Concurrence et parallélisme

Architecture logicielle

Concurrence != Parallélisme

- Concurrence: Gérer beaucoup de choses en même temps.
  - Gérer un grand nombre d’opérations ou demandes sur une même ressource.
- Parallélisme: Faire beaucoup de choses en même temps.
  - Découper un problème en sous-parties qui sont faisables en même temps et regrouper le résultat à la fin.

Exercice sur la concurrence

- Problème
  - La base de données dans votre projet de session se fait trop accéder, le nombre de connexions réseau et la latence associée aux recherches pourraient être optimisés.
- Solution
  - Implémentation d'une cache pour optimiser les requêtes faites à la base de données.
- Code du laboratoire
  - https://github.com/bgagnonadam/ConcurrecyTesting/archive/master.zip

Exercice sur la concurrence

- MultithreadedTC
  - https://www.cs.umd.edu/projects/PL/multithreadedtc/overview.html
  - S'intègre facilement à une suite de tests JUnit
- Trois phases à un test MtTC
  - initialize()
    - équivalent d'un @Before
  - thread#()
    - code de chaque thread à exécuter
  - finish()
    - les asserts qui sont dans le "//then" d'un test
- Synchronizer vos thread à un endroit
  - waitForTick(1);

```
class MTCBoundedBufferTest extends MultithreadedTestCase {
    ArrayBlockingQueue<Integer> buf;
    @Override public void initialize() {
        buf = new ArrayBlockingQueue<Integer>(1);
    }

    public void thread1() throws InterruptedException {
        buf.put(42);
        buf.put(17);
        assertTick(1);
    }

    public void thread2() throws InterruptedException {
        waitForTick(1);
        assertEquals(Integer.valueOf(42), buf.take());
        assertEquals(Integer.valueOf(17), buf.take());
    }

    @Override public void finish() {
        assertTrue(buf.isEmpty());
    }
}
```

Exercice sur la concurrence

- Exécuter votre test MtTC

```
public void testMTCBoundedBuffer() throws Throwable {
    TestFramework.runOnce( new MTCBoundedBufferTest() );
}
```

- Dans notre problème, nous voulons voir si la base de données sera appelée plus d'une fois lorsqu'on essaie d'obtenir les informations de la même résidence plus d'une fois en simultané

```
public RealEstate getRealEstate(String id) {
    RealEstate realEstate = realEstates.get(id);
    if (realEstate == null) {
        realEstate = realEstateRepository.findById(id);
        realEstates.put(id, realEstate);
    }
    return realEstate;
}
```

Exercice sur la concurrence

- Quelques minutes pour essayer notre cache avec MultithreadedTC.

Exercice sur la concurrence

- Le test MtTC ne passe pas…
- Solution
  - Double Check Lock!
- Attention au JIT si votre condition est déterministe
  - http://www.javaworld.com/article/2074979/java-concurrency/double-checked-locking--clever--but-broken.html

```
class SomeClass {
  private Resource resource = null;
  public Resource getResource() {
    if (resource == null) {
      synchronized {
        if (resource == null)
          resource = new Resource();
      }
    }
    return resource;
  }
}
```

Exercice sur la concurrence

- Quelques minutes pour essayer notre cache avec un DCL

- Un exemple un peu plus poussé avec une expiration sur la cache par un autre thread.

Memory barrier

Shared mutable state

Attention aux optimisations

Présentation de l’exercice

- Code : https://github.com/jni-/concurrency-lab
- Assurez-vous de comprendre le code en premier…
- Vous ne pouvez pas changer App.java, ni enlever/déplacer les simulateDelay().
- **Attention**! Effacez vos modifications entre chaque “étape” du lab (sinon, je vous garantie douleur et angoisse à essayer de débugger le résutlat…)

Exercice: synchronized method

- Ajoutez le mot-clé ‘synchronized’ à la méthode ‘transferMoneyTo()’ dans ‘Account’.
- Est-ce que ça fonctionne?
- Pourquoi?

Exercice: un lock global

- Le mot-clé ‘synchronized’ sur une méthode fait qu’on ne peut pas appelé cette méthode en même temps sur une instance de la classe. Par contre, sur 2 instances c’est possible.
- On pourrait utiliser un lock global.
- N’utilisez pas un ‘synchronized block’ sur une variable static: utilisez plutot le ReentrantLock et sa méthode .lockInterruptibly()
- Mettez le lock au début de la méthode et le unlock à la fin (c’est comme synchronized, mais partagé par toutes les instances maintenant).
- Est-ce que ça règle le problème? Pourquoi?
- Avez-vous un deadlock? Oops… pourquoi?

Exercice: synchronized localisé

- Le lock global est lent.
- Le lock sur la méthode ne fonctionne pas.
- Mais on peut avoir un lock sur une plus petite partie du code pour que ça fonctionne!
- Trouvez comment utiliser un lock (soit une méthode synchronized, soit un block synchronized, soit un ReentrantLock) afin de régler le problème.
- Attention de ne pas locké inutilement!

Exercice: les types atomic en java

- Au lieu d’un int pour la balance, utilisez un AtomicInteger.
- Est-ce que ça règle le problème? Pourquoi?
- Comment combiner cette approche aux synchronized/locks pour régler complètement le problème?

Exercice: Akka et les agents

- Essayons Akka!
- Akka est un framework d'actors principalement (on y reviendra), mais il offre aussi le concept d'agents (comme en clojure).
- Pour cette solution, vous aurez à modifier un peu App.java également afin que tout devienne asynchrone.

Exercice: Complèment FP

- En FP (functionnal programming), on essaie d’éviter complètement les Shared Mutable State.
- Comment pouvez-vous faire ça avec le problème présenté?

Actor model

Map reduce