Solutions aux exercices du chapitre 10 : Programmation Java

Automne 2017

Exercice 10.1 Méthode `valeur` : `synchronized` ou pas?

Est-il nécessaire ou approprié que la méthode `valeur` soit aussi `synchronized`?

Solution :

Sur la base de ce qu'on a vu jusqu'à présent, non, ce n'est ni nécessaire, ni utile : l'attribut `val` étant un `int`, sa lecture est une opération atomique. L'ajout de `synchronized` n'aurait donc aucun effet en termes d'atomicité.

Par contre, on verra plus loin... que si `synchronized` n'est pas utilisé, le résultat pourrait ne pas être correct — *à cause du modèle de mémoire de Java.*

Exercice 10.2 Différences entre deux classes `Compteur`?

Est-ce que le comportement des deux classes suivantes diffère? Si oui de quelle façon?

```
class Compteur {
  private int val1 = 0, val2 = 0;

  public synchronized void inc1() {
    val1 += 1;
  }

  public synchronized void inc2() {
    val2 += 1;
  }
}

class Compteur {
  private int val1 = 0, val2 = 0;

  private Object l1 = new Object(),
                 l2 = new Object();

  public void inc1() {
    synchronized(l1) { val1 += 1; }
  }

  public void inc2() {
    synchronized(l2) { val2 += 1; }
  }
}
```

Solution :

Oui, il y a une différence.

- Dans le premier cas, les appels à `inc1` et `inc2` sont mutuellement exclusifs, *donc ne peuvent s'exécuter en parallèle,* puisque le verrou utilisé est le même, i.e., le verrou implicite associé à l'objet.
- Dans le deuxième cas, *seuls des appels concurrents de la même méthode sont mutuellement exclusifs*, puisque deux verrous distincts sont utilisés.

Exercice 10.3 Que fait un sémaphore?

1. Que fait un sémaphore?
2. Que font les opérations `P()` et `V()`?

Solution :

1. Variable spéciale qui contient un nombre entier *non-négatif* et qui est manipulée par deux opérations atomiques : `P` et `V`.

   Les sémaphores peuvent jouer différents rôles (voir plus bas) :

   - Exclusion mutuelle
   - Synchronisation conditionnelle
   - Gestion de ressources
   - etc.

   ≈ *goto*s de la programmation concurrente

2. (a) `P` = *Passeren* ≈ passer, prendre ⇒ Décrémente la variable (à moins qu'elle ne soit déjà 0).

   Utilisée (lorsque déjà 0) pour retarder (suspendre) un processus jusqu'à ce qu'un événement survienne.

   (b) `V` = *Vrijgeven* ≈ relâcher ⇒ Incrémente la variable.

   Utilisée pour *signaler* un événement et, possiblement, réactiver un processus en attente.

Exemples de patrons d'utilisation :

- Exclusion mutuelle (verrou) :

```
Semaphore mutex = new Semaphore( 1 );

...
  mutex.P();
  ... section critique ...
  mutex.V();
...
```

- Synchronisation conditionnelle :

```
T buf;
Semaphore empty = new Semaphore( 1 );
Semaphore full  = new Semaphore( 0 );

// Thread pour producteur.
```

```
  while(true) {
    T data = ...  // Produire item dans data.
    empty.P();    // Attend que buf soit vide.
    buf = data;
    full.V();     // Signale que buf plein.
  }


// Thread pour consommateur
  while(true) {
    full.P();       // Attend que buf est plein.
    T result = buf;
    empty.V();      // Signale que buf est vide.
    ... traiter result ...
  }
```

Exercice 10.4 Méthode `valeur` : `synchronized` ou pas?

Est-il nécessaire ou approprié que la méthode `valeur` ci-bas soit `synchronized`?

```
class Compteur {
  private int val = 0;

  public synchronized void inc() {
    val += 1;
  }

  public int valeur() {
    return val;
  }
}
```

Solution :

Non, pas strictement nécessaire, car on pourrait plutôt indiquer que l'attribut `val` est `volatile`.

Exercice 10.5 Exemple spécial illustrant le modèle de mémoire de Java.

Qu'est-ce qui sera imprimé par le programme ci-bas?

```
class MemoryModel {
  static int x = 0,
             y = 0,
             r1,
             r2;

  public static void main( String[] args ) {
    Thread t1 = new Thread( () -> {
        r1 = x;  // I1
        y = 1;   // I2
      } );
    Thread t2 = new Thread( () -> {
        r2 = y;  // I3
        x = 2;   // I4
      } );
    t1.start(); t2.start();

    try {
      t1.join(); t2.join();
    } catch( Exception e ){}

    System.out.println( "r1 = " + r1 +
                        "; " +
                        "r2 = " + r2 );
  }
}
```

Solution :

Voici les diverses possibilités, avec les entrelacements des quatre instructions `I1`, ..., `I4` qui conduisent à ces résultats :

- `I1; I2; I3; I4` :

  `r1 = 0; r2 = 1`

- `I1; I3; I4; I2` :

  `r1 = 0; r2 = 0`

- `I1; I3; I2; I4` :

  `r1 = 0; r2 = 0`

- `I3; I4; I1; I2` :

  `r1 = 2; r2 = 0`

- `I3; I1; I2; I4` :

  `r1 = 0; r2 = 0`

- `I3; I1; I4; I2` :

  `r1 = 0; r2 = 0`

Toutefois, même le résultat suivant est possible (sic!), car à l'intérieur d'un *thread*, le compilateur «a le droit» de réordonner les instructions *si cela respecte les dépendances locales* — donc il pourrait réordonner les deux instructions du *thread* `t1`, ou les deux instructions du *thread* `t2`, etc. :

- `I2; I3; I4; I1` :

  `r1 = 2; r2 = 1`

- `I4; I1; I2; I3` :

  `r1 = 2; r2 = 1`

- ...

  To some programmers, this behavior may seem "broken". However, it should be noted that this code is improperly synchronized*:*

  - *there is a write in one thread,*
  - *a read of the same variable by another thread,*

- *and the write and read are not ordered by synchronization.*

This situation is an example of a data race. When code contains a data race, counterintuitive results are often possible.

Source : `https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html`

Exercice 10.6 Interface `Lock` et classes associées.

Pourquoi la bibliothèque `java.util.concurrent` définit-elle une interface `Lock` ainsi que des classes qui mettent en oeuvre cette interface — `ReentrantLock` et `ReadWriteReentrantLock` — alors que des verrous sont déjà disponibles avec `synchronized` et `synchronized(this){...}`?

Solution :

- Les verrous implicites ne peuvent être utilisés que de façon structurée (*block-structured*) :

  ```
  synchronized(o) {
    ...
  }
  ```

  Or, pour certains problèmes, cela ne suffit pas.

- Il est impossible de vérifier si un verrou implicite est disponible ou non avant de tenter de l'acquérir. Par contre, pour le type `Lock`, on peut utiliser l'opération `tryLock()!`

- Un `ReadWriteLock` a un comportement différent selon le type d'accès requis : soit un (et un seul) «écrivain» actif, soit *plusieurs* «lecteurs» actifs!

- *Performances :* Voir plus loin.

Exercice 10.7 Appels multiples à `synchronized`.

Que se passe-t-il si un *thread* exécute une méthode `synchronized` puis tente d'appeler une autre méthode elle aussi `synchronized` de la même classe?

Solution :

Rien de particulier, en ce que le *thread* «réobtient» normalement le verrou — un compte du nombre de fois où le verrou a été acquis est incrémenté, puis décrémenté lors de la libération. Le verrou redevient donc effectivement libre uniquement lorsque le compte retombe à 0.

Donc, on dit que les verrous implicites sont des verrous «réentrants», i.e., peuvent être acquis à nouveau par un *thread* qui a déjà ce verrou en sa possession.

Puisque les acquisitions et libérations sont toujours balancées — accès structuré — il n'y a donc jamais de danger «d'oublier» une libération de verrou.

Exercice 10.8 Différences entre appels à `acquire`?

Est-ce que «`acquire(2);`» a le même effet que «`acquire(); acquire()`»?

Solution :

Non, pas le même effet car dans le deuxième cas les deux acquisitions ne se feront pas de façon atomique.

Exercice 10.9 Valeur possible pour un `AtomicInteger`

Quelle est la valeur — ou *les valeurs* — qui peut être retournée par le `ai.get()` dans le *thread* principal (dernière ligne du segment de code)?

Solution :

- 19 si un *thread* exécute `ai.get()` et `ai.compareAndSet()` avant que l'autre n'exécute `ai.get()`.
- 18 si les deux *threads* exécutent `ai.get()` avant que l'un des *threads* exécute `ai.compareAndSet()`.

Exercice 10.10 Méthode `compareAndSet` et son utilisation pour la mise en oeuvre d'une méthode `incrementer`.

Dans la méthode `increment`, quand l'appel à `compareAndSet` peut-il retourner `false` — et donc quand le corps de la boucle `while` sera-t-il exécuté à nouveau?

Solution :

Quand un autre *thread* a, lui aussi, obtenu la valeur courante, l'a incrémentée, puis tente de faire la mise à jour.

Exercice 10.11 Méthodes `foo_tranche` et `foo_pr`.

Soit le segment de code suivant :

```
int n = 1000;
double[] a = new double[n];
for( int i = 0; i < n; i++ ) {
  a[i] = 1.0;
}

ExecutorService pool = ?          ?;

double r =
   foo_pr(a, 0, a.length-1, 100, pool);
System.out.println( r );
```

Qu'est-ce qui sera imprimé par le segment de code ci-haut selon les différentes valeurs possibles suivantes pour `pool` :

1. `Executors.newCachedThreadPool()`
2. `new ForkJoinPool(4)`
3. `Executors.newFixedThreadPool(4)`

Solution :

1. 1000.0
2. 1000.0
3. Rien! Il y a un *deadlock* ☹

   Dès que suffisamment de tâches ont été soumises pour exécution et que l'on aura les 4 *threads* qui les traitent bloquées sur le `gauche.get()`, l'exécution deviendra bloquée, car le *pool* n'utilise qu'exactement 4 *threads*.

Exercice 10.12 Méthodes `foo_tranche` et `foo_pr` (bis).

Soit le segment de code suivant :

```
int n = 100000;
double[] a = new double[n];
for( int i = 0; i < n; i++ ) {
  a[i] = 1.0;
}

ExecutorService pool = ?          ?;

double r =
   foo_pr(a, 0, a.length-1, 2, pool);
System.out.println( r );
```

Qu'est-ce qui sera imprimé par le segment de code ci-haut selon les différentes valeurs possibles suivantes pour `pool` :

1. `Executors.newCachedThreadPool()`
2. `new ForkJoinPool(4)`

Solution :

1. ```
   Exception in thread "main" java.lang.OutOfMemoryError:
           unable to create new native thread
           at java.lang.Thread.start0(Native Method)
           at java.lang.Thread.start(Thread.java:714)
           at java.util.concurrent.ThreadPoolExecutor.addWorker(ThreadPoolExecutor.java:950
           at java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:1368)
           at java.util.concurrent.AbstractExecutorService.submit(AbstractExecutorService.j
           at Sommation.sommation_pr(Sommation.java:25)
           at Sommation.sommation_pr(Sommation.java:26)
           at Sommation.sommation_pr(Sommation.java:26)
           at Sommation.sommation_pr(Sommation.java:26)
           at Sommation.sommation_pr(Sommation.java:26)
           at Sommation.sommation_pr(Sommation.java:26)
           at Sommation.sommation_pr(Sommation.java:26)
           at Sommation.sommation_pr(Sommation.java:26)
           at Sommation.sommation_pr(Sommation.java:26)
           at Sommation.sommation_pr(Sommation.java:26)
           at Sommation.main(Sommation.java:64)
   ```

   Le trop grand nombre de tâches (à cause du `seuil` = 2) et l'utilisation d'un `newCachedThreadPool` génère un trop grand nombre de *threads*.

2. 100000.0

   Parce qu'avec un `ForkJoinPool`, le nombre de *threads* n'augmente pas et un *thread* qui bloque sur `get()` ne bloque pas l'exécution de l'ensemble du programme.

Remarque : Sur ma machine Linux (Cent OS) à huit (8) coeurs, si on exécute les deux versions ci-haut avec `n`=1000000, `seuil`=100 et avec 8 *threads*, on obtient les temps d'exécution suivants (sic) :

1. Version avec `newCachedThreadPool()` :

   ```
   real 0.55
   user 1.30
   sys 0.62
   ```

2. Version avec `new ForkJoinPool(8)` :
   ```

```
real 1.10
user 6.70
sys 0.30
```

Et sur la machine `japet` :

1. Version avec `newCachedThreadPool()` :

   ```
   real 2.12
   user 27.28
   sys 6.38
   ```

2. Version avec `new ForkJoinPool(8)` :

   ```
   real 2.96
   user 96.25
   sys 4.44
   ```

Une autre solution avec un `ForkJoinPool` mais beaucoup plus efficace est celle présentée ci-bas. Cette solution utilise des `RecursiveTask`. Pour les mêmes paramètres, on obtient alors le temps d'exécution suivant (au lieu de 1.10, donc 15 fois plus rapide!):

```
real 0.07
user 0.16
sys 0.01
```

Donc, si on veut utiliser efficacement un `ForkJoinPool`, il est préférable d'éviter le `submit` de base et d'utiliser plutôt les `RecursiveTask`.

Exercice 10.14 Effets des optimisations sur les accélérations.

Supposons un programme ayant le comportement suivant, où `S` indique une partie du programme qui ne peut s'exécuter que de façon *strictement séquentielle* et où `P` indique une partie qui peut s'exécuter de façon parallèle si des processeurs multiples sont disponibles :

```
S-S-P-P-P-P-P-P-P-P-S-S
```

Pour simplifier l'analyse qui suit, on suppose que *i*) chaque `S` ou `P` requiert le même temps et *ii*) les parties `P` peuvent être parallélisées de façon complète et aussi fine que nécessaire (*embarrassingly parallel*).†

1. Déterminez quelle sera l'accélération pour divers nombres de processeurs?
   (a) Quelle sera l'accélération pour 2 processeurs?
   (b) Quelle sera l'accélération pour 4 processeurs?
   (c) Quelle sera l'accélération pour 8 processeurs?
   (d) Quelle sera l'accélération pour 16 processeurs?
   (e) Quelle sera la *meilleure accélération* possible pour ce programme?

2. Supposons qu'on ait réussit à *optimiser* le programme initial et à réduire le temps d'exécution, *mais uniquement de la partie parallèle* et qu'on obtienne le comportement suivant :

   ```
   S-S-P-P-P-P-P-P-S-S
   ```

   (a) Quelle sera l'accélération pour 2 processeurs?
   (b) Quelle sera l'accélération pour 4 processeurs?
   (c) Quelle sera l'accélération pour 8 processeurs?
   (d) Quelle sera l'accélération pour 16 processeurs?
   (e) Quelle sera la *meilleure accélération* possible pour ce programme?

† Notez qu'une telle allure pour un programme parallèle est assez typique : le programme débute par une partie séquentielle (par ex., lecture des paramètres et données et initialisation), suivie d'une partie parallélisable, puis terminée par une partie séquentielle (par ex., combinaison des résultats intermédiaires ou écriture des résultats finaux).

Solution :

1. Accélération (absolue) du programme initial, non optimisé, pour divers nombres de processeurs :

   Temps séquentiel = 2 + 8 + 2 = 12.00

   (a) $T_P(2) = \dfrac{12}{2 + \frac{8}{2} + 2} = \dfrac{12}{8} = 1.50$

   (b) $T_P(4) = \dfrac{12}{2 + \frac{8}{4} + 2} = \dfrac{12}{6} = 2.00$

   (c) $T_P(8) = \dfrac{12}{2 + \frac{8}{8} + 2} = \dfrac{12}{5} = 2.40$

   (d) $T_P(16) = \dfrac{12}{2 + \frac{8}{16} + 2} = \dfrac{12}{4.5} = 2.67$

   (e) $T_P = T_P(+\infty) = \dfrac{12}{2 + \frac{8}{+\infty} + 2} = \dfrac{12}{4.0} = 3.00$

   On constate que les parties `S` représentent $\frac{1}{3}$ (4`S` vs. 8`P`) du temps d'exécution séquentiel... et que l'accélération maximale est de 3.0!

2. Accélération (absolue) du programme optimisé, pour divers nombres de processeurs :

   Temps séquentiel = 2 + 6 + 2 = 10.00

   (a) $T_P(2) = \dfrac{10}{2 + \frac{6}{2} + 2} = \dfrac{10}{7} = 1.43$

   (b) $T_P(4) = \dfrac{10}{2 + \frac{6}{4} + 2} = \dfrac{10}{5.5} = 1.81$

   (c) $T_P(8) = \dfrac{10}{2 + \frac{6}{8} + 2} = \dfrac{10}{4.75} = 2.11$

   (d) $T_P(16) = \dfrac{10}{2 + \frac{6}{16} + 2} = \dfrac{10}{4.375} = 2.29$

   (e) $T_P = T_P(+\infty) = \dfrac{10}{2 + \frac{6}{+\infty} + 2} = \dfrac{10}{4.0} = 2.50$

   On constate que les parties `S` représentent $\frac{4}{10}$ (4`S` vs. 6`P`) du temps d'exécution séquentiel... et que l'accélération maximale est de 2.5!