Théorie et pratique de la concurrence — Master 1 Informatique

TP 1 : Programmation concurrente en Java

1 En Java

1.1 Rappel création de *thread*

Les *threads* Java est un processus léger, c'est-à-dire un processus qui a sa pile d'exécution, son contexte d'exécution, et accès direct aux variables de sa portée.

Pour programmer avec des *threads*, il faut suivre les étapes suivantes :

1. écrire une nouvelle classe qui étend la classe `Thread` ou qui implémente l'interface `Runnable` ;
2. dans cette nouvelle classe, définir une méthode `run` qui contient le code à exécuter par le *thread* ;
3. créer un objet de cette classe avec `new` ;
4. lancer le *thread* en appelant la méthode `start`.

1.2 Synchronisation de *thread*

L'exclusion mutuelle en Java est fournie par le mot clé `synchronized`, qui peut être utilisé comme attribut d'une méthode où d'une séquence d'instructions. Par exemple, si on a une donnée partagée `data` dans une classe `Interfere`, sa mise à jour en exclusion mutuelle peut être faite de deux façons :

1. exclusion mutuelle sur la méthode de mise à jour :

```
class Interfere {
  private int data = 0;
  public synchronized void update () {
    data++;
  }
}
```

   Cette façon de faire montre comment implémenter les moniteurs en Java : les variables abstraite du moniteur sont des variables privées de la classe et les procédures du moniteur sont des méthodes `synchronized`.

2. exclusion mutuelle sur une séquence d'instructions :

```
class Interfere {
  private int data = 0;
  public void update () {
    synchronized (this) { data++; }
  }
}
```

   Cette façon est similaire aux instructions `atomic`

De plus, la classe `Object` déclare deux méthodes pour la synchronisation : `wait` et `notify`. Ces deux méthodes doivent être appelées que dans une portion de code `synchronized`, donc quand l'objet est utilisé en exclusion mutuelle.

La méthode `wait` bloque le processus appelant dans la file d'attente de l'objet courant et libère cet objet. Java ne fournit pas des variables conditionnelles. Toutefois, on peut utiliser la file d'attente d'un objet `synchronized` comme variable conditionnelle implicite (une seule par objet).

La méthode `notify` libère un processus bloqué dans la file d'attente de l'objet. Le processus qui l'appelle continue d'avoir l'accès exclusif à l'objet, donc le processus libéré sera exécuté quand il pourra acquérir l'objet. Il s'agit donc d'une sémantique *Signale et continue*. Java fournit également une méthode `notifyAll` qui libère tous les processus bloqués dans la file d'attente.

Toutes les méthodes ci-dessus ont une liste vide de paramètres.

Un processus qui exécute un code en exclusion mutuelle peut appeler une autre méthode `synchronized`. Si cette méthode appartient à un autre objet, l'exclusion mutuelle est maintenue pour le premier objet durant l'appel. Ceci peut provoquer des inter-blocages si une méthode `synchronized` d'un objet O1 appelle une méthode `synchronized` d'un autre objet O2 et vice-versa.

1.3 Exercices

Dans les exercices ci-dessous on se propose à programmer en Java le problème des lecteurs/écrivains. Pour chacun des exercices, essayer d'observer le comportement du programme en obtenant des ensemble d'exécution différentes (une astuce peut considérer à endormir des threads pendant un moment grâce à la méthode statique `sleep` de la classe `Thread`).

Exercice 1:

Dans une première version, on s'intéresse qu'à la concurrence, sans synchronisation (exclusion mutuelle).

1. Écrire une classe `RWbasic` qui encapsule une donnée entière `data` auquel l'accès est fait (sans exclusion mutuelle) par deux méthodes :
   (a) `read` qui renvoie la valeur de la donnée, et
   (b) `write` qui incrémente la valeur de la donnée.
2. Écrire ensuite une classe `Reader` qui étend `Thread`. Le constructeur des objets de cette classe reçoit comme arguments le nombre de lectures à effectuer (`nr`) et une référence vers un objet de classe `RWbasic`. Dans son code exécutable (méthode `run`), il appelle `nr` fois la méthode `read` ci-dessus.
3. Écrire de façon similaire une classe `Writer`.
4. Écrire une classe `Main` qui crée un objet de classe `RWbasic` et puis crée et lance deux *threads*, un `Reader` et un `Writer`.

Observer le comportement de votre programme.

Exercice 2:

Dans une deuxième version, on s'intéresse également à la synchronisation.

1. Écrire une classe `RWexclusive` qui étend `RWbasic` en rendant exclusif l'accès à la donnée.
2. Modifier le code déjà écrit pour les classes `Reader`, `Writer` et `Main` pour qu'elles utilise la classe `RWexclusive` à la place de `RWbasic`.

Observer le comportement de votre programme.

Exercice 3:

La troisième version implémente une solution correcte au problème des lecteurs et écrivains : plusieurs lecteurs peuvent lire à la fois mais un seul écrivain peut accéder à la donnée.

1. Écrire une classe `ReadersWriters` qui étend `RWbasic` comme suit :
   - le nombre de lecteurs est comptabilisé par une compteur privé ;
   - une méthode privée `synchronized startRead` contrôle l'accès des lecteurs à la donnée ;
   - une méthode privée `synchronized endRead` contrôle la fin de la lecture par un lecteur ; si le nombre de lecteurs est nul, elle libère un processus ;
   - la méthode publique `read` appelle `startRead`, lit la donnée, puis appelle `endRead` ;
   - la méthode publique `synchronized write` attend que le nombre de lecteur soit nul avant d'écrire la donné.
2. Modifier le code déjà écrit pour les classes `Reader`, `Writer` et `Main` pour qu'elles utilise la classe `ReadersWriters` à la place de `RWexclusive`.

Exercice 4:

La technique de synchronisation, dite de *la barrière*, consiste à synchroniser un ensemble de processus en un point de programme : les processus sont bloqués (endormis) en un point de programme jusqu'à ce que tous les processus soient à ce point de programme. À ce moment-là, les processus bloqués ont l'autorisation de continuer leur exécution.

On désire étendre le programme lecteurs/écrivains de l'exercice précédent de la façon suivante. Toute donnée produite par un écrivain doit être lue par au moins un lecteur. Pour cela, la classe encapsulant la donnée se sert d'un champ `disponible` qui passe à `true` quand un lecteur a lu la donnée courante et est mis à `false` lorsqu'un écrivain produit une nouvelle donnée (un écrivain ne pouvant pas produire une nouvelle donnée si `disponible` est à `false`). De plus, on assumera que les écrivains produisent chacun à leur tour une nouvelle donnée, pour garantir ce point les écrivains se synchroniseront sur une barrière après avoir écrit une donnée.

1. Modifier la classe `ReadersWriters` pour y inclure la donnée booléenne `disponible` et modifier les méthodes de cette classe pour assurer que tout donnée produite sera lue.
2. Construire une classe `Barriere` ayant une unique méthode `synchronisation` qui bloque jusqu'à $k-1$ threads (k étant un champ de la classe) et lorsque le k-ème thread rentre dans la méthode `synchronisation`, les k threads sont autorisés à sortir de la méthode et à poursuivre leur exécution..
3. Modifier les classes de l'exercice précédent pour obtenir un programme avec 2 lecteurs et 2 écrivains qui a le comportement désiré.