INF5171: Laboratoire #6
Mise en oeuvre de moniteurs en Ruby

2 novembre 2017
13h30–15h30
PK-S1565

Le but de ce laboratoire est de vous familiariser avec l'utilisation des variables de condition et la mise en oeuvre de moniteurs.

---

1 Mise en oeuvre de verrous réentrants (Exercice 6.12)

Pour cet exercice, obtenez le code source à compléter à l'aide de la commande suivante :

```
$ git clone http://www.labunix.uqam.ca/~tremblay/git/ReentrantLock.git
```

Ce répertoire contient quatre fichiers, mais un seul que vous devez compléter :

- `reentrant_lock.rb` : Contient le squelette d'une classe que vous devez compléter pour mettre en oeuvre des *verrous réentrants*.

  Il s'agit d'une classe semblable à celle disponible dans la bibliothèque `java.util.concurrent`. Consultez la page suivante pour connaitre le comportement attendu de chacune des opérations : `http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/locks/ReentrantLock.html`

Indices et suggestions :

- La classe `Thread` possède une méthode de classe, nommée `current`, qui retourne **le *thread* courant** — donc celui qui exécute l'appel à cette méthode.
- Les variables d'instance de la classe `ReentrantLock`, telles que spécifiées dans la méthode `initialize`, utilisent les noms anglais implicites dans les `attr_reader`s, i.e., `@hold_count` et `@owner`.

2 Mise en oeuvre d'un `Reducteur` (Exercice 6.15)

Pour cet exercice, obtenez le code source à compléter à l'aide de la commande suivante :

```
$ git clone http://www.labunix.uqam.ca/~tremblay/git/Reducteur.git
```

Vous devez compléter le fichier `reducteur.rb`, tel que décrit ci-bas.

On veut définir un objet `Reducteur` qui permet à un groupe de *threads* d'effectuer une réduction, spécifiée par une valeur initiale, un nombre de *threads* et un bloc :

- Lorsqu'un *thread* appelle `réduire`, le code du `bloc` passé lors de la création du `Reducteur` est évalué en lui passant l'argument reçu via l'appel à `reduire`.
- Tant que tous les `nb_threads` *threads* n'ont pas effectué leur appel à `reduire`, **aucun résultat n'est retourné**. Donc, les `nb_threads-1` premiers *threads* qui appellent `reduire` vont **bloquer**, en attente du résultat.
- Lorsque tous les `nb_threads` *threads* ont effectué leur appel à `reduire`, le résultat (combinaison par le bloc des diverses valeurs reçues) est retourné à chacun des *threads*, lesquels poursuivent ensuite leur exécution.

```
class Reducteur
  def initialize( val_initiale, nb_threads, &bloc )
    ...
  end

  def reduire( valeur )
    ...
  end
end
```

Votre moniteur doit être **réutilisable** — c'est-à-dire qu'on doit pouvoir effectuer plusieurs réductions, l'une après l'autre. Pour simplifier, vous pouvez supposer que les séries d'appels **ne se chevauchent pas** — donct tous les *threads* auront obtenu la valeur retournée par `reduire` **avant** qu'une nouvelle série d'appels soit faite.

Voici un exemple d'utilisation (extrait d'un cas de test) :

```
c = Reducteur.new( 100, 10 ) { |x, y| x + y }
ts = (1..10).map { |k| Thread.new { c.reduire(k) } }
ts.each { |t| t.value.must_equal 155 }
```

Indice :

- Voir les autres cas de test, dans `reducteur_spec.rb`, pour d'autres exemples d'utilisation.
- Voir la classe `Barriere` vue en cours (Section 6.4.3 du chapitre «Programmation concurrente avec *threads* en Ruby»).