TP n° 11 : ForkJoin, CompletableFuture, Flow

I) Problèmes récursifs avec `ForkJoin`

Exercice 1 : Tri fusion

Le code suivant présente une implémentation du tri fusion en Java :

```java
import java.util.*;

public class TriFusion {

   protected static <E extends Comparable<? super E>> List<E> fusion(List<E> l1, List<E> l2) {
      // ArrayList plutôt que LinkedList (pour get en temps constant)
      List<E> l3 = new ArrayList<>(l1.size() + l2.size());
      ListIterator<E> it1 = l1.listIterator(), it2 = l2.listIterator();
      while (it1.hasNext() || it2.hasNext())
         if (it1.hasNext()
            && (!it2.hasNext()
               || l1.get(it1.nextIndex()).compareTo(l2.get(it2.nextIndex())) < 0))
            l3.add(it1.next());
         else l3.add(it2.next());
      return l3;
   }

   public static <E extends Comparable<? super E>> List<E> triMonoThread(List<E> l) {
      if (l.size() <= 1) return l;
      else {
         int pivot = Math.floorDiv(l.size(), 2);
         List<E> l1 = triMonoThread(l.subList(0, pivot));
         List<E> l2 = triMonoThread(l.subList(pivot, l.size()));
         return fusion(l1, l2);
      }
   }

   public static void main(String[] args) {
      // doit afficher : [1, 2, 12, 81, 99, 122, 122, 234, 2134]
      System.out.println(triMonoThread(Arrays.asList(234,2134,1,122,122,2,99,12, 81)));
   }
}
```

Programmez une version concurrente de ce code à l'aide de `ForkJoinTask`.

Exercice 2 : Factorisation d'entiers

But/prétexte de l'exercice : écrire une méthode qui factorise les nombres entiers en facteurs premiers en suivant l'algorithme récursif suivant :
factorize(n) :

- entrée : n, le nombre à factoriser
- on calcule : $m = \lfloor \sqrt{n} \rfloor$ (arrondi vers l'entier en dessous de la racine carrée de n)
- on part de m et on décroit jusqu'à trouver d le premier diviseur de n inférieur ou égal à m.
- on appelle factorize(d) et factorize(n/d)
- on retourne l'union des 2 ensembles obtenus ci-dessus.

1. Implémentez cet algorithme à l'aide d'une méthode faisant des appels récursifs sur le même *thread* (faites comme si vous n'aviez jamais entendu parler de *threads*). N'utilisez pas des `int` mais des `long`.

2. Réécrivez cette méthode pour que les tâches soient des `ForkJoinTask` qu'on envoie sur un `ForkJoinPool` de taille fixée. Pour cela, implémenter une classe `Factorisation` qui **`extends`** la classe `RecursiveTask<>`.

   Testez sur des entiers pour lesquels vous savez qu'il y a beaucoup de facteurs. Testez par exemple sur l'entier `1730884069530000l` (notez le `l` : on travaille sur des **`long`**).

II) Traitements de flux de données avec `Flow`

Exercice 3 : Données épidémiologiques

La pandémie de Covid-19 faisant rage, la clé de l'organisation de la réponse tient à la bonne interprétation des données statistiques disponibles.
À cette fin, les experts ont mis au point divers indicateurs, qui souvent doivent être calculés à partir des données brutes au fur et à mesure qu'elles arrivent.
Parmi les données brutes disponibles, on peut trouver les suivantes (par jour) :

- Le nombre de tests pratiqués.
- Le nombre de nouveaux cas détectés.
- Le nombre de nouvelles hospitalisations.
- Le nombre de sorties de l'hôpital.

Les indicateurs dérivés jugés pertinents sont :

- Le nombre total de cas détectés.
- La moyenne de chacun des indicateurs bruts sur les 7 derniers jours [1].
- Le taux de reproduction effectif (le fameux R). Il s'agit du nombre de personnes contaminées par un porteur en moyenne.
  En considérant qu'il se passe 2 semaines en moyenne entre le moment où l'on est contaminé et celui où l'on contamine quelqu'un d'autre, on peut approximer R effectif en divisant le nombre de nouveaux cas du jour par celui de 14 jours avant (on peut aussi lisser comme à la question précédente).
- Le nombre de patients hospitalisés à un instant donné.
- Le taux d'incidence (c'est à dire pourcentage de postitifs parmi les tests réalisés).

À faire : écrire un simulateur, utilisant l'API `Flow`, qui imprime les indicateurs dérivés au fur et à mesure que les données brutes quotidiennes arrivent (implémentez seulement quelques exemples, puis passez à la suite du TP !).

- Les données brutes seront soumises (méthode `submit`) à une instance de `SubmissionPublisher` différente pour chaque série de données. L'alimentation de ces *publishers* se fera soit depuis des tableaux de données, soit depuis l'entrée standard.
- Les indicateurs seront calculés dans des `Flow.Processor` [2], abonnés aux flux de données brutes (méthode `subscribe`).
- L'affichage sera réalisé dans des `Flow.Subscriber` abonnés aux indicateurs.
- Tous ces objets seront instanciés, connectés entre eux (`subscribe`) et alimentés (`submit`) dans le `main` de votre programme.
- Instrumentez votre code pour que tous les appels à `onNext` affichent le nom du *thread* courant, ce qui vous permettra de constater le parallélisme effectif.

1. En effet, les statistiques rapportées le weekend sont incomplètes et généralement reportées sur le début de semaine suivante. La moyenne glissante sur 7 jours permet de gommer ces variations artificielles.
2. Vous pouvez étendre la classe `AbstractProcessor` montrée en cours, par exemple, ou bien travailler directement à partir de `SubmissionPublisher`.

Attention

- Certains indicateurs dépendent de plusieurs flux de données brutes, qu'il faudra lire de façon synchronisée depuis plusieurs `Publisher`. Pour cela, vous pouvez instancier, étendre ou composer la classe `FanInProcessor` donnée ci-dessous (ou la modifier selon vos besoins).
- `FanInProcessor` est complètement inutile si l'indicateur implémenté n'utilise qu'une seule série de données brutes ! Ne l'utilisez pas dans ce cas !

La classe `FanInProcessor` :

```
1  import java.util.concurrent.Flow.*;
2  import java.util.concurrent.SubmissionPublisher;
3  import java.util.function.*;
4
5  public class FanInProcessor<U, V, R> implements Publisher<R>, AutoCloseable {
6     public class FanInSubscriber<T> implements Subscriber<T> {
7        private final Consumer<T> store;
8        private Subscription subscription;
9
10       public FanInSubscriber(Consumer<T> store) {
11          this.store = store;
12       }
13
14       @Override
15       public void onSubscribe(Subscription subscription) {
16          this.subscription = subscription;
17          subscription.request(1);
18       }
19
20       @Override
21       public void onNext(T message) {
22          synchronized (FanInProcessor.this) {
23             store.accept(message);
24             tryProcessNext();
25          }
26       }
27
28       @Override
29       public void onError(Throwable arg0) {
30       }
31
32       @Override
33       public void onComplete() {
34       }
35
36    }
37
38    public final FanInSubscriber<U> leftInput = new FanInSubscriber<>(message -> {
39       lastLeft = message;
40    });
41
42    public final FanInSubscriber<V> rightInput = new FanInSubscriber<>(message -> {
43       lastRight = message;
44    });
45
46    private U lastLeft; // null quand dernière donnée disponible déjà  consommée
47    private V lastRight; // null quand dernière donnée disponible déjà  consommée
48    private final SubmissionPublisher<R> output = new SubmissionPublisher<>();
49    private final BiFunction<U, V, R> transform;
50
51    @Override
52    public void subscribe(Subscriber<? super R> subscriber) {
53       output.subscribe(subscriber);
54    }
55
56    @Override
57    public void close() {
```

```
58          output.close();
59      }
60
61      private void tryProcessNext() {
62          if (lastLeft == null || lastRight == null)
63              return; // la source en avance s'arrête là  et ne fait rien de particulier
64          output.submit(transform.apply(lastLeft, lastRight));
65          lastLeft = null;
66          lastRight = null;
67          leftInput.subscription.request(1);
68          rightInput.subscription.request(1);
69      }
70
71      public FanInProcessor(BiFunction<U, V, R> transform) {
72          this.transform = transform;
73      }
74
75  }
```

Exemple d'utilisation :

```
1   import java.util.concurrent.SubmissionPublisher;
2   import static java.util.stream.Stream.of;
3   import static java.util.concurrent.ForkJoinPool.commonPool;
4   import static java.util.concurrent.TimeUnit.MILLISECONDS;
5
6   public class FanInTest {
7       public static void main(String[] args) throws InterruptedException {
8           try (var source1 = new SubmissionPublisher<Integer>();
9                 var source2 = new SubmissionPublisher<Integer>();
10                var multiplicator = new FanInProcessor<Integer, Integer, Integer>((a, b) -> a *
                      b)) {
11              // 1. on construit l'application
12              source1.subscribe(multiplicator.leftInput);
13              source2.subscribe(multiplicator.rightInput);
14              multiplicator.subscribe(new PrintSubscriber<>());
15              // 2. on l'alimente avec des données
16              of(34, 1234, 123, 12, 98, 93).forEach(source1::submit); // première série de données
17              of(49, 34, 313, 132, 898, 293).forEach(source2::submit); // deuxième série de données
18              // 3. on attend que ça se termine
19              commonPool().awaitTermination(1000, MILLISECONDS);
20          } // les ressources sont fermées en sortant du try-with-resource
21      }
22  }
```

avec

```
1   import java.util.concurrent.Flow.*;
2
3   public class PrintSubscriber<T> implements Subscriber<T> {
4       private Subscription subscription;
5
6       @Override
7       public void onSubscribe(Subscription subscription) {
8           this.subscription = subscription;
9           subscription.request(1);
10      }
11
12      @Override
13      public void onNext(T message) {
14          subscription.request(1);
15          System.out.println(message);
16      }
17
18      @Override
19      public void onError(Throwable arg0) {
20      }
21
22      @Override
```

```
23      public void onComplete() {
24      }
25  }
```

III) Calculs concurrents simples avec `CompletableFuture`

Exercice 4 :

Ci-dessous, un certain nombre de calculs arithmétiques :

1. 12
2. $2 * 5$
3. $(3 + 2) * 9$
4. $(x + 5)$, où x est lu dans l'entrée standard dans une tâche concurrente
5. $(3 + 2) * (3 + 2)$, sans recalculer le terme qui se répète.

Écrivez ces calculs sous forme de tâches successives décrites par des `CompletableFuture`, avec pour règle : effectuer une seule opération par tâche.
Exemple pour $6 + 7$:

```
1       var e1 = completedFuture(6);
2       var e2 = completedFuture(7);
3       var e3 = e1.thenCombine(e2, (a,b) -> a + b);
4       System.out.println(e3.join()); // affiche 13
```

(Ne pas oublier le `import static java.util.concurrent.CompletableFuture.*;` initial !)
Faites d'abord en sorte que le résultat soit imprimé dans `main`, puis modifiez ensuite vos programmes pour que le `println` soit aussi exécuté dans une tâche concurrente.
Enfin ajoutez `System.out.printn(Thread.currentThread().getName());` dans chaque tâche concurrente pour savoir dans quel *thread* elle s'exécute.

Conseil : n'hésitez pas à vous référer à la documentation de `CompletableFuture`[3].

3. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/CompletableFuture.html