Table des matières

Chapitre 6

Programmation concurrente avec *threads* en Ruby (`Thread` et al.)

6.1 Classe `Thread` : *threads* et futures

Dans ce qui suit, nous présentons les notions de base des *threads* Ruby, donc les méthodes de la classe `Thread` ainsi que diverses classes associées.

Il est important de souligner qu'en Ruby version MRI (appellé aussi CRuby, la mise en oeuvre la plus connue puisque développée par le concepteur du langage Ruby), les *threads* sont *concurrents mais ne sont pas parallèles*.

Ceci implique notamment, pour des raisons trop longues à expliquer ici, que ces *threads* ne peuvent pas vraiment exploiter les coeurs multiples d'un processeur multi-coeurs. Par contre, JRuby permet d'avoir de vrais *threads* — et ce en utilisant les *threads* de la JVM. C'est pour cette raison que dans le cours INF5171, nous utiliserons JRuby et que la bibliothèque `PRuby` ne fonctionne qu'avec JRuby. Toutefois, les exemples qui suivent sont valides pour n'importe quelle mise en oeuvre de Ruby, puisqu'ils utilisent exclusivement les classes `Thread` et `Mutex`.

Note : Bien que l'exécution de programmes Ruby avec JRuby soit efficace, un désavantage de JRuby est que le lancement d'un programme est un peu plus long — notamment parce que la machine virtuelle Java doit elle aussi être lancée. On verra en cours (ou en laboratoire) comment spécifier certaines options d'exécution qui accélèrent le lancement des programmes avec JRuby.

6.1.1 Méthode `Thread.new`

Programme Ruby 6.1 Un programme «Hello World!» avec *threads*.

```
$ cat hello-threads.rb
require 'system'

nb_threads = ARGV[0] ? ARGV[0].to_i : System::CPU.count

nb_threads.times do |k|
  Thread.new do
    puts "[\##{k} de #{nb_threads}] Bonjour le monde!"
  end
end
```

Remarques et explications pour le Programme Ruby 6.1 :

- Comme en Java et C, le tableau `ARGV` contient les arguments spécifiés lors de l'appel du programme sous forme de chaînes de caractères (`String`). Pour ce programme, on peut donc spécifier le nombre de *threads* qu'on désire exécuter.
- Si un argument est spécifié sur la ligne de commande (`ARGV[0]` n'est pas `nil`), sa valeur numérique est obtenue, avec `to_i`. Si aucun argument n'est spécifié, alors on utilise le nombre de coeurs/processeurs de la machine.
- On crée et on lance l'exécution d'un *thread* en appelant le constructeur de la classe `Thread`. C'est le bloc passé à `new` qui représente le code qui sera exécuté par le *thread*.
- L'appel à `Thread.new` retourne un objet `Thread` — qu'on appelle parfois le *descripteur du thread*. Dans le cas présent, rien n'est fait avec ces descripteurs de *thread*.
- L'exemple Ruby 6.1 présente diverses exécutions de ce programme à partir de la ligne de commande.

  Question : Que se passe-t-il?

Exemple Ruby 6.1 Des exemples d'exécution du programme «Hello World!» avec *threads* (version «naïve»).

```
$ ruby hello-threads.rb 10
[#4 de 10] Bonjour le monde!
[#3 de 10] Bonjour le monde!$ ruby hello-threads.rb 10
[#3 de 10] Bonjour le monde!
[#0 de 10] Bonjour le monde!$ ruby hello-threads.rb 10
[#3 de 10] Bonjour le monde!$ ruby hello-threads.rb 10
[#3 de 10] Bonjour le monde!$ ruby hello-threads.rb 10
[#1 de 10] Bonjour le monde!
$ ruby hello-threads.rb 10
[#3 de 10] Bonjour le monde!
$ ruby hello-threads.rb 10
[#2 de 10] Bonjour le $ ruby hello-threads.rb 10
[#3 de 10] Bonjour le monde!
[#1 de 10] Bonjour le monde!$
```

6.1.2 Méthode `Thread#join`

Programme Ruby 6.2 Un programme «Hello World!» avec *threads* et `join` — style impératif.

```
$ cat hello-threads.rb
require 'system'

nb_threads = ARGV[0] ? ARGV[0].to_i : System::CPU.count

threads = []
nb_threads.times do |k|
  threads << Thread.new do
    puts "[\##{k} de #{nb_threads}] Bonjour le monde!"
  end
end

threads.each do |tr|
  tr.join
end
```

Programme Ruby 6.2 Un programme «Hello World!» avec *threads* et `join` — style fonctionnel.

```
$ cat hello-threads.rb
require 'system'

nb_threads = ARGV[0] ? ARGV[0].to_i : System::CPU.count

threads = (0...nb_threads).map do |k|
  Thread.new do
    puts "[\##{k} de #{nb_threads}] Bonjour le monde!"
  end
end

threads.map(&:join)
```

Le Programme Ruby 6.2 présente le même programme «*Hello world*», mais cette fois écrit dans un style fonctionnel.

Remarques et explications pour le Programme Ruby 6.2 :

- L'appel à `Thread.new` retourne un objet `Thread` — le descripteur du thread. C'est ce descripteur qui permet de ne terminer l'exécution du programme que lorsque tous les *threads* ont eux-même terminé leur exécution : l'appel de la méthode `join` sur un `Thread t` *bloque jusqu'à ce que le* thread `t` *ait terminé son exécution* — donc lorsque le bloc associé au *thread* a terminé.

- Voir l'exemple Ruby 6.2 pour diverses exécutions de ce programme.

Exemple Ruby 6.2 Des exemples d'exécution du programme «Hello World!» avec *threads* (version avec `join`).

```
$ ruby hello-threads.rb 10
[#4 de 10] Bonjour le monde!
[#1 de 10] Bonjour le monde![#2 de 10] Bonjour le monde!
[#6 de 10] Bonjour le monde![#8 de 10] Bonjour le monde!
[#7 de 10] Bonjour le monde!
[#5 de 10] Bonjour le monde!
[#3 de 10] Bonjour le monde!
[#0 de 10] Bonjour le monde!
[#9 de 10] Bonjour le monde!


$ ruby hello-threads.rb 10
[#2 de 10] Bonjour le monde!
[#4 de 10] Bonjour le monde!
[#1 de 10] Bonjour le monde!
[#6 de 10] Bonjour le monde!
[#5 de 10] Bonjour le monde![#7 de 10] Bonjour le monde![#0 de 10]
Bonjour le monde!
[#3 de 10] Bonjour le monde!

[#8 de 10] Bonjour le monde![#9 de 10] Bonjour le monde!


$ ruby hello-threads.rb 10
[#0 de 10] Bonjour le monde!
[#2 de 10] Bonjour le monde!
[#4 de 10] Bonjour le monde!
[#9 de 10] Bonjour le monde!
[#1 de 10] Bonjour le monde!
[#8 de 10] Bonjour le monde!
[#5 de 10] Bonjour le monde!
[#7 de 10] Bonjour le monde!
[#6 de 10] Bonjour le monde!
[#3 de 10] Bonjour le monde!
$
```

Remarques et explications pour l'exemple Ruby 6.2 :

- Chaque *thread* effectue son impression avant que le programme ne se termine — on a donc toujours 10 impressions.
- Les impressions effectuées par les divers *threads* sont parfois *entrelacées*.

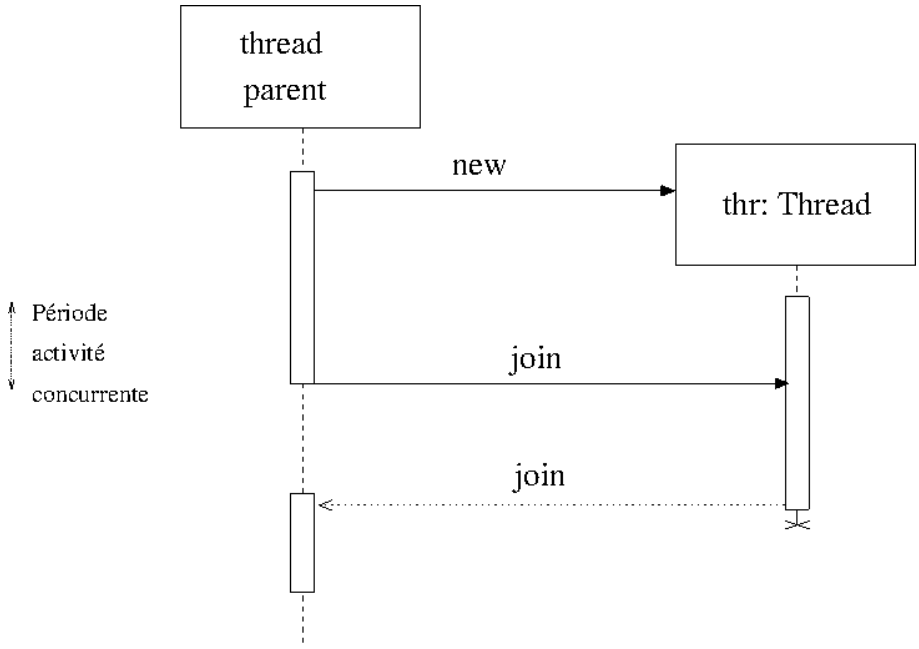

Figure 6.1: Diagramme d'interactions UML illustrant le comportement de la méthode `join`.

La Figure 6.1 illustre, à l'aide d'un diagramme d'interactions UML, le comportement d'un *thread* parent qui crée puis attend un *thread* enfant avec `join`. On remarque que pendant une certaine période, les deux *threads* sont actifs. Par contre, lorsque le *thread* parent fait un appel à `join` et que le *thread* enfant n'a pas terminé, le parent devient inactif jusqu'à ce que le `join` soit complété, au moment où le *thread* enfant se termine.

6.1.3 Méthode `Thread#value`

Un *thread* Ruby est en fait… un *future* : si un *thread* retourne un résultat, on peut simplement appeler la méthode `value` pour obtenir la valeur rétournée, en bloquant si la valeur n'est pas encore disponible :

```
>> tr = Thread.new { sleep 1; 2  + 2 }
=> #<Thread:0x290d210d run>

>> puts tr.value
4
=> nil
```

6.2 Classe `Mutex` : verrous d'exclusion mutuelle

Programme Ruby 6.3 Un programme «Hello World!» avec *threads* et verrou d'exclusion mutuelle.

```
require 'system'

nb_threads = ARGV[0] ? ARGV[0].to_i : System::CPU.count

verrou = Mutex.new

threads = (0...nb_threads).map do |k|
  Thread.new do
    verrou.synchronize do
      puts "[\##{k} de #{nb_threads}] Bonjour le monde!"
    end
  end
end

threads.map(&:join)
```

Remarques et explications pour le Programme Ruby 6.3 :

- On crée un `verrou` *d'exclusion mutuelle* — `mut`*ual* `ex`*clusion*.
- Lorsqu'on désire exécuter un segment de code de façon exclusive, donc sans interférence par d'autres *threads*, il suffit d'exécuter ce segment de code à l'intérieur d'un bloc passé à la méthode `synchronize`.
- Le segment de code avec `verrou.synchronize do... end` est équivalent au segment de code suivant :

```
verrou.lock
  puts "[\##{k} de #{nb_threads}] Bonjour...!"
verrou.unlock
```

En Ruby version MRI (resp. JRuby), la méthode `synchronize` est mise en oeuvre en C (resp. Java). Voici toutefois une version Ruby équivalente, avec `yield` puisque `synchronize` doit recevoir un bloc :

```
class Mutex
  def synchronize
    unless block_given?
      raise "... 'synchronize': must be called with a block..."
    end

    lock
      yield
    unlock
  end
end
```

- Avec le verrou, les impressions se font donc sans interférence, comme le montre l'exemple Ruby 6.3. Évidemment, dans ce cas simple où chaque *thread* n'exécute qu'une seule instruction protégée par le verrou, il n'y a plus de concurrence, i.e., l'exécution des *threads* se fait dans un ordre *arbitraire*, mais *séquentialisé*.

Exemple Ruby 6.3 Des exemples d'exécution du programme «Hello World!» avec *threads* (version avec verrou).

```
$ ruby hello-threads.rb 10
[#3 de 10] Bonjour le monde!
[#4 de 10] Bonjour le monde!
[#1 de 10] Bonjour le monde!
[#0 de 10] Bonjour le monde!
[#2 de 10] Bonjour le monde!
[#5 de 10] Bonjour le monde!
[#9 de 10] Bonjour le monde!
[#8 de 10] Bonjour le monde!
[#7 de 10] Bonjour le monde!
[#6 de 10] Bonjour le monde!
$ ruby hello-threads.rb 10
[#2 de 10] Bonjour le monde!
[#0 de 10] Bonjour le monde!
[#1 de 10] Bonjour le monde!
[#4 de 10] Bonjour le monde!
[#5 de 10] Bonjour le monde!
[#7 de 10] Bonjour le monde!
[#3 de 10] Bonjour le monde!
[#6 de 10] Bonjour le monde!
[#8 de 10] Bonjour le monde!
[#9 de 10] Bonjour le monde!
$ ruby hello-threads.rb 10
[#3 de 10] Bonjour le monde!
[#0 de 10] Bonjour le monde!
[#4 de 10] Bonjour le monde!
[#6 de 10] Bonjour le monde!
[#2 de 10] Bonjour le monde!
[#1 de 10] Bonjour le monde!
[#7 de 10] Bonjour le monde!
[#5 de 10] Bonjour le monde!
[#8 de 10] Bonjour le monde!
[#9 de 10] Bonjour le monde!
$
```

6.3 Classe `ConditionVariable` : variables de condition

On parle de **synchronisation conditionelle** lorsqu'un *thread*, pour faire son travail correctement, doit *attendre qu'une certaine condition devienne vraie*... et que cette condition ne pourra devenir vraie *que grâce à un autre* thread. En d'autres mots, c'est un autre *thread* qui va rendre cette condition vraie et permettre au premier *thread* de réaliser sa tâche — il s'agit donc d'une forme de synchronisation entre les *threads*.

Les verrous, vus à la section précédente, permettent d'assurer l'exclusion mutuelle entre *threads* mais ne permettent pas la synchronisation conditionelle.

Pour ce faire, il faut plutôt utiliser des *variables de condition*. Une telle variable est utilisée pour suspendre un *thread* jusqu'à qu'une certaine condition devienne vraie. Le *thread* qui doit attendre *se suspend lui-même*, de façon explicite et inconditionnelle. Par la suite, c'est un autre *thread* qui transmettra un *signal* pour indiquer que la condition ayant causé l'attente est maintenant devenue vraie.

La classe `ConditionVariable`

En Ruby, on manipule des variables de condition comme suit :

- `cond = ConditionVariable.new` : Crée une nouvelle variable de condition.

  À cette variable de condition est implicitement associée une *file d'attente de* threads : voir les prochaines instruction, `wait` et `signal`.

- `cond.wait( v )` : Le *thread* qui appelle `cond.wait` est *suspendu* — mis en attente, et ce de façon *inconditionnelle* — sur la variable de condition `cond`.

  Pour que l'exécution de `wait( v )` soit valide, le *thread* **doit être en possession** du *verrou* `v`. L'appel à `wait` a alors pour effet secondaire **de libérer le verrou** `v`, et donc de le rendre accessible à d'autres *threads* qui pourront ainsi accéder à leur région critique.

- `cond.signal` : Si un ou plusieurs *threads* sont actuellement en attente sur la variable de condition `cond`, alors *un (et un seul) d'entre eux sera réactivé* — typiquement, le premier dans la *file d'attente*, bien que cela ne soit pas nécessairement une spécification de la mise en oeuvre.[1] Lorsque le *thread* ainsi réactivé poursuivra son exécution *à la sortie* de l'instruction `wait`, il sera assuré d'être **en possession** du verrou `v` qui avait été spécifié dans le

[1] Donc, un programme qui dépend pour son bon fonctionnement que ce soit toujours le premier *thread* sur la file d'attente qui soit réactivé n'est pas un programme correct.

`wait`. Par contre, il ne pourra obtenir ce verrou que lorsque le *thread* ayant lancé le signal aura lui-même libéré ce verrou — approche dite «`Signal & Continue`».

De plus, rien n'assure que ce soit le *thread* ayant reçu le signal qui prendra effectivement possession du verrou, car d'autres *threads* peuvent aussi être en attente pour ce verrou. C'est pour cette raison que l'utilisation d'une clause `wait` a toujours (ou presque) l'allure suivante — notez le `while` et non un `if` :

```
cond.wait( v ) while condition à satisfaire
```

- `cond.broadcast` : Semblable à `signal`, à la différence que tous les *threads* en attente recevront le signal et sortiront de la file d'attente de la variable `cond`. Par contre, s'il y a plusieurs *threads*, un seul reprendra possession du verrou `v` ; les autres ne seront plus bloqués sur la variable de condition `cond`, mais plutôt en attente d'acquérir le verrou `v`.

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"
```

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

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"
```

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.wait(mutex)
   puts "Apres wait"
  end
end

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

t.join
puts "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
  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"
```

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

6.4 La notion de «moniteur»

Un *moniteur* est un module — par exemple, une classe — qui *encapsule* (dissimule) une ressource *partagée* (par ex., des données) entre différents *threads*.

Le *moniteur* définit des *méthodes* qui permettent *de manipuler la ressource de façon sécure*, sans situation de compétition entre les *threads* :

- Interface publique : *L'exclusion mutuelle* entre les clients *est implicite*, car les opérations du moniteur sont protégées, à l'interne, par un verrou privé d'exclusion mutuelle, verrou connu uniquement du moniteur. Les clients n'ont donc pas à gérer ou utiliser ce verrou[2].

- Mise en oeuvre interne : Le moniteur utilise une (ou plusieurs) variable(s) de condition, elles aussi internes au moniteur, pour assurer le bon comportement des opérations publiques du moniteur — notamment, les *synchronisations conditionnelles*.

Autre caractéristique d'un programme avec moniteurs = Le programme est composé de deux sortes «d'entités» :

1. Des *threads actifs* — les **clients** des moniteurs — qui communiquent/échangent des informations par l'intermédiaire de moniteurs.
2. Des moniteurs *passifs* — les serveurs — qui gèrent les accès aux ressources partagées :

    - Les méthodes publiques d'un moniteur assurent (implicitement) l'exclusion mutuelle entre les clients.
    - Les variables de condition sont utilisées pour synchroniser les clients.

Les interactions et synchronisations entre *threads* (actifs) se font donc exclusivement *par l'intermédiaire* de moniteurs (passifs).

[2] Pour certains moniteurs plus complexes, il peut parfois être nécessaire d'avoir plusieurs verrous.

6.4.1 Exemple : Une classe `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
```

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
      @condition.signal
    end
  end

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

      @valeur
    end
  end
end
```

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

6.4.2 Exemple : tampon borné

Le Programme Ruby 6.4 présente une classe `TamponBorne` qui réalise un moniteur pour gérer un *tampon de taille bornée — bounded buffer*, *bounded queue*. Ce tampon peut être utilisé par un ou plusieurs producteurs pour transmettre des données à un ou plusieurs consommateurs. C'est le tampon, défini comme un *moniteur*, qui assure l'exclusion mutuelle pour la gestion correcte du tampon — avec `@mutex` — ainsi que la synchronisation conditonnelle entre producteurs et consommateurs — avec `@pas_vide` et `@pas_plein`.

Les éléments du tampon sont gérés par l'intermédiaire d'un *tampon circulaire* (*ring buffer*) : la tête et la queue avancent... puis retournent au début — donc calcul modulo la `taille`. Voir Figure 6.2.

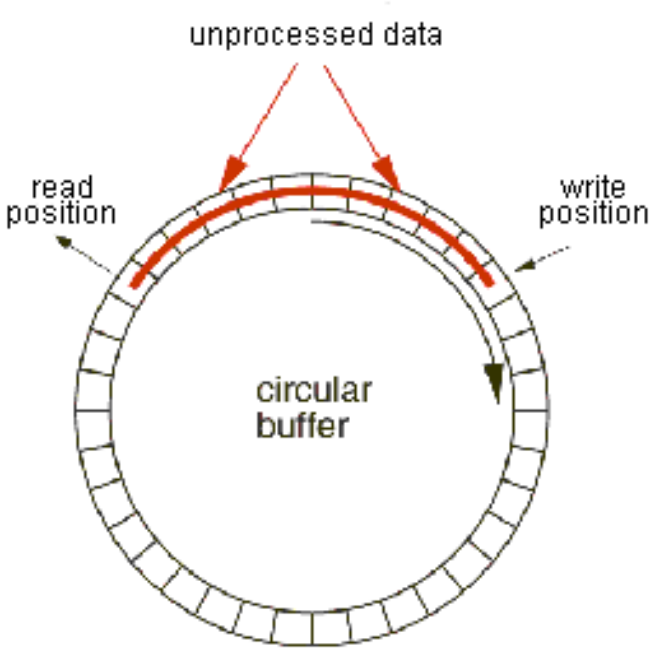

Figure 6.2: Un *tampon circulaire* (*circular buffer*, *ring buffer*). Source : `http://stackoverflow.com/questions/30569897/benchmark-results-forkjoin-vs-disruptor`.

Programme Ruby 6.4 Classe `TamponBorne` définie comme un moniteur pour gérer un tampon borné.

```ruby
class TamponBorne
  def initialize( taille )
    @taille = taille
    @tampon = Array.new( taille ) # Tampon circulaire.
    @tete = @queue = @nb_elems = 0
    @mutex = Mutex.new # Pour moniteur!
    @pas_plein = ConditionVariable.new
    @pas_vide = ConditionVariable.new
  end

  def ajouter( data )
    @mutex.synchronize do # On attend si plein.
      @pas_plein.wait( @mutex ) while @nb_elems == @taille
      @tampon[@queue] = data # On ajoute en queue.
      @queue = (@queue + 1) % @taille
      @nb_elems += 1
      @pas_vide.signal # N'est pas vide!
    end
  end

  def retirer
    result = nil
    @mutex.synchronize do # On attend si vide.
      @pas_vide.wait( @mutex ) while @nb_elems == 0
      result = @tampon[@tete] # On prend la tete.
      @tete = (@tete + 1) % @taille
      @nb_elems -= 1
      @pas_plein.signal # N'est pas plein!
    end
    result
  end
end
```

Programme Ruby 6.5 Une méthode de test avec trois *threads* qui partagent deux TamponBornes.

```ruby
def test_TamponBorne1
  buf1 = TamponBorne.new( 10 )
  buf2 = TamponBorne.new( 1 )

  Thread.new do
    (0...10).each { |i| buf1.ajouter i }
    buf1.ajouter :FIN
  end

  Thread.new do
    while (v = buf1.retirer) != :FIN
      buf2.ajouter v if v.even?
    end
    buf2.ajouter :FIN
  end

  t = Thread.new do
    res = []
    while (v = buf2.retirer) != :FIN
      res << v
    end
    res
  end

  p t.value  # p x = puts x.inspect
end
```

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`?

Exercice 6.7: Programme qui utilise `TamponBorne`.

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

```
def ajouter( data )
  @mutex.synchronize do
    @pas_plein.wait(@mutex) while @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) while @nb_elems == 0
    result = @tampon[@tete]
    @tete = (@tete + 1) % @taille
    @nb_elems -= 1
    @pas_plein.signal
  end

  result
end
```

Exercice 6.8: Mise en oeuvre de `TamponBorne`.

```ruby
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?

Exercice 6.9: Un autre programme qui utilise `TamponBorne`.

6.4.3 Exemple : barrière de synchronisation

Un mécanisme de synchronisation disponible dans la plupart des langages de programmation parallèle est celui de *barrière de synchronisation*.

Une barrière représente un point de synchronisation **qui doit être atteint par tous les *threads* d'un groupe** avant qu'ils puissent poursuivre leur exécution.

Exemple Ruby 6.4 Un (segment de) programme qui utilise une barrière de synchronisation.

```
nb_threads = 4

barriere = Barriere.new( nb_threads )

threads = (0...nb_threads).map do |k|
  Thread.new do
    print "--- \##{k}: etape 1\n"

    barriere.attendre
    print "#{'-' * 20}\n" if k == 0

    print "--- \##{k}: etape 2\n"

    barriere.attendre
    print "#{'-' * 20}\n" if k == 0

    print "--- \##{k}: etape 3\n"
  end
end

threads.map(&:join)
```

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

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

Exemple Ruby 6.5 Un segment de code utilisant une barrière de synchronisation.

```
nb_threads = 4

barriere = Barriere.new( nb_threads )

threads = (0...nb_threads).map do |k|
  Thread.new do
    print "--- \##{k}: etape 1\n"

    barriere.attendre
    print "#{'-' * 20}\n" if k == 0
    barriere.attendre

    print "--- \##{k}: etape 2\n"

    barriere.attendre
    print "#{'-' * 20}\n" if k == 0
    barriere.attendre

    print "--- \##{k}: etape 3\n"
  end
end

threads.map(&:join)
```

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

Exemple d'exécution 6.1: Un exemple d'exécution du segment de code de l'exemple Ruby 6.5.

6.5 Programmes SPMD et barrières de synchronisation

Dans certains langages, il n'est pas possible de créer et lancer des *threads* (ou processus) durant l'exécution du programme. Il faut plutôt indiquer, *au lancement du programme*, le nombre de *threads* (ou processus) à utiliser.

Ce modèle, appelé *SPMD* (*Single Program, Multiple Data*), est un des plus courants en programmation parallèle sur les machines à mémoire distribuée — notamment avec la bibliothèque MPI.

Dans ce modèle, les divers processus exécutent tous *le même programme*, mais pas nécessairement la même instruction à un instant donné ($\neq$ SIMD!).

Lorsque cela est nécessaire pour assurer le bon fonctionnement d'un programme SPMD, les différents processus peuvent se synchroniser entre eux, généralement à l'aide d'une *barrière* de synchronisation.

Exemple Ruby 6.6 La mise en oeuvre de la classe `Barriere`.

```ruby
class Barriere

  # Creation de la barriere pour le nombre de threads indique.
  def initialize( nb_threads )
    @nb_threads = nb_threads

    # Pour l'exclusion mutuelle entre les clients.
    @mutex = Mutex.new

    # Pour suspendre un thread en attente des autres threads.
    @tous_arrives = ConditionVariable.new

    # Nombre de threads actuellement en attente.
    @nb_en_attente = 0
  end

  # Attendre jusqu'a ce que tous les threads soient arrives.
  def attendre
    @mutex.synchronize do
      @nb_en_attente += 1

      if @nb_en_attente < @nb_threads
        # Pas le dernier thread => On attend!
        @tous_arrives.wait(@mutex)
      else
        # Dernier thread => on reactive les threads.
        @tous_arrives.broadcast

        # On prepare la prochaine utilisation.
        @nb_en_attente = 0
      end
    end
  end

end
```

Exemple Ruby 6.7 Un petit programme simple en style non-SPMD, avec création répétitive et dynamique de *threads*.

```
def calculer( k, a, a_bis )
  # Examine une portion de a, specifique a k,
  # et met le resultat dans une portion specifique
  # de a_bis.
  ...
end

a = Array.new( n ){ val_initiale }
a_bis = Array.new( n )

# On itere pendant un certain nombre d'iterations
(0...nb_iterations).each do |i|
  PRuby.pcall( 0...nb_threads,
               lambda do |k|
                 calculer( k, a, a_bis )
               end
               )

  # On utilise le resultat calcule comme nouveau
  # point de depart, donc on interchange.
  a, a_bis = a_bis, a
end

# Resultat final dans a!
```

Exemple Ruby 6.8 Une version SPMD, sans création répétitive et dynamique de *threads*, de l'exemple Ruby 6.7.

```
def calculer( k, a, a_bis ) ... end

a = Array.new( n ){ val_initiale }
a_bis = Array.new( n )

barriere = Barriere.new( nb_threads )

PRuby.pcall( 0...nb_threads,
             lambda do |k|
               (0...nb_iterations).each do |i|
                 calculer( k, a, a_bis )
                 barriere.attendre

                 a, a_bis = a_bis, a if k == 0
                 barriere.attendre
               end
             end
             )

# Resultat final dans a!
```

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

Exercice 6.11: Un calcul fait dans un style SPMD.

6.6 Autres méthodes de la classe `Thread`

6.6.1 Méthodes `Thread.abort_on_exception` et `Thread#abort_on_exception`

`Thread.abort_on_exception` :

> *When set to true, all threads will abort if an exception is raised. Returns the new state.*

Important pour déboguer... sinon un *thread* peut signaler une exception et on ne la voit jamais ☹

```
$ cat abort.rb
Thread.abort_on_exception = ARGV[0] == 'true'

Thread.new do
  raise "Erreur"
end

sleep 1
puts "** Fin! **"
```

```
$ ruby abort.rb true
abort.rb:4:in 'block in <main>':
           Erreur (RuntimeError)



$ ruby abort.rb false
** Fin! **
```

6.6.2 Méthode `Thread.current`

Retourne le *thread* courant :

```
$ cat current.rb
t1 = Thread.new do
  sleep 0.1
  puts "t1: #{Thread.current}"
end

t2 = Thread.new do
  puts "t2: #{Thread.current}"
end

t1.join
t2.join

puts "main: #{Thread.current}"
```

```
$ ruby current.rb
t2: #<Thread:0x7ffe8408>
t1: #<Thread:0x20366593>
main: #<Thread:0x1a407d53>
```

6.6.3 Variables locales à un *thread*

```
$ cat local-vars.rb
threads =
  [Thread.new { Thread.current[:var] = "A" },
   Thread.new { Thread.current[:var] = "B" },
   Thread.new { Thread.current[:foo] = "C" },
   ]

threads.map(&:join)

threads.each do |th|
  puts "#{th}: #{th[:var]}, #{th[:foo]}"
  puts "#{th.inspect}: #{th[:var].inspect}, "\
                      "#{th[:foo].inspect}"
  puts
end
```

```
$ ruby local-vars.rb
#<Thread:0x000000019010e0>: A,
#<Thread:0x000000019010e0 dead>: "A", nil

#<Thread:0x00000001900ff0>: B,
#<Thread:0x00000001900ff0 dead>: "B", nil

#<Thread:0x00000001900f00>: , C
#<Thread:0x00000001900f00 dead>: nil, "C"
```

6.6.4 Thread#status

```
a = Thread.new { raise("die now") }
b = Thread.new { Thread.stop }
c = Thread.new { Thread.exit }
d = Thread.new { sleep }
d.kill                  #=> #<Thread:0x401b3678 aborting>
a.status                #=> nil
b.status                #=> "sleep"
c.status                #=> false
d.status                #=> "aborting"
Thread.current.status #=> "run"
```

Source : http://ruby-doc.org/core-2.2.0/Thread.html#method-c-abort_on_exception-3D

Les différents statuts :

"sleep"	*Returned if this thread is sleeping or waiting on I/O*
"run"	*When this thread is executing*
"aborting"	*If this thread is aborting*
false	*When this thread is terminated normally*
nil	*If terminated with an exception.*

6.7 Classes `Queue` et `SizedQueue` : communication entre *threads*

Des structures de données pour tampons *thread-safe*, bornés et non-bornés, sont disponibles dans la bibliothèque Ruby :

- Classe `Queue` : «*This class provides a way to synchronize communication between threads.*»[3]
- Classe `SizedQueue` : «*This class represents queues of specified size capacity. The push operation may be blocked if the capacity is full.*»[4]

Exemple simple tiré de la documentation Ruby : [5]

```
queue = Queue.new

producer = Thread.new do
  5.times do |i|
    sleep rand(i) # simulate expense
    queue << i
    puts "#{i} produced"
  end
end

consumer = Thread.new do
  5.times do |i|
    value = queue.pop
    sleep rand(i/2) # simulate expense
    puts "consumed #{value}"
  end
end

consumer.join
```

---

[3] http://ruby-doc.org/stdlib-2.0.0/libdoc/thread/rdoc/Queue.html
[4] http://ruby-doc.org/stdlib-2.0.0/libdoc/thread/rdoc/SizedQueue.html
[5] http://ruby-doc.org/stdlib-2.0.0/libdoc/thread/rdoc/Queue.html

Résultat possible d'exécution :

```
0 produced
1 produced
2 produced
consumed 0
3 produced
consumed 1
consumed 2
consumed 3
4 produced
consumed 4
```

6.A Mise en oeuvre simplifiée de la classe `ConditionVariable`

Programme Ruby 6.6 Mise en oeuvre (très simplifiée) des `ConditionVariable`.

```
class ConditionVariable
  def initialize
    @attente = Semaphore.new( 0 )
    @nb_en_attente = 0
  end

  def wait( mutex )
    @nb_en_attente += 1
    mutex.unlock
    @attente.acquire  # Va bloquer!
    mutex.lock
  end

  def signal
    if @nb_en_attente > 0
      @nb_en_attente -= 1
      @attente.release
    end
  end

  def broadcast
    while @nb_en_attente > 0
      @nb_en_attente -= 1
      @attente.release
    end
  end
end
```

Le programme Ruby 6.6 présente une mise en oeuvre *(très!) simplifiée* de la classe `ConditionVariable`, et ce en utilisant un `Semaphore`.

Un sémaphore est sorte *spéciale* de variable contenant un nombre entier *non-négatif* et manipulée uniquement par deux opérations atomiques : `acquire` (appelée aussi `P`) et `release` (appelée aussi `V`).

1. `acquire` = `P` = *Passeren* ≈ passer, prendre ⇒ Décrémente la variable… à moins qu'elle ne soit déjà 0, auquel cas l'opération bloque.

   Utilisée pour suspendre un *thread* jusqu'à ce qu'un événement survienne.

2. `release` = `V` = *Vrijgeven* ≈ relâcher ⇒ Incrémente la variable.

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

Cette mise en oeuvre simplifiée vise essentiellement à illustrer que la libération du verrou lors d'un `wait` et sa réacquisition lors de la réception d'un signal se fait implicitement au niveau de la mise en oeuvre de l'opération `wait`, et non explicitement dans le code du client du moniteur.

6.B Exercices

6.B.1 Exercice : verrous réentrants

On dit d'un verrou qu'il est *réentrant* si un *thread* qui possède déjà le verrou peut «acquérir à nouveau» ce même verrou *sans bloquer*.

Fait : Les verrous de base — classe `Mutex` — en Ruby *ne sont pas réentrants!*

```
$ cat mutex.rb
mutex = Mutex.new

t = Thread.new do
  mutex.synchronize do
    puts "*** apres premier sync"
    mutex.synchronize do
      puts "--- apres deuxieme sync"
    end
  end
end

t.join

==================================

$ ruby mutex.rb
*** apres premier sync
ThreadError: Mutex relocking by same thread
         lock at org/jruby/ext/thread/Mutex.java:90
  synchronize at org/jruby/ext/thread/Mutex.java:147
       (root) at mutex.rb:6
  synchronize at org/jruby/ext/thread/Mutex.java:149
       (root) at mutex.rb:4
```

Exemple d'exécution 6.2: Exemple qui montre que les `Mutex` de Ruby ne sont pas réentrants.

La bibliothèque `java.util.concurrent` définit une classe `ReentrantLock`. Le programme Ruby 6.7 présente l'interface d'une classe Ruby équivalente.

Programme Ruby 6.7 Interface d'une classe `ReentrantLock` en `Ruby`.

```
class ReentrantLock
  attr_reader :hold_count, :owner

  def initialize
    ...
  end

  def is_locked?
    ...
  end

  def lock
    # Retourne immediatement si le thread appelant
    # est deja proprietaire du verrou.
    ...
  end

  def unlock
    ...
  end

  def trylock
    # Retourne true si le thread appelant
    # a pu obtenir le verrou.
    # Sinon retourne false et ne bloque pas!
    ...
  end

  def is_held_by_current_thread?
    ...
  end
end
```

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

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

6.B.2 Exercices : moniteurs

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
```

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

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`

Exercice 6.14: Moniteur pour un `Echangeur`.

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
```

Exercice 6.15: Moniteur pour un `Reducteur`.

Références