Synchronisation des Threads en Java

Plongeons dans la synchronisation des threads en Java ! Lorsque plusieurs threads accèdent à des ressources partagées (comme des variables, des collections ou des fichiers), vous devez empêcher les conditions de course —où les threads interfèrent les uns avec les autres, conduisant à des résultats imprévisibles. Java fournit plusieurs outils pour cela : `synchronized`, `Lock`, et `Semaphore`. Je vais expliquer chacun, montrer des exemples et les comparer afin que vous puissiez choisir le bon pour vos besoins.

---

1. Mot-clé `synchronized`

Le mot-clé `synchronized` garantit qu'un seul thread peut exécuter un bloc de code ou une méthode à la fois. C'est l'outil de synchronisation le plus simple, intégré au cœur de Java.

Fonctionnement

- Verrouille le moniteur d'un objet (chaque objet Java en possède un).
- Les autres threads attendent jusqu'à ce que le verrou soit libéré.

Exemple : Compteur avec `synchronized`

```
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Runnable task = () -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
```

```
            }
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Compte final : " + counter.getCount()); // Toujours 2000
    }
}
```

- Sans `synchronized`, `count++` (lecture-modification-écriture) pourrait se chevaucher, donnant un résultat inférieur à 2000.
- `synchronized` verrouille l'instance `Counter`, assurant des mises à jour atomiques.

Bloc Synchronisé Vous pouvez également verrouiller un code spécifique au lieu d'une méthode entière :

```
public void increment() {
    synchronized (this) {
        count++;
    }
}
```

- Utilisez un objet différent comme verrou si vous souhaitez un contrôle plus fin :

```
private final Object lock = new Object();
public void increment() {
    synchronized (lock) {
        count++;
    }
}
```

Avantages et Inconvénients

- **Avantages** : Simple, intégré, bon pour une utilisation de base.
- **Inconvénients** : Pas de flexibilité (par exemple, ne peut pas interrompre un thread en attente), bloque jusqu'à libération.

2. Interface `Lock` (java.util.concurrent.locks)

L'interface `Lock` (par exemple, `ReentrantLock`) offre plus de contrôle que `synchronized`. Elle fait partie du package `java.util.concurrent` et a été introduite pour pallier les limitations de `synchronized`.

Fonctionnalités Clés

- Appels explicites `lock()` et `unlock()`.
- Prend en charge les verrouillages conditionnels, les délais d'attente et les verrouillages interruptibles.
- Option d'équité (threads attendent dans l'ordre).

Exemple : Compteur avec `ReentrantLock`

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class Counter {
    private int count = 0;
    private final Lock lock = new ReentrantLock();

    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock(); // Toujours déverrouiller dans finally
        }
    }

    public int getCount() {
        return count;
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Runnable task = () -> {
```

```
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Compte final : " + counter.getCount()); // Toujours 2000
    }
}
```

- `try-finally` assure que le verrou est libéré même si une exception se produit.

Fonctionnalités Avancées

- **Try Lock** : Tentative non bloquante d'acquérir le verrou :

```
if (lock.tryLock()) {
    try {
        count++;
    } finally {
        lock.unlock();
    }
} else {
    System.out.println("Impossible d'acquérir le verrou");
}
```

- **Délai** : Attendre un temps limité :

```
if (lock.tryLock(1, TimeUnit.SECONDS)) { ... }
```

- **Interruptible** : Permettre aux threads en attente d'être interrompus :

```
lock.lockInterruptibly();
```

Avantages et Inconvénients

- **Avantages** : Flexible, prend en charge les fonctionnalités avancées, contrôle explicite.
- **Inconvénients** : Plus verbeux, déverrouillage manuel requis (risque d'oubli).

---

3. `Semaphore`

Un `Semaphore` contrôle l'accès à une ressource en maintenant un ensemble de permis. Il est idéal pour limiter la concurence (par exemple, max 5 threads peuvent accéder à une ressource).

Fonctionnement

- Les threads acquièrent des permis avec `acquire()`.
- Libèrent des permis avec `release()`.
- Si aucun permis n'est disponible, les threads attendent.

Exemple : Limitation des Connexions de Base de Données

```
import java.util.concurrent.Semaphore;

class ConnectionPool {
    private final Semaphore semaphore = new Semaphore(3); // Max 3 connexions

    public void connect() {
        try {
            semaphore.acquire();
            System.out.println(Thread.currentThread().getName() + " connecté");
            Thread.sleep(1000); // Simuler le travail
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            semaphore.release();
            System.out.println(Thread.currentThread().getName() + " déconnecté");
        }
    }
}

public class Main {
```

```
    public static void main(String[] args) {
        ConnectionPool pool = new ConnectionPool();
        Runnable task = () -> pool.connect();

        Thread[] threads = new Thread[10];
        for (int i = 0; i < 10; i++) {
            threads[i] = new Thread(task, "Thread-" + i);
            threads[i].start();
        }
    }
}
```

- Seulement 3 threads peuvent “se connecter”à la fois ; les autres attendent jusqu’à ce qu’un permis soit libéré.

Fonctionnalités Avancées

- **Équité** : `new Semaphore(3, true)` assure l’ordre FIFO.
- **Try Acquire** : `semaphore.tryAcquire()` vérifie sans bloquer.

Avantages et Inconvénients

- **Avantages** : Idéal pour les pools de ressources, système de permis flexible.
- **Inconvénients** : Plus complexe que `synchronized` ou `Lock`, pas pour une exclusion mutuelle simple.

Combinaison avec `ExecutorService`

Voici comment ces outils fonctionnent avec un pool de threads :

```
import java.util.concurrent.*;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class SharedResource {
    private int value = 0;
    private final Lock lock = new ReentrantLock();

    public void increment() {
        lock.lock();
```

```
        try {
            value++;
        } finally {
            lock.unlock();
        }
    }

    public int getValue() {
        return value;
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        SharedResource resource = new SharedResource();
        ExecutorService executor = Executors.newFixedThreadPool(4);

        for (int i = 0; i < 10; i++) {
            executor.submit(() -> {
                for (int j = 0; j < 100; j++) {
                    resource.increment();
                }
            });
        }

        executor.shutdown();
        executor.awaitTermination(5, TimeUnit.SECONDS);
        System.out.println("Valeur finale : " + resource.getValue()); // Toujours 1000
    }
}
```

Comparaison

Fonctionnalité	`synchronized`	`Lock` (ReentrantLock)	`Semaphore`
Facilité d'utilisation	Élevée	Moyenne	Faible
Flexibilité	Faible	Élevée	Moyenne
Granularité	Méthode/bloc	Explicite	Basé sur permis

Fonctionnalité	`synchronized`	`Lock` (ReentrantLock)	`Semaphore`
Interruptible	Non	Oui	Oui
Option d'équité	Non	Oui	Oui
Cas d'utilisation	Exclusion mutuelle simple	Contrôle avancé	Limitation de ressource

---

Quand utiliser quoi ?

- `synchronized` : Rapide, exclusion mutuelle simple (par exemple, protection d'une seule variable).
- `Lock` : Besoin de verrouillages conditionnels, délais ou interruptibilité (par exemple, flux de travail complexes).
- `Semaphore` : Limitation de l'accès concurrent à une ressource (par exemple, pools de connexions, limitation de débit).