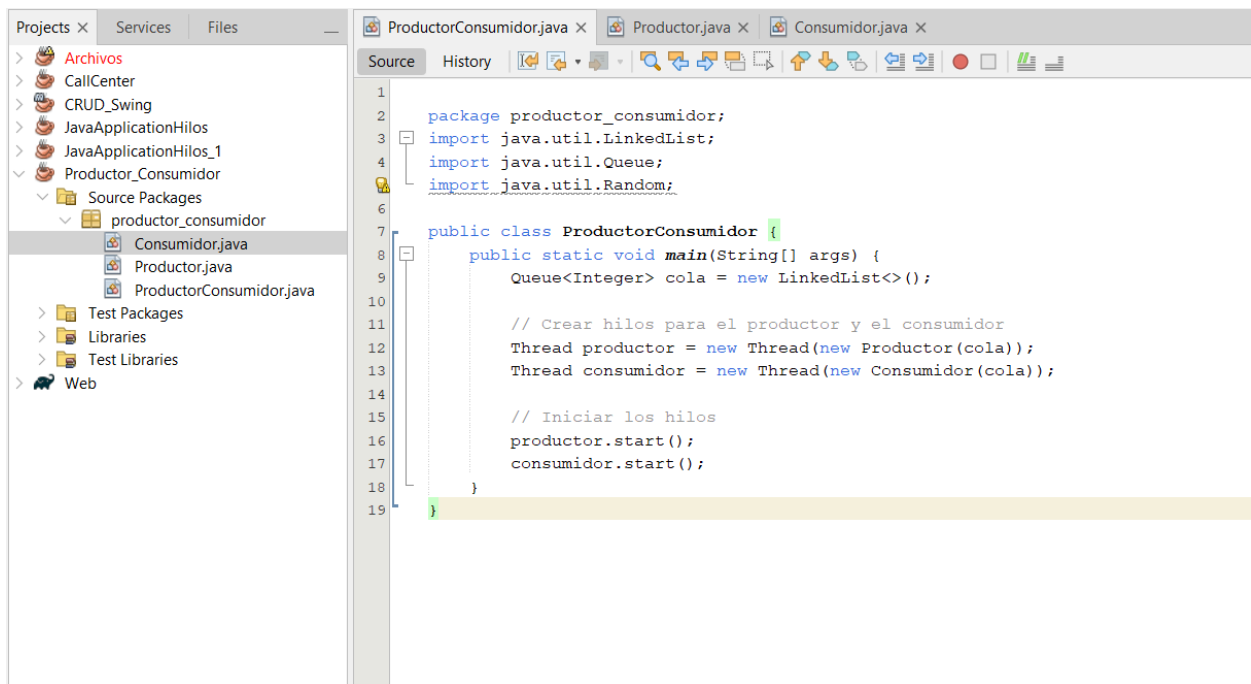
```

Cajera Laura procesando la compra del cliente Maria
Producto 1: Leche: $3.0
Producto 2: Arroz: $1.75
Total a pagar por Maria: $4.75
Compra procesada.

```

Sesión 2

1. Se le solicita realizar un ejemplo: productor/ consumidor que trabaje con 2 Threads, el primer Thread generará números aleatorios entre 1 y 100 que serán leídos y multiplicados por 2 en el segundo Thread el cual imprimirá el resultado







The screenshot shows an IDE with a project named 'Productor_Consumidor'. The project structure in the left pane includes 'Source Packages' and 'productor_consumidor', which contains 'Consumidor.java', 'Productor.java', and 'ProductorConsumidor.java'. The main editor displays the source code of 'ProductorConsumidor.java'.

```
1 package productor_consumidor;
2
3 import java.util.LinkedList;
4 import java.util.Queue;
5 import java.util.Random;
6
7 public class ProductorConsumidor {
8     public static void main(String[] args) {
9         Queue<Integer> cola = new LinkedList<>();
10
11         // Crear hilos para el productor y el consumidor
12         Thread productor = new Thread(new Productor(cola));
13         Thread consumidor = new Thread(new Consumidor(cola));
14
15         // Iniciar los hilos
16         productor.start();
17         consumidor.start();
18     }
19 }
```

```
ProductorConsumidor.java x Productor.java x Consumidor.java x
Source History
4 import java.util.Queue;
5 import java.util.Random;
6 class Productor implements Runnable {
7     private Queue<Integer> cola;
8     private final int LIMITE = 5; // Limite de tamaño de la cola
9     private Random random = new Random();
10
11     public Productor(Queue<Integer> cola) {
12         this.cola = cola;
13     }
14
15     @Override
16     public void run() {
17         while (true) {
18             synchronized (cola) {
19                 // Espera si la cola está llena
20                 while (cola.size() == LIMITE) {
21                     try {
22                         cola.wait();
23                     } catch (InterruptedException e) {
24                         e.printStackTrace();
25                     }
26                 }
27                 // Generar un número aleatorio
28                 int numero = random.nextInt(100) + 1;
29                 System.out.println("Productor genera: " + numero);
30                 cola.add(numero);
31                 // Notifica al consumidor que hay un número disponible
32                 cola.notify();
33             }
34             try {
35                 // Simulación de tiempo de espera
36                 Thread.sleep(2000);
37             } catch (InterruptedException e) {
38                 e.printStackTrace();
39             }
40         }
41     }
42 }
```

```
ProductorConsumidor.java x Productor.java x Consumidor.java x
Source History
1 package productor_consumidor;
2 import java.util.LinkedList;
3 import java.util.Queue;
4 import java.util.Random;
5 class Consumidor implements Runnable {
6     private Queue<Integer> cola;
7
8     public Consumidor(Queue<Integer> cola) {
9         this.cola = cola;
10     }
11
12     @Override
13     public void run() {
14         while (true) {
15             synchronized (cola) {
16                 // Espera si la cola está vacía
17                 while (cola.isEmpty()) {
18                     try {
19                         cola.wait();
20                     } catch (InterruptedException e) {
21                         e.printStackTrace();
22                     }
23                 }
24                 // Consumir el número
25                 int numero = cola.poll();
26                 int resultado = numero * 2;
27                 System.out.println("Consumidor procesa: " + numero + " * 2 = " + resultado);
28                 // Notifica al productor que puede generar más números
29                 cola.notify();
30             }
31             try {
32                 // Simulación de tiempo de espera
33                 Thread.sleep(2000);
34             } catch (InterruptedException e) {
35                 e.printStackTrace();
36             }
37         }
38     }
39 }
```


Output - Productor_Consumidor (run)

```
run:
Productor genera: 60
Consumidor procesa: 60 * 2 = 120
Productor genera: 22
Consumidor procesa: 22 * 2 = 44
Productor genera: 17
Consumidor procesa: 17 * 2 = 34
Productor genera: 2
Consumidor procesa: 2 * 2 = 4
Productor genera: 77
Consumidor procesa: 77 * 2 = 154
Productor genera: 89
Consumidor procesa: 89 * 2 = 178
Productor genera: 66
Consumidor procesa: 66 * 2 = 132
```

2. Implemente un programa secuencial que calcule el producto de dos grandes matrices. Después modifíquelo para que esta tarea se realice entre cuatro Threads, cada uno ocupado de un subconjunto de la matriz resultado. Mida el tiempo que emplea cada una de las versiones.

Secuencial

```
import java.util.Random;

public class Secuencial {

    public static void main(String[] args) {
        int size = 1000; // Tamaño de las matrices
        int[][] A = generarMatriz(size, size);
        int[][] B = generarMatriz(size, size);
        int[][] C = new int[size][size]; // Matriz resultado

        long inicio = System.currentTimeMillis(); // Iniciar tiempo

        // Multiplicación de matrices secuencial
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                C[i][j] = 0;
                for (int k = 0; k < size; k++) {
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }

        long fin = System.currentTimeMillis(); // Finalizar tiempo

        System.out.println("Tiempo secuencial: " + (fin - inicio) + " ms");
    }

    // Método para generar una matriz con valores aleatorios
    private static int[][] generarMatriz(int filas, int columnas) { 2 usages
        Random random = new Random();
        int[][] matriz = new int[filas][columnas];
        for (int i = 0; i < filas; i++) {
            for (int j = 0; j < columnas; j++) {
                matriz[i][j] = random.nextInt(bound: 10); // Números entre 0 y 9
            }
        }
        return matriz;
    }
}
```

Tiempo secuencial: 1585 ms

Paralelo

```

import java.util.Random;

public class Paralela { new *
    public static void main(String[] args) throws InterruptedException { new *
        int size = 1000; // Tamaño de las matrices
        int[][] A = generarMatriz(size, size);
        int[][] B = generarMatriz(size, size);
        int[][] C = new int[size][size]; // Matriz resultado

        long inicio = System.currentTimeMillis(); // Iniciar tiempo

        // Crear y lanzar 4 threads para hacer la multiplicación en paralelo
        Thread t1 = new Thread(new Multiplicador(A, B, C, inicio: 0, fin: size / 4));
        Thread t2 = new Thread(new Multiplicador(A, B, C, inicio: size / 4, fin: size / 2));
        Thread t3 = new Thread(new Multiplicador(A, B, C, inicio: size / 2, fin: 3 * size / 4));
        Thread t4 = new Thread(new Multiplicador(A, B, C, inicio: 3 * size / 4, fin: size));

        //El hilo 1 procesa las filas 0 a 249.
        //El hilo 2 procesa las filas 250 a 499.
        //El hilo 3 procesa las filas 500 a 749.
        //El hilo 4 procesa las filas 750 a 999.

        t1.start();
        t2.start();
        t3.start();
        t4.start();

        // Esperar a que los 4 threads terminen
        t1.join();
        t2.join();
        t3.join();
        t4.join();

        long fin = System.currentTimeMillis(); // Finalizar tiempo

        System.out.println("Tiempo paralelo: " + (fin - inicio) + " ms");
    }
}

```

```

// Método para generar una matriz con valores aleatorios
private static int[][] generarMatriz(int filas, int columnas) { 2 usages new *
    Random random = new Random();
    int[][] matriz = new int[filas][columnas];
    for (int i = 0; i < filas; i++) {
        for (int j = 0; j < columnas; j++) {
            matriz[i][j] = random.nextInt(bound: 10); // Números entre 0 y 9
        }
    }
    return matriz;
}
}

```

```
// Clase que representa la tarea que hará cada Thread
class Multiplicador implements Runnable { no usages new *
    private int[][] A, B, C; 3 usages
    private int inicio, fin; 2 usages

    public Multiplicador(int[][] A, int[][] B, int[][] C, int inicio, int fin) {
        this.A = A;
        this.B = B;
        this.C = C;
        this.inicio = inicio;
        this.fin = fin;
    }

    @Override new *
    public void run() {
        int size = A.length; // Asumimos matrices cuadradas
        for (int i = inicio; i < fin; i++) {
            for (int j = 0; j < size; j++) {
                C[i][j] = 0;
                for (int k = 0; k < size; k++) {
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }
    }
}
```

Tiempo paralelo: 478 ms

Preguntas Orientadoras.

1. ¿Cuáles fueron los aprendizajes obtenidos al realizar esta guía?, liste como mínimo 3 aprendizajes y relaciónelos con su futuro que hacer profesional.

Concurrencia y paralelismo: Uno de los principales aprendizajes fue comprender cómo distribuir el trabajo de manera efectiva utilizando hilos (threads). Esto es fundamental en el ámbito profesional, especialmente al desarrollar aplicaciones que requieren un alto rendimiento o que deben escalar en sistemas multicore. Este conocimiento es clave para optimizar el uso de recursos en sistemas modernos.

Optimización de tareas intensivas en cálculo: La comparación entre la versión secuencial y la paralela de la multiplicación de matrices permitió entender cómo la paralelización puede reducir significativamente los tiempos de ejecución en tareas intensivas en cálculos matemáticos. Esta habilidad es esencial para mejorar el rendimiento de aplicaciones que procesan grandes volúmenes de datos, algo muy relevante en áreas como el desarrollo de software para simulaciones científicas, inteligencia artificial o procesamiento de datos masivos.

Medición del rendimiento del software: Aprendí a medir el tiempo de ejecución y evaluar el impacto de diferentes enfoques de programación. Esto tiene una relación directa con mi futuro profesional, ya que siempre será necesario evaluar el rendimiento de las soluciones que se implementan para asegurarse de que son eficientes y escalables. Medir, analizar y optimizar el rendimiento es una habilidad valiosa para cualquier desarrollador de software.

2. ¿Dónde presentó mayor dificultad resolviendo la guía? y ¿cómo lo resolvieron?

Coordinación entre hilos: Una de las principales dificultades fue manejar correctamente la sincronización entre los hilos para evitar problemas de concurrencia y garantizar que cada hilo completara su trabajo correctamente antes de continuar. Esto fue resuelto utilizando el método `join()` para asegurar que el programa principal esperará a que todos los hilos terminaran antes de seguir adelante.

División de trabajo en subrangos: Otra dificultad fue calcular correctamente cómo dividir las matrices de forma equitativa entre los hilos. La solución fue usar una estrategia de dividir las filas en bloques iguales entre los hilos, asegurando que cada hilo procesara un subconjunto específico de filas sin superposición.

3. ¿Cuáles fueron las estrategias de solución?

Descomposición del problema: Dividir el problema en partes más pequeñas y manejables fue una estrategia clave. La tarea de multiplicación de matrices fue descompuesta en subproblemas (cada uno a cargo de un hilo), lo que facilitó la paralelización. Esta estrategia es útil en la resolución de problemas grandes en la vida profesional, ya que permite trabajar de manera modular y en paralelo.

Pruebas y medición del rendimiento: Para resolver las dificultades, se implementaron pruebas de rendimiento tanto en la versión secuencial como en la paralela. Comparar los tiempos de ejecución de ambos enfoques ayudó a ajustar la solución paralela y entender cuándo y cómo el paralelismo proporciona beneficios reales.

Uso de la documentación y ejemplos: Consultar la documentación y ejemplos de manejo de hilos y concurrencia en Java fue una estrategia útil para superar las dificultades técnicas. Esta estrategia es aplicable a cualquier área profesional, ya que el uso de recursos externos y la consulta de la documentación oficial permiten resolver problemas de forma más eficiente y profesional.

Actividad de Trabajo Autónomo

PRIMERA SESIÓN 2.

Consulta bibliográfica en bases de datos digitales.

Realiza la búsqueda de información sobre:

1. ¿Qué es el paralelismo de datos?

El paralelismo de datos es un paradigma informático paralelo en el que una tarea grande se divide en subtareas más pequeñas, independientes y procesadas simultáneamente. Este enfoque permite que diferentes procesadores o unidades de cálculo realicen la misma operación en múltiples elementos de datos al mismo tiempo.

El objetivo principal del paralelismo de datos es mejorar la eficiencia y la velocidad de la computación.

Características del paralelismo de datos:

- División de tareas: La tarea grande se divide en subtareas más pequeñas que se pueden procesar de forma independiente.

- Procesamiento simultáneo: Las subtareas se procesan simultáneamente en diferentes procesadores o unidades de cálculo.
- Mejora de la eficiencia: El paralelismo de datos puede mejorar la eficiencia de la computación al permitir que se realicen más tareas en menos tiempo.
- Velocidad mejorada: El paralelismo de datos puede mejorar la velocidad de la computación al permitir que se procesen más datos en menos tiempo.

Ejemplos de paralelismo de datos:

- Procesamiento de imágenes: El procesamiento de imágenes es un ejemplo común de paralelismo de datos. Las imágenes se pueden dividir en píxeles, y cada píxel se puede procesar de forma independiente.
- Cálculos científicos: Los cálculos científicos son otro ejemplo común de paralelismo de datos. Los cálculos científicos a menudo implican la realización de la misma operación en grandes conjuntos de datos.

2. ¿Qué es el paralelismo de tareas?

El paralelismo de tareas es un paradigma informático paralelo en el que se ejecutan múltiples tareas al mismo tiempo. Las tareas son unidades de trabajo que realizan una tarea específica. Se pueden usar para escribir código asíncrono y para hacer que una operación se produzca después de que finalice otra operación asíncrona. También se pueden usar para descomponer el trabajo paralelo en partes más pequeñas.

Características del paralelismo de tareas:

- Ejecución simultánea: Las tareas se ejecutan simultáneamente en diferentes procesadores o unidades de cálculo.
- Código asíncrono: El paralelismo de tareas se puede usar para escribir código asíncrono.
- Dependencias entre tareas: Las tareas pueden tener dependencias entre sí.
- Descomposición del trabajo paralelo: El paralelismo de tareas se puede usar para descomponer el trabajo paralelo en partes más pequeñas.

Ejemplos de paralelismo de tareas:

- Descarga de archivos: La descarga de archivos es un ejemplo común de paralelismo de tareas. Los archivos se pueden descargar en partes, y cada parte se puede descargar de forma independiente.
- Procesamiento de solicitudes web: El procesamiento de solicitudes web es otro ejemplo común de paralelismo de tareas. Las solicitudes web se pueden procesar de forma independiente, y cada solicitud se puede procesar en un hilo separado.

Referencias Bibliográficas:

-Pure Storage. (2023). ¿Qué es el paralelismo de datos? [en línea]. Disponible en:

<https://www.purestorage.com/la/knowledge/what-is-data-parallelism.html> [Consultado el 24 de octubre de 2024].

-Microsoft. (2023). Paralelismo de tareas (tiempo de ejecución de concurrencia) [en línea]. Disponible en:

<https://learn.microsoft.com/es-es/cpp/parallel/concrt/task-parallelism-concurrency-runtime?view=msvc-170> [Consultado el 24 de octubre de 2024].

SEGUNDA SESIÓN.

3. Implemente un ejemplo de paralelismo de datos y otro de paralelismo de tareas.

Paralelismo de datos

```
class DataParallelism implements Runnable { new *
    private int[] array; 2 usages
    private int start; 2 usages
    private int end; 2 usages
    private double sum; // Resultado parcial 3 usages

    public DataParallelism(int[] array, int start, int end) { 2 usages new *
        this.array = array;
        this.start = start;
        this.end = end;
        this.sum = 0;
    }

    @Override new *
    public void run() {
        for (int i = start; i < end; i++) {
            sum += array[i]; // Sumar elementos de la porción asignada
        }
    }

    public double getSum() { 2 usages new *
        return sum;
    }
}
```

```

public static void main(String[] args) throws InterruptedException {
    new *
    int size = 1000000;
    int[] array = new int[size];

    // Llenar el arreglo con valores
    for (int i = 0; i < size; i++) {
        array[i] = i + 1;
    }

    // Crear dos hilos para paralelizar el cálculo
    DataParallelism p1 = new DataParallelism(array, start: 0, end: size / 2);
    DataParallelism p2 = new DataParallelism(array, start: size / 2, size);

    Thread t1 = new Thread(p1);
    Thread t2 = new Thread(p2);

    long startTime = System.nanoTime();
    t1.start();
    t2.start();

    t1.join();
    t2.join();
    long endTime = System.nanoTime();

    // Calcular el promedio sumando las partes y dividiendo entre el tamaño total
    double totalSum = p1.getSum() + p2.getSum();
    double average = totalSum / size;

    System.out.println("Promedio: " + average);
    System.out.println("Tiempo de ejecución (ms): " + (endTime - startTime) / 1_000_000);
}
}

```

Promedio: 500000.5

Tiempo de ejecución (ms): 8

Process finished with exit code 0

Paralelismo de tareas

```
class Task1 implements Runnable { 2 usages new *
    private String searchFileName; 3 usages

    public Task1(String searchFileName) { 1 usage new *
        this.searchFileName = searchFileName;
    }

    @Override new *
    public void run() {
        System.out.println("Iniciando búsqueda del archivo: " + searchFileName);
        // Simulación de búsqueda de archivo
        try {
            Thread.sleep(millis: 2000); // Simular tiempo de búsqueda
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Archivo encontrado: " + searchFileName);
    }
}
```

```
class Task2 implements Runnable { 2 usages new *
    private String compressFileName; 3 usages

    public Task2(String compressFileName) { 1 usage new *
        this.compressFileName = compressFileName;
    }

    @Override new *
    public void run() {
        System.out.println("Iniciando compresión del archivo: " + compressFileName);
        // Simulación de compresión de archivo
        try {
            Thread.sleep(millis: 3000); // Simular tiempo de compresión
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Archivo comprimido: " + compressFileName);
    }
}
```

```

public class TaskParallelism { new *
    public static void main(String[] args) throws InterruptedException { new *
        // Crear tareas
        Task1 searchTask = new Task1( searchFileName: "documento.txt");
        Task2 compressTask = new Task2( compressFileName: "imagen.png");

        // Crear y lanzar hilos
        Thread t1 = new Thread(searchTask);
        Thread t2 = new Thread(compressTask);

        long startTime = System.nanoTime();
        t1.start();
        t2.start();

        // Esperar que ambas tareas terminen
        t1.join();
        t2.join();
        long endTime = System.nanoTime();

        System.out.println("Tareas completadas.");
        System.out.println("Tiempo total (ms): " + (endTime - startTime) / 1_000_000);
    }
}

```

```

Iniciando compresión del archivo: imagen.png
Iniciando búsqueda del archivo: documento.txt
Archivo encontrado: documento.txt
Archivo comprimido: imagen.png
Tareas completadas.
Tiempo total (ms): 3013

```

4. Consulte acerca de los potenciales problemas en el paralelismo de datos y de tareas.

Problemas Potenciales en el Paralelismo de Datos:

1. Condiciones de carrera:

- Ocurre cuando dos o más hilos intentan acceder y modificar simultáneamente el mismo recurso compartido, como una variable o una estructura de datos, sin la debida sincronización.
- Solución: Utilizar mecanismos de sincronización como bloques sincronizados (**synchronized** en Java) o variables atómicas para evitar que varios hilos accedan al mismo recurso al mismo tiempo.

2. Contención de memoria:

- Cuando varios hilos intentan acceder a la misma región de memoria al mismo tiempo, se puede generar un cuello de botella. Aunque los hilos son paralelos, el acceso a la memoria puede ser secuencial debido a las limitaciones del hardware.
- Solución: Minimizar el acceso compartido a la memoria y distribuir los datos de manera eficiente entre los hilos.

3. Falsa compartición (False sharing):

- Cuando diferentes hilos trabajan en datos que se encuentran muy próximos en memoria, el sistema de caché puede comportarse de manera ineficiente, invalidando y refrescando partes innecesarias de la caché, lo que reduce el rendimiento.

- Solución: Alinear los datos en la memoria para evitar que diferentes hilos trabajen en datos que se encuentren en la misma línea de caché.

4. Carga de trabajo desbalanceada:

- Si los datos no se dividen equitativamente entre los hilos, algunos hilos pueden terminar su trabajo antes que otros, lo que crea un desbalance y reduce la eficiencia del paralelismo.
- Solución: Implementar estrategias de balanceo de carga que distribuyan de manera equitativa los datos entre los hilos.

5. Sobrecarga de creación y gestión de hilos:

- Si el número de hilos es demasiado grande, el sistema operativo puede gastar más tiempo gestionando los hilos que realizando el trabajo real.
- Solución: Limitar el número de hilos en función del número de núcleos del procesador y el tipo de tarea que se esté ejecutando.

Problemas Potenciales en el Paralelismo de Tareas:

1. Sincronización de tareas:

- Cuando varias tareas dependen entre sí, es crucial garantizar que las tareas se completen en el orden adecuado o que los resultados de una tarea estén disponibles antes de que otra comience. Sin la sincronización adecuada, puede haber errores de ejecución o resultados incorrectos.
- Solución: Utilizar herramientas de sincronización como bloqueos (locks), semáforos o barreras para coordinar las tareas.

2. Sobrecarga de contexto:

- Cambiar el contexto entre diferentes tareas o hilos puede ser costoso en términos de rendimiento. Esto es particularmente problemático si hay muchas tareas y el sistema tiene que cambiar continuamente entre ellas.
- Solución: Minimizar el número de cambios de contexto mediante el uso de tareas más largas o menos frecuentes.

3. Problemas de dependencias:

- Si las tareas paralelas no son completamente independientes, pueden necesitar intercambiar datos o resultados, lo que puede causar problemas si no se gestiona correctamente. Por ejemplo, si una tarea necesita datos de otra tarea que aún no ha finalizado, esto introduce una dependencia que puede ralentizar el rendimiento.
- Solución: Utilizar técnicas como paso de mensajes o sincronización mediante eventos para gestionar las dependencias entre tareas.

4. Deadlocks (bloqueo mutuo):

- Los deadlocks ocurren cuando dos o más tareas quedan bloqueadas, esperando que la otra libere un recurso necesario para continuar. Esto puede hacer que todo el programa quede en un estado de espera indefinida.
- Solución: Evitar la adquisición de múltiples bloqueos y utilizar técnicas de prevención de deadlocks como la detección de ciclos o la adquisición ordenada de recursos.

5. Starvation (inanición):

- Ocurre cuando una tarea nunca obtiene acceso a los recursos que necesita para avanzar, porque otras tareas siempre ocupan esos recursos.
- Solución: Implementar políticas de planificación justas o utilizar bloqueos justos que aseguren que todas las tareas obtengan una oportunidad para ejecutar.

6. Sobrecarga de comunicación entre tareas:

- Si las tareas necesitan intercambiar datos con frecuencia, la comunicación entre ellas puede crear un cuello de botella. Esto es común en sistemas distribuidos o en paralelismo de tareas cuando las tareas deben coordinarse frecuentemente.
- Solución: Minimizar la necesidad de comunicación entre tareas y, cuando sea necesario, utilizar mecanismos de comunicación eficientes como colas concurrentes o buffers.

The student makes a video IN ENGLISH of a minimum of 1 and a maximum of 3 minutes exposing a summary of the theoretical framework of the guide and the answers requested in the independent work.

El estudioso realiza un video EN INGLÉS de mínimo 1 y máximo 3 minutos exponiendo un resumen del marco teórico de la guía y las respuestas solicitadas en el trabajo independiente.

github:

En este repositorio se encuentra la guía:

<https://github.com/juandatiner/Guias-Taller-de-Programacion>

En este repositorio se encuentran los códigos (Grandes matrices, Parallelism, Supermercado, Call center, Productor-Consumidor)

https://github.com/juandatiner/Juandatiner_DevVault/tree/Java/Before%20know%20plati

youtube:

https://youtu.be/mfDFPex_HJM?si=4i0KJ_-ZUrvy5yvi