Solution devoir # 2

1 Extraits du fichier `svar.rb`

```ruby
def self.all( *svars )
  DBC.require( svars.size >= 1, "*** Il doit y avoir au moins un argument" )

  new( :write_once, :async ) { svars.map(&:value) }
end

def self.any( *svars )
  DBC.require( svars.size >= 1, "*** Il doit y avoir au moins un argument" )

  new( :write_once, :async ) do
    mutex = Mutex.new
    val_definie = ConditionVariable.new
    nb_termines = 0  # Au cas ou toutes les svars retournent nil.
    val = nil

    svars.each do |sv|
      Thread.new do
        ma_val = sv.value
        mutex.synchronize do
          val = ma_val unless ma_val.nil?
          nb_termines += 1
          val_definie.signal
        end
      end
    end

    mutex.synchronize do
      val_definie.wait(mutex) while val.nil? && nb_termines < svars.size
    end

    val
  end
end

#
```

```ruby
class SVar
  def value
    @mutex.synchronize do
      unfreeze if frozen?
      @is_full.wait( @mutex ) until full?

      @value
    end
  end

  def frozen?
    !@block.nil?
  end

  def unfreeze
    DBC.require( frozen?, "*** Appel a unfreeze alors que pas gele!?" )

    bloc = @block
    @block = nil
    @state = :in_evaluation
    @value = bloc.call  # Maitenant on bloque...
    @state = :full
  end

  private :unfreeze, :frozen?

  def eval
    @mutex.synchronize do
      DBC.assert( !empty?, "*** Ne peut pas etre execute sur SVar vide" )
      return unless frozen?

      @state = :in_evaluation
      Thread.new { value }
    end
  end

  def then
    SVar.new( :write_once, :async ) { yield(value) }
  end
end

#
```

```
  class SVarWritable < SVar
    def value=( v )
      @mutex.synchronize do
        fail "SVar deja definie: #{self.inspect}" unless empty?

        @value = v
        @state = :full
        @is_full.broadcast # Il peut y avoir plusieurs lecteurs bloques!

        v
      end
    end
  end

  class SVarMutable < SVarWritable
    def take
      @mutex.synchronize do
        unfreeze if frozen?
        @is_full.wait( @mutex ) until full?

        val = @value
        @value = nil
        @state = :empty

        val
      end
    end

    def mutate!
      @mutex.synchronize do
        unfreeze if frozen?
        @is_full.wait( @mutex ) until full?

        @value = yield(@value)

        @value
      end
    end
  end
end
```

Quelques erreurs rencontrées à plusieurs reprises

```ruby
# a. Quel est le probleme?
def value=( v )
  @mutex.synchronize do
    fail "SVar deja definie: #{self.inspect}" unless empty?

    @value = v
    @state = :full
    @is_full.signal

    v
  end
end

# b. Quel est le probleme?
def value=( v )
  fail "SVar deja definie: #{self.inspect}" unless empty?

  @mutex.synchronize do
    @value = v
    @state = :full
    @is_full.broadcast

    v
  end
end

# c. Quel est le probleme?
def eval
  DBC.assert( !empty?, "*** Ne peut pas etre execute sur SVar vide" )
  return unless frozen?

  @mutex.synchronize do
    @state = :in_evaluation
    Thread.new { value }
  end
end
```

Donc : Sauf lors d'une utilisation "immédiate" de sa valeur, tout accès à une variable partagée doit être protégée par un verrou... sinon il y a danger de *situation de compétition* ☹

Question : L'acquisition du verrou est-il nécessaire ou utile dans ce segment de code?

```ruby
def empty?
  @mutex.synchronize { @state == :empty }
end
```

Extraits du fichier `wavefront.rb`

```
def self.run_par( n )
  m = Matrice.new( n, n ) { SVar.new }

  m[0, 0].value = 1
  (1...n).peach do |k|
    m[k, 0].value = 1
    m[0, k].value = 1
  end

  (1...n).peach do |i|
    (1...n).peach do |j|
      m[i, j].value = m[i-1, j].value + m[i-1, j-1].value + m[i, j-1].value
    end
  end

  m.to_a.map { |l| l.map!(&:value) }
end
```

Extraits du fichier graphe.rb

```
class NoeudPar < Noeud
  def initialize( id, val, graphe, voisins )
    super
    @marque = SVar.new(:mutable)
    @marque.value = false
  end

  # Indice pour optimisation possible: Double-check!
  def marquer!
    return true if @marque.value

    marque = @marque.take
    @marque.value = true

    marque
  end

  def reset_marque
    @marque.take
    @marque.value = false
  end

  def somme
    if marquer!
      0
    else
      @val +
        @voisins.map { |v| @graphe[v] }
        .pmap(dynamic: true, &:somme)
        .preduce(0, &:+)
    end
  end
end
```

Extraits du fichier `services.rb`

```ruby
class TraiterRequeteThread
  extend ServicesExternes

  def self.run( produit, qte_desiree, id_usager )
    prix_qte, fournisseur = FOURNISSEURS
      .map { |f| Thread.new { externes.prix_et_qte_disponible( f, produit, qte_d
      .map(&:value)
      .select { |prix, qte| qte >= qte_desiree }
      .each_with_index.min_by { |x| x.first.first }

    mutex = Mutex.new
    verification_faite = ConditionVariable.new
    agence = nil
    AGENCES.each do |a|
      Thread.new do
        if externes.paiement_ok?(a, id_usager, prix_qte.first * prix_qte.last)
          mutex.synchronize do
            unless agence
              agence = a
              verification_faite.signal
            end
          end
        end
      end
    end

    mutex.synchronize { verification_faite.wait(mutex) until agence }

    [fournisseur, prix_qte.first, agence]
  end
end


#
```

```
class TraiterRequeteSVar
  extend ServicesExternes

  def self.run( produit, qte_desiree, id_usager )
    prix_et_qtes = FOURNISSEURS.map do |f|
      SVar.new { externes.prix_et_qte_disponible( f, produit, qte_desiree ) }
    end
    prix_qte, fournisseur = SVar
      .all( *prix_et_qtes )
      .value
      .select { |prix, qte| qte >= qte_desiree }
      .each_with_index.min_by { |x| x.first.first }

    oks = AGENCES.map do |a|
      SVar.new { a if externes.paiement_ok?(a, id_usager, prix_qte.first * prix_
    end
    agence = SVar.any( *oks ).value

    [fournisseur, prix_qte.first, agence]
  end
end
```