Solutions aux exercices du chapitre 6 : Programmation concurrente Ruby

Automne 2017

Exercice 6.1 Un petit programme qui utilise une `ConditionVariable`.

Qu'est-ce qui sera imprimé par le segment de code ci-bas.

```
mutex = Mutex.new
cond = ConditionVariable.new

Thread
  .new { cond.wait(mutex) }
  .join

puts "FIN"
```

Solution :

```
ThreadError: Mutex is not locked
  unlock at org/jruby/ext/thread/Mutex.java:106
    wait at org/jruby/ext/thread/ConditionVariable.java:94
  (root) at condition_variable.rb:5
```

Parce qu'un appel à `wait` est fait avec `mutex` *sans que le* thread *ne soit propriétaire du verrou.*

Exercice 6.2 Un petit programme qui utilise une `ConditionVariable`.

Qu'est-ce qui sera imprimé par le segment de code ci-bas.

```
mutex = Mutex.new
cond = ConditionVariable.new

t = Thread.new do
  sleep 1
  mutex.synchronize do
    cond.signal
    puts "Apres signal"
  end
end

mutex.synchronize do
  cond.wait(mutex)
  puts "Apres wait"
end

t.join
puts "FIN"
```

Solution :

```
Apres signal
Apres wait
FIN
```

Exercice 6.3 Un petit programme qui utilise une `ConditionVariable`.

Qu'est-ce qui sera imprimé par le segment de code ci-bas.

```
mutex = Mutex.new
cond = ConditionVariable.new

t = Thread.new do
  sleep 1
  mutex.synchronize do
   cond.wait(mutex)
   puts "Apres wait"
  end
end

mutex.synchronize do
  cond.signal
  puts "Apres signal"
end

t.join
puts "FIN"
```

Solution :

```
Apres signal
```

Parce qu'un le *thread* enfant s'endort, le *thread* maître envoie le signal, mais aucun *thread* ne l'attend, puis libère le verrou et bloque sur le `join`. Le *thread* enfant acquiert le verrou, puis attend un signal... qui ne vient jamais car il n'y a plus d'autres *threads* actifs. Donc, *deadlock*!

Exercice 6.4 Un petit programme qui utilise une `ConditionVariable`.

Qu'est-ce qui sera imprimé par le segment de code ci-bas.

```
mutex = Mutex.new
cond = ConditionVariable.new

t = Thread.new do
  mutex.synchronize do
    cond.wait(mutex)
    mutex.lock
    puts "Apres wait"
  end
end

sleep 1
mutex.synchronize do
  cond.signal
  puts "Apres signal"
end

t.join
puts "FIN"
```

Solution :

Avec `jruby` :

```
Apres signal
ThreadError: Mutex relocking by same thread
         lock at org/jruby/ext/thread/Mutex.java:90
       (root) at condition_variable.rb:8
  synchronize at org/jruby/ext/thread/Mutex.java:149
       (root) at condition_variable.rb:6
```

Avec `ruby/MRI` :

```
Apres signal
condition_variable.rb:8:in 'lock': deadlock;
                           recursive locking (ThreadError)
    from condition_variable.rb:8:in 'block (2 levels) in <main>'
    from condition_variable.rb:6:in 'synchronize'
    from condition_variable.rb:6:in 'block in <main>'
```

Avec les verrous ordinaires de Ruby, un *thread* ne peut pas tenter d'acquérir un verrou qu'il possède déjà. Les verrous de base *ne sont donc pas réentrants*.

Exercice 6.5 Un moniteur pour un `Accumulateur`.

Soit la classe `Accumulateur` ci-bas.
Qu'est-ce qui sera imprimé par le segment de code suivant :

```
r = nil
acc = Accumulateur.new( 0 )

[Thread.new { acc.ajouter( 10 ) },
 Thread.new { acc.ajouter( 20 ) },
 Thread.new { acc.ajouter( 30 ) },
 Thread.new { r = acc.valeur_si { |x| x >= 30 } }
].map(&:join)

puts r
```

```
class Accumulateur
  def initialize( valeur_initiale = 0 )
    @valeur = valeur_initiale
    @mutex = Mutex.new
    @condition = ConditionVariable.new
  end

  def ajouter( k )
    @mutex.synchronize do
      @valeur += k
    end
  end

  def valeur_si
    @mutex.synchronize do
      @condition.wait(@mutex) until yield(@valeur)

      @valeur
    end
  end
end
```

Solution :

Pourrait imprimer 30, 40, 50, 60... ou

```
accumulateur.rb:37:in 'join':
     No live threads left. Deadlock? (fatal)
        from accumulateur.rb:37:in 'map'
        from accumulateur.rb:37:in '<main>'
```

Exercice 6.6 Un moniteur pour un `Accumulateur` (bis).

Soit la classe `Accumulateur` ci-bas.
Qu'est-ce qui sera imprimé par le segment de code suivant :

```
r = nil
acc = Accumulateur.new( 0 )

[Thread.new { acc.ajouter( 10 ) },
 Thread.new { acc.ajouter( 20 ) },
 Thread.new { acc.ajouter( 30 ) },
 Thread.new { r = acc.valeur_si { |x| x >= 30 } }
].map(&:join)

puts r
```

```
class Accumulateur
  def initialize( valeur_initiale = 0 )
    @valeur = valeur_initiale
    @mutex = Mutex.new
    @condition = ConditionVariable.new
  end

  def ajouter( k )
    @mutex.synchronize do
      @valeur += k
      @condition.signal
    end
  end

  def valeur_si
    @mutex.synchronize do
      @condition.wait(@mutex) until yield(@valeur)

      @valeur
    end
  end
end
```

Solution :

Pourrait imprimer 30, 40, 50 ou 60.

Exercice 6.7 Programme qui utilise `TamponBorne`.

1. Qu'est-ce qui sera imprimé par la méthode de test `test_TamponBorne1`?
2. À quel patron de programmation parallèle est-ce que cela correspond?
3. Quel est l'effet de mettre la valeur `1` lors de la création de `buf2`?

Solution :

1. `[0, 2, 4, 6, 8]`
2. Parallélisme de flux avec filtres et pipeline : chaque `TamponBorne` sert de «canal de communication» entre deux *threads*!
3. Cela empêche la concurrence entre le deuxième et le troisième *thread*, donc il y a alternance stricte entre les deux *threads* ☹

Exercice 6.8 Mise en oeuvre de `TamponBorne`.

Que se passe-t-il si, dans `ajouter` ou `retirer`, on remplace le `while` par un `if`?

```
def ajouter( data )
  @mutex.synchronize do
    @pas_plein.wait(@mutex) if @nb_elems == @taille
    @tampon[@queue] = data
    @queue = (@queue + 1) % @taille
    @nb_elems += 1
    @pas_vide.signal
  end
end

def retirer
  result = nil

  @mutex.synchronize do
    @pas_vide.wait(@mutex) if @nb_elems == 0
    result = @tampon[@tete]
    @tete = (@tete + 1) % @taille
    @nb_elems -= 1
    @pas_plein.signal
  end

  result
end
```

Solution :

Le résultat pourrait être incorrect, à cause d'une *situation de compétition*.

Par exemple, un *thread* `t0` appelle `retirer` et bloque parce que le tampon est vide.

Un *thread* `t1` fait un appel `ajouter(99)`, ce qui réactive `t0` à cause du `signal`.

Le *thread* `t1` sort du `wait`, tente d'obtenir le verrou, mais c'est plutôt un autre *thread* `t2`, faisant lui aussi un appel à `retirer`, qui acquière le verrou, et donc qui obtient 99! Le verrou est ensuite libéré par `t2`.

Le *thread* `t0` obtient alors le verrou... sauf que le tampon est vide à nouveau ☹

De façon générale, il faut utiliser un `while` pour assurer que l'état de la condition **n'a pas changé entre le moment où le signal a été émis et le moment où le *thread* réactivé obtient effectivement le verrou**.

Exercice 6.9 Un autre programme qui utilise `TamponBorne`.

```
def test_TamponBorne2
  nb_cons = 3

  buf = TamponBorne.new( 5 )

  prod = Thread.new do
    (0...10).each { |i| buf.ajouter i }
    nb_cons.times { buf.ajouter :FIN }
  end

  thrs = (0...nb_cons).map do |k|
    PRuby.future do
      elems = []
      while (v = buf.retirer) != :FIN
        elems << v if v.odd?
      end

      elems
    end
  end

  p thrs.map(&:value)
        .flatten
        .sort
end
```

1. Qu'est-ce qui sera imprimé par la méthode `test_TamponBorne2`?
2. À quel patron de programmation parallèle est-ce que cela correspond?

Solution :

1. `[1, 3, 5, 7, 9]`

2. Style «coordonnateur–travailleurs» : le `TamponBorne` joue le rôle du sac de tâches partagé entre les travailleurs.

Exercice 6.10 Programme avec *threads* qui utilise une barrière de synchronisation.

1. Qu'est-ce qui sera imprimé par ce programme?
2. Pourquoi `print` plutôt que `puts`?

Solution :

1. Différents résultats sont possibles, par exemple :

```
--- #0: etape 1
--- #1: etape 1
--- #2: etape 1
--- #3: etape 1
--- #2: etape 2
--- #1: etape 2
--------------------
--- #0: etape 2
--- #3: etape 2
--- #3: etape 3
--- #1: etape 3
--- #2: etape 3
--------------------
--- #0: etape 3
```

```
--- #0: etape 1
--- #1: etape 1
--- #2: etape 1
--- #3: etape 1
--- #3: etape 2
--- #2: etape 2
--- #1: etape 2
--------------------
--- #0: etape 2
--------------------
--- #0: etape 3
--- #1: etape 3
--- #3: etape 3
--- #2: etape 3
```

2. L'instruction `print` semble s'exécuter de façon atomique, alors que `puts` n'est assurément pas atomique — l'impression du saut de ligne se fait souvent après d'autres impressions, donc les lignes sont entrelacées.

Exercice 6.11 Un calcul fait dans un style SPMD.

1. Que se passe-t-il si on omet «`if k == 0`»?
2. Que se passe-t-il si on omet le deuxième `attendre`?

Solution :

1. Tous les *threads* font un échange, donc si le nombre est pair, on retournera au même état.

2. Un *thread* pourrait débuter avec les anciennes valeurs.

Exercice 6.12 Une classe `ReentrantLock` pour des verrous réentrants.

Complétez la mise en oeuvre de la classe `ReentrantLock`.

Solution :

```
require 'dbc'

class ReentrantLock
  attr_reader :hold_count, :owner

  def initialize
    # Nombre d'appels a lock sans unlock correspondant.
    @hold_count = 0

    # Thread proprietaire du verrou.
    @owner = nil

    # Pour l'exclusion mutuelle dans le moniteur.
    @mutex = Mutex.new

    # File d'attente pour que le verrou se libere.
    @free = ConditionVariable.new
  end

  def is_locked?
    @mutex.synchronize do
      hold_count > 0
    end
  end

  #
```

```
def lock
  @mutex.synchronize do
    while hold_count > 0 && @owner != Thread.current
      # Un autre thread a le verrou.
      @free.wait(@mutex)
    end

    DBC.assert @hold_count == 0 ||
               @owner == Thread.current

    # Le thread courant a maintenant le verrou...
    # ou l'avait deja!
    @owner = Thread.current
    @hold_count += 1
  end
end

def unlock
  @mutex.synchronize do
    DBC.assert @owner == Thread.current

    @hold_count -= 1
    if @hold_count == 0
      # Dernier unlock en suspens:
      # on signale un des threads en attente.
      @owner = nil
      @free.signal
    end
  end
end

def trylock
  @mutex.synchronize do
    return false if hold_count > 0 &&
                    owner != Thread.current

    # hold_count == 0 || owner == Thread.current
    @owner = Thread.current
    @hold_count += 1
    true
  end
end
```

```
  def is_held_by_current_thread?
    @mutex.synchronize do
      @owner == Thread.current
    end
  end

  def to_s
    self.inspect
  end
end
```

Exercice 6.13 Moniteur pour un compte bancaire (classe `CompteBancaire`).

On veut définir un *moniteur* `CompteBancaire` pour gérer un compte bancaire simple, sans marge de crédit — donc sans solde négatif. Deux opérations sont exportées par ce moniteur — voir plus bas pour le squelette de la classe :

1. `deposer( montant )` : Permet de déposer le `montant` indiqué dans le compte. Si un ou des retraits étaient en attente, alors un ou plusieurs d'entre eux pourront être satisfaits. Ne retourne aucun résultat.

2. `retirer( montant )` : Permet de retirer le `montant` indiqué du compte. Si le `solde` courant n'est pas suffisant pour couvrir le montant du retrait, *l'opération bloque jusqu'à ce que le solde devienne suffisant.* Retourne le `montant` retiré.

```
class CompteBancaire
  attr_reader :solde

  def initialize( solde_initial )
    @solde = solde_initial
    @mutex = Mutex.new
    ...
  end

  def deposer( montant )
    ...
  end

  def retirer( montant )
    ...
  end
end
```

Solution :

```
class CompteBancaire
  attr_reader :solde

  def initialize( solde_initial )
    @solde = solde_initial
    @mutex = Mutex.new
    @solde_modifie = ConditionVariable.new
  end

  def deposer( montant )
    @mutex.synchronize do
      @solde += montant
      @solde_modifie.broadcast
    end
  end

  def retirer( montant )
    @mutex.synchronize do
      @solde_modifie.wait(@mutex) while montant > @solde
      @solde -= montant

      montant
    end
  end
end
```

Exercice 6.14 Moniteur pour un `Echangeur`.

On veut définir un *moniteur* `Echangeur` qui permet à deux *threads* de s'échanger une valeur, et ce par l'intermédiaire de l'opération `echanger( valeur )` — voir plus bas pour le squelette de la classe.

Lorque le 1^{er} *thread* arrive, il bloque. Lorsque le 2^{e} *thread* arrive, le moniteur échange les valeurs reçues et réactive les *threads* en retournant les valeurs échangées.

Le moniteur doit évidemment être réutilisable, c'est-à-dire qu'il doit permettre d'effectuer l'échange entre deux *threads*, puis deux autres, etc.

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

```
t1 = Thread.new { e.echanger(10) }
t2 = Thread.new { e.echanger(20) }
t1.value.must_equal 20
t2.value.must_equal 10
```

```
class Echangeur
  def initialize
    @mutex = Mutex.new
    ...
  end

  def echanger( valeur )
    ...
  end
end
```

P.S. Cette classe est semblable à la classe `Exchanger` de Java : `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Exchanger.html`

Solution :

```
class Echangeur
  def initialize
    @valeur_du_premier = nil
    @mutex = Mutex.new
    @deuxieme_est_arrive = ConditionVariable.new
  end

  def echanger( valeur )
    @mutex.synchronize do
      if @valeur_du_premier.nil?
        # Premiere arrivee.
        @valeur_du_premier = valeur
        @deuxieme_est_arrive.wait( @mutex )
        @valeur_du_deuxieme
      else
        # Deuxieme arrivee.
        @valeur_du_deuxieme = valeur
        resultat = @valeur_du_premier
        @valeur_du_premier = nil
        @deuxieme_est_arrive.signal

        resultat
      end
    end
  end
end
```

Exercice 6.15 Moniteur pour un `Reducteur`.

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

- Tant que tous les `nb_threads` *threads* n'ont pas effectué leur appel à `reduire`, aucun résultat n'est produit. 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 est retourné à chacun des *threads*, lesquels poursuivent ensuite leur exécution.

Le moniteur doit être réutilisable — doit permettre d'effectuer plusieurs réductions. Pour simplifier, vous pouvez supposer que les séries d'appels ne se chevauchent pas — 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 }
```

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

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

Solution :

```
class Reducteur
  def initialize( val_initiale, nb_threads, &bloc )
    @val_initiale = val_initiale
    @nb_threads = nb_threads
    @bloc = bloc

    @nb_arrives = 0
    @mutex = Mutex.new
    @tous_arrives = ConditionVariable.new
    @total = nil
  end

  #
```

```
  def reduire( valeur )
    @mutex.synchronize do
      @nb_arrives += 1
      @total = @bloc.call(@total || @val_initiale, valeur)

      if @nb_arrives < @nb_threads
        # Pas le dernier, donc on attend...
        @tous_arrives.wait( @mutex )
        @resultat
      else
        # Le dernier thread vient d'arriver...

        # On prend en note le resultat final.
        @resultat = @total

        # On reinitialise pour la vague suivante.
        @nb_arrives = 0
        @total = @val_initiale

        # On signale les threads arrives avant.
        @tous_arrives.broadcast

        # On retourne le resultat
        @resultat
      end
    end
  end
end
```
