Proyecto de Investigación

Java Concurrency

Universidad Nacional
Facultad de Ciencias Exactas y Naturales
Escuela de Informática

Curso
EIF400 Paradigmas de Programación

Profesor
Dr. Carlos Loría Sáenz

Estudiantes
Jorge Durán Campos
Jennifer Lobo Vásquez
Daniela Madrigal Morales
Diego Quirós Artiñano

II Ciclo 2023

Índice

1. Introducción

En el ámbito de la programación la concurrencia es la capacidad de descomponer un programa en partes que pueden ejecutarse independientemente unas de otras, a estas partes se les llama procesos o hilos (threads). Las distintas tareas en las que se compone un proceso se ejecutan secuencialmente unas detrás de otras de manera intercalada mediante el cambio de contexto, completándose en un período de tiempo superpuesto gestionando el acceso a recursos compartidos como lo serian la CPU y la memoria, todo esto ocurre en intervalos de tiempo tan cortos que parece que los procesos o hilos se ejecutan de manera simultánea.

La concurrencia permite desarrollar software en el que se ejecutan eventos que están ocurriendo o existiendo en un mismo momento, por lo que es la concurrencia está en todas partes en la programación moderna y a futuro va a continuar siendo esencial ya que aunque la velocidad de reloj del procesador ya no aumenta si hay un cambio continuo con la cantidad de un núcleos que van en aumento con cada nueva generación de chips, por lo que para que el software se ejecute más rápido tendrá que ser dividido en partes concurrentes.

El objetivo del presente documento es proporcionar información clara y concisa sobre lo que es la concurrencia y como se esta se desarrolla en java, esto por medio de ejemplos, aclaración de conceptos. . . etc.

2. Nociones básicas de concurrencia

2.1. Conceptos básicos de concurrencia

- **Rendimiento (Performance):** Se refiere a la medida de la eficiencia y la velocidad con la que una aplicación cumple con sus tareas y objetivos.
- **Proceso:** Es un programa en ejecución que cuenta con espacio de memoria y recursos propios. Cada proceso tiene al menos un hilo de ejecución llamado hilo principal y puede contar con múltiples hilos de trabajo.
- **Tareas:** Son unidades de trabajo que pueden ejecutarse de manera independiente dentro de un programa o proceso. A diferencia de los procesos, las tareas no son entidades independientes ya que comparten los mismos recursos que el proceso principal que las contiene. Son utilizadas para descomponer un programa en partes más pequeñas y manejables que pueden ejecutarse concurrentemente.
- **Hilos:** Son las unidades de ejecución más pequeñas dentro de un proceso. Varios hilos pueden existir dentro de un proceso y comparten el mismo espacio de memoria y recursos del proceso principal.
- **Multihilo:** Capacidad de un programa para ejecutar múltiples hilos de manera concurrente.
- **Programa:** Conjunto de sentencias/instrucciones que se ejecutan secuencialmente.
- **Executor:** Es una interfaz en Java que proporciona una abstracción para la ejecución de tareas e hilos de forma asíncrona, administrando la ejecución de tareas en un grupo de hilos, esta interfaz se encuentra en el paquete 'java.util.concurrent'.
- **Concurrencia vs Paralelismo:** Estos dos conceptos suelen confundirse, ya que están relacionados, pero no son lo mismo. El paralelismo se refiere a la ejecución simultánea de varios procesos, para lo cual se requieren varios medios de ejecución física, por lo que está relacionado con la capacidad del sistema en el que se ejecuta el programa. Por otro lado, la concurrencia se podría entender como el encuentro de varios procesos en un mismo intervalo de tiempo, que se ejecutan de forma independiente pero no necesariamente al mismo tiempo.

2.2. Problemas comunes en la concurrencia

- **Condiciones de carrera (Race Conditions)**: Estos problemas ocurren cuando múltiples hilos acceden o modifican datos simultáneamente, lo que puede llevar a resultados inesperados debido a la secuencia en la que se ejecutan los hilos.
- **Bloqueo mutuo (Deadlock)**: Un deadlock se produce cuando dos o más hilos se bloquean entre sí, esperando que el otro libere un recurso que necesitan, dando como resultado que los hilos queden atrapados y la aplicación se bloquea.
- **Inanición (Starvation)**: Se da cuando no tiene acceso a los recursos necesarios para su ejecución debido a que otros hilos tienen una prioridad más alta, como consecuencia puede que el hilo nunca termine su ejecución.
- **Problemas de sincronización**: Una mala sincronización puede llevar a errores como lectura o modificación incorrecta de datos compartidos, bloqueo de hilos o incluso la falta de bloqueos causando problemas difíciles de corregir ya que son mayormente visibles en el proceso de ejecución.

2.3. Ventajas y desventajas de la concurrencia

La concurrencia en Java conlleva a mejoras significativas en el rendimiento y la utilización eficiente de recursos, sin embargo, también conlleva a desafíos que deben ser considerados.

2.3.1. Ventajas de la concurrencia

- **Mejora del rendimiento**: La concurrencia puede aumentar significativamente el rendimiento de un sistema ya que al manejar o permitir que se ejecuten múltiples tareas de manera intercalada o simultánea se aprovechan mejor recursos como la CPU y la memoria.
- **Mayor utilización de recursos**: La concurrencia permite aprovechar al máximo y de manera eficiente los recursos disponibles, como la CPU y la memoria, lo que puede ser beneficioso en sistemas con múltiples núcleos de procesador, aumentando la velocidad y capacidad de respuesta.

2.3.2. Desventajas de la concurrencia

- **Complejidad**: Debido a que varios procesos se pueden ejecutar de forma simultánea, la programación concurrente se considera compleja y propensa a errores en caso de que no se maneje adecuadamente. Por lo tanto, se requiere un conocimiento profundo de la forma en que los procesos interactúan entre sí para evitar problemas.
- **Dificultad de depuración**: Los errores en la concurrencia pueden ser difíciles de diagnosticar y depurar, ya que suelen ocurrir bajo situaciones o condiciones específicas y dependen de la concurrencia de eventos, lo que dificulta su reproducción para poder diagnosticar el problema.
- **Consumo de recursos**: La concurrencia puede consumir una cantidad excesiva de recursos como memoria y CPU, lo que podría llevar a reducir el rendimiento en lugar de mejorarlo si no se gestiona adecuadamente.

3. Concurrencia en java

En el lenguaje de programación Java hay dos unidades básicas de ejecución fundamentales en la programación concurrente, hilos y procesos, sin embargo, aunque ambos son importantes, Java se ocupa principalmente de los hilos mas que de los procesos, esta característica convierte a Java en un lenguaje particularmente orientado hacia la ejecución multihilo. En java la ejecución multihilo es una característica esencial, ya que la JVM (Máquina Virtual de Java) está diseñada para ser un sistema multihilo, lo que permite que un programa pueda llevar a cabo múltiples tareas simultáneamente, llevando a mejoras significativas en el rendimiento y en la capacidad de respuesta de la aplicación.

4. Modelo de hilos en Java

4.1. Creación y manejo de hilos

Java cuenta con dos estrategias para la creación y gestión de hilos, una de ellas es extendiendo de `java.lang.Thread` y por otro lado implementando la interfaz `java.lang.Runnable`, ambas son detalladas en este apartado mediante ejemplos sencillos.

4.1.1. Creación de una instancia de la clase Thread

En esta estrategia se crea una instancia de la clase `Thread` cada vez que se necesite iniciar una tarea asincrónica o concurrente, para ello se detallan los siguientes pasos a seguir:

- Crear una clase que extienda de la clase `Thread`, dicha clase representará la tarea a ejecutar de forma concurrente.
- Se anula el método `run()` en la clase creada con el fin de colocar las instrucciones que se espera ejecute el nuevo hilo dentro de ese método.
- Crear una instancia de la clase que extiende `Thread`, dicha instancia representa la tarea que se desea ejecutar en un hilo.
- Llamar el método `start()` en la instancia del hilo con el fin de iniciar la ejecución de la tarea en un nuevo hilo.

Listing 1: Ejemplo de código Java con Thread

```
public class ThreadPrueba1 extends Thread {
    private String nombreHilo;

    public ThreadPrueba1(String nombreHilo) {
        this.nombreHilo = nombreHilo;
    }

    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Iteraci n-" + i + ":-" + nombreHilo);
            try {
                sleep(1000);
            } catch (InterruptedException e) {
            }
        }
    }

    public static void main(String[] args) {
        ThreadPrueba1 hilo1 = new ThreadPrueba1("Hilo1");
        ThreadPrueba1 hilo2 = new ThreadPrueba1("Hilo2");

        hilo1.start();
```

```
        hilo2.start();
    }
}
```

Esta estrategia es adecuada para casos en los que es necesario un control preciso sobre la creación y gestión de hilos, sin embargo, se puede volver complicado si se implementa con aplicaciones grandes.

4.1.2. Implementación de la interfaz Runnable

Esta estrategia se utiliza para crear tareas o trabajos que pueden ejecutarse en un hilo de manera concurrente además de brindar mayor flexibilidad y modularidad que la estrategia anterior. La interfaz `Runnable` es una forma de representar una tarea que se puede ejecutar en segundo plano sin tener que extender la clase `Thread`. Para llevar a cabo su implementación se detallan los siguientes pasos:

- Crear una clase que implemente la interfaz `Runnable`, dicha clase representará la tarea a ejecutar de forma concurrente.
- Se anula el método `run()` en la clase creada con el fin de colocar las instrucciones que se espera ejecute el nuevo hilo dentro de ese método.
- Crear una instancia de la clase que implemente `Runnable`, dicha instancia representa la tarea que se desea ejecutar en un hilo.
- Ejecutar la tarea en un hilo creando un objeto `Thread` para luego pasarle la instancia del `Runnable` en su constructor.
- Por último, se inicializa el hilo llamando al método `start()`, esto comenzará la ejecución de la tarea en segundo plano.

Listing 2: Ejemplo de código Java

```
public class RunnablePrueba1 implements Runnable {
    private String nombreHilo;

    public RunnablePrueba1(String nombreHilo) {
        this.nombreHilo = nombreHilo;
    }

    public void run() {
        for (int i = 1; i <= 5; i++) {
```

```
            System.out.println("Iteraci n-" + i + ":-" + nombreHilo);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
        }
    }

    public static void main(String[] args) {
        RunnablePrueba1 runnable1 = new RunnablePrueba1("Hilo1");
        RunnablePrueba1 runnable2 = new RunnablePrueba1("Hilo2");

        Thread hilo1 = new Thread(runnable1);
        Thread hilo2 = new Thread(runnable2);

        hilo1.start();
        hilo2.start();
    }
}
```

Con base en los dos ejemplos anteriores se puede ver que ambas formas de implementar los Threads en Java nos llevan a una solución valida, sin embargo, hay algunas consideraciones que se pueden tener en cuanta a la hora de decidirse por una o la otra. Al implementar la interfaz Runnable se puede utilizar la misma instancia para crear varios hilos además permite separar la lógica de la tarea del manejo y control de hilos, facilitando el mantenimiento. Por otro lado, extender de la clase Thread brinda mayor facilidad al ejecutar tareas simples, ya que no se necesita crear un objeto Thread adicional debido a que se puede inicializar el hilo desde la instancia de la clase que extiende de Thread, sin embargo, al Java no permitir la herencia múltiple se generan limitaciones si la clase ya extiende de otra clase.

4.2. Problemas comunes en la concurrencia en Java

La concurrencia en Java puede llevar a problemas complejos de diagnosticar, algunos de ellos son:

4.2.1. Condición de carrera (race condition)

Se producen cuando el resultado de un programa depende de la secuencia de ejecución de sus instrucciones por lo que suelen darse en aplicaciones multihilo, cuando dos o más hilos compiten por acceder a un recurso compartido sin una sincronización adecuada, dando resultados errores.

Listing 3: Ejemplo de código Java con condición de carrera

```
public class RaceConditionEjemplo1 {
    private static int contador = 0;
    private static int valor = 0;

    public static void main(String[] args) {

        Thread hilo1 = new Thread(() -> {
            for (int i = 0; i < 6; i++) {
                contador++;
                System.out.println("Hilo 1: Iteracion " + i
                + " valor actual del contador: " + contador);
            }
        });

        Thread hilo2 =
            new Thread(() -> {
                valor = contador / 2;
                System.out.println("Hilo 2: Operacion " +
                contador + "/2" + " =" + valor);
            });

        hilo1.start();
        hilo2.start();

        try {
            hilo1.join();
            hilo2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Valor final del contador: " + contador);
```

```
        System.out.println("Resultado final de la operacion sobre" +
        "el contador:" + valor);
    }
}
```

```
resultados:
Hilo 2: Operacion 1/2 =0
Hilo 1: Iteración 0 valor actual del contador: 1
Hilo 1: Iteración 1 valor actual del contador: 2
Hilo 1: Iteración 2 valor actual del contador: 3
Hilo 1: Iteración 3 valor actual del contador: 4
Hilo 1: Iteración 4 valor actual del contador: 5
Hilo 1: Iteración 5 valor actual del contador: 6
Valor final del contador: 6
```

En el ejemplo se puede visualizar que hay un problema de acarreo ya que ambos hilos están compitiendo por acceder y modificar la variable compartida 'contador' sin una sincronización adecuada por lo que se realizan operaciones sobre la variable sin un orden determinado, causando que el resultado varie cada vez que se ejecute.

Por medio de la implementación de la clase 'CountDownLatch' que proporciona una forma de coordinar y sincronizar múltiples hilos para que esperen hasta que se cumpla una condición antes de continuar se puede resolver el problema de carrera del ejemplo anterior.

Listing 4: Ejemplo de código Java con condición de carrera y CountDownLatch

```
import java.util.concurrent.CountDownLatch;

public class RaceConditionEjemplo2 {
    private static int contador = 0;
    private static int valor = 0;

    public static void main(String[] args) {
        // Se utiliza para sincronizar la ejecuci n de varios hilos
        CountDownLatch latch = new CountDownLatch(1);

        Thread hilo1 = new Thread(() -> {
            for (int i = 0; i < 6; i++) {
                contador++;
```

```
                System.out.println("Hilo 1: Iteraci n " + i
                + " valor actual del contador: " + contador);
            }
            latch.countDown();
        });

        Thread hilo2 = new Thread(() -> {
            try {
                // Espera a que el hilo1 complete su ejecucion
                latch.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            valor = contador / 2;
            System.out.println("Hilo 2: Operacion " + contador
            + "/2" + " =" + valor);
        });

        hilo1.start();
        hilo2.start();

        try {
            hilo1.join();
            hilo2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Valor final del contador: " + contador);
        System.out.println("Resultado final de la operaci n sobre el"
        + " contador: " + valor);
    }
}
```

```
resultados:
Hilo 1: Iteración 0 valor actual del contador: 1
Hilo 1: Iteración 1 valor actual del contador: 2
Hilo 1: Iteración 2 valor actual del contador: 3
Hilo 1: Iteración 3 valor actual del contador: 4
Hilo 1: Iteración 4 valor actual del contador: 5
```

```
Hilo 1: Iteración 5 valor actual del contador: 6
Hilo 2: Operacion 6/2 =3
Valor final del contador: 6
```

4.2.2. Deadlock

Un deadlock también conocido como interbloqueo, se da cuando un conjunto de hilos que compiten por los recursos del sistema se bloquean permanentemente al no conseguir acceder a dichos recursos, por ende un conjunto de hilos esta deadlock cuando cada hilo del conjunto está esperando un evento que sólo otro hilo del conjunto puede causar, por lo que al estar cada uno en estado de espera ninguno de ellos podría causar un evento que active a alguno de los otros miembros causando que cada mimbro del grupo espere indefinidamente a tener accesos al recurso que necesita.

Listing 5: Ejemplo de código Java que muestra un deadlock

```
/*
Codigo proporcionado por un asistente de OpenAI
como parte de una conversaci n de asistencia t cnica.
Fuente: OpenAI GPT-3.5, asistente virtual de
procesamiento de lenguaje natural.
*/

public class DeadlockEjemplo1 {
    private static Object recurso1 = new Object();
    private static Object recurso2 = new Object();

    public static void main(String[] args) {
        Thread hilo1 = new Thread(() -> {
            synchronized (recurso1) {
                System.out.println("Hilo 1: Bloqueando recurso 1");
                try { Thread.sleep(100); } catch (InterruptedException e)
                {}
                System.out.println("Hilo 1: Esperando por recurso 2");
                synchronized (recurso2) {
                    System.out.println("Hilo 1: Accediendo a recurso 2");
                }
            }
        });
```

```
        Thread hilo2 = new Thread(() -> {
            synchronized (recurso2) {
                System.out.println("Hilo 2: Bloqueando recurso 2");
                try { Thread.sleep(100); } catch (InterruptedException e)
                {}
                System.out.println("Hilo 2: Esperando por recurso 1");
                synchronized (recurso1) {
                    System.out.println("Hilo 2: Accediendo a recurso 1");
                }
            }
        });

        hilo1.start();
        hilo2.start();
    }
}
```

```
resultados:
Hilo 1: Bloqueando recurso 1
Hilo 2: Bloqueando recurso 2
Hilo 1: Esperando por recurso 2
Hilo 2: Esperando por recurso 1
```

En el ejemplo anterior se muestra como ambos hilos bloquean un recurso que el otro necesitaba para continuar con su ejecución por loque quedan es estado de espera indefinido.

4.2.3. Inanición (starvation)

La inanición se produce cuando un hilo de baja prioridad se ve privado de los recursos necesarios para su ejecución debido a que otros hilos tienen mayor prioridad y acaparan esos recursos, provocando que espere indefinidamente. Una forma de resolver o evitar este problema es por medio del envejecimiento, el cual aumenta gradualmente la prioridad de un hilo que ha estado esperando durante mucho tiempo para tener acceso a los recursos.

Un ejemplo textual de inanición podría ser el siguiente:

- En una soda estudiantil hay una fila con 3 personas que esperan ordenar sus alimentos, la primera persona en llegar a la fila lleva esperando 5 minutos, la segunda 4 y la tercera acaba de llegar, a esto el encargado de entender atiende primero al tercer cliente para luego pasar a

entender a cada nuevo cliente que llega, dejando al primer y segundo cliente esperando indefinidamente a que se les brinden los recursos que necesitan, en este caso hacer su pedido. En este ejemplo el primer y segundo cliente son los que presentan la prioridad más baja en comparación con los clientes nuevos que llegan, una solución a esto sería el envejecimiento que implicaría que con el tiempo el primer y segundo cliente ganen prioridad gradualmente, lo que eventualmente les permitiría ordenar.

4.2.4. Conclusion

La conclusión será redactada en la recta final de la investigación, cuando se cuente con mayor información y conocimientos.

Referencias

[1] D. Díaz Rodriguez, *Introducción a la Concurrencia en Java (I)*, En línea, 29 de ago. de 2019. dirección: https://blog.softtek.com/es/java-concurrency.

[2] *Diferencia entre Deadlock y Starvation en OS*, https://es.gadget-info.com/difference-between-deadlock, En línea, 2019.

[3] B. Kurniawan, *Java for Android*. Brainy Software, 2014. dirección: https://ebookcentral.proquest.com/lib/sidunalibro-ebooks/detail.action?docID=3003889.

[4] OpenAI. "OpenAI ChatGPT." En línea. (2021), dirección: https://openai.com/research/chatgpt.

[5] Oracle, *Defining and Starting a Thread (The Java™ Tutorials > Essential Java Classes > Concurrency)*, En línea. dirección: https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html.

[6] H. Patricio, "La diferencia entre concurrencia y paralelismo," *Dojo MX Blog*, 17 de abr. de 2019, En línea. dirección: https://blog.thedojo.mx/2019/04/17/la-diferencia-entre-concurrencia-y-paralelismo.html.

[7] picodotdev, "Iniciación a la programación concurrente en Java," *Blog Bitix*, 15 de jul. de 2017, En línea. dirección: https://picodotdev.github.io/blog-bitix/2017/07/iniciacion-a-la-programacion-concurrente-en-java.

[8] Radio Michi, *xn–h1h Ventajas y Desventajas de la Programación Concurrente Radio Michi*, En línea, 14 de dic. de 2022. dirección: https://radiomichi.com/ventajas-y-desventajas-de-la-programacion-concurrente.

[9] UNIR Revista, *¿Qué es la programación concurrente? | UNIR*, En línea, 30 de mar. de 2023. dirección: https://www.unir.net/ingenieria/revista/programacion-concurrente.