Solutions aux exercices du chapitre 3 : Concepts de base

Automne 2017

Exercice 3.1 Un seul doigt suffit-il vraiment?

Est-ce vrai qu'un seul doigt suffit pour indiquer à quel endroit on est rendu dans l'exécution d'un programme séquentiel?

Solution :

Non, pas vraiment, à cause des appels de méthodes. En fait, on a besoin d'une «pile de doigts», et à chaque instant un seul doigt suffit, celui au sommet de la pile, pour nous dire où on est actuellement rendu.

Dans un programme concurrent, c'est aussi vrai, on a plusieurs piles d'activation des méthodes, une pour chaque *thread*.

Exercice 3.2 Effet d'ajouter des coeurs sur des programmes *CPU-bound* ou *IO-bound*.

Soit deux machines multi-coeurs M_1 et M_8, ayant des configurations semblables sauf pour le nombre de coeurs : un seul (1) coeur pour M_1 et huit (8) coeurs pour M_8.

1. Soit un programme P_1 qui est *IO-bound* et qui s'exécute sur M_1 en 1.6 secondes.

   Si on exécute P_1 sur M_8, quel temps d'exécution peut-on s'attendre à obtenir?

2. Soit un programme P_2 qui est *CPU-bound* et qui s'exécute sur M_1 est 1.6 secondes.

   Si on exécute P_2 sur M_8, quel temps d'exécution peut-on s'attendre à obtenir?

Solution :

1. Le programme P_1 étant *IO-bound*, l'ajout de coeurs risque d'avoir peu d'effet, sinon aucun, donc le temps d'exécution sera très près de 1.6 secondes.

2. Si le problème traité par le programme P_2 *peut être décomposé en sous-problèmes relativement* **indépendants**, alors le temps d'exécution pourrait être très près de 0.2 secondes.

Exercice 3.3 Parallélisation de la recherche de motifs dans un fichier.

Soit l'algorithme suivant, où l'on suppose que la ligne retournée est `EOF` lorsque la fin de fichier est atteinte :

```
PROCEDURE trouver_motif( fich, motif )
DEBUT
  ouvrir le fichier fich
  ligne ← lire une ligne du fichier fich
  TANTQUE ligne ≠ EOF FAIRE
    ecrire ligne SI motif est present dans ligne
    ligne ← lire une ligne du fichier fich
  FIN
FIN
```

Est-il possible de paralléliser la procédure `trouver_motif`, c'est-à-dire, de faire en sorte que certaines tâches à l'intérieur de la procédure s'exécutent de façon concurrente?

On désire évidemment que les lignes du fichiers soient lues dans le bon ordre, et émises dans le bon ordre si elles satisfont le motif.

Indice : Introduire une variable auxiliaire...

Solution :

Solution avec *double buffering* : on utilise une variable pour obtenir/conserver le résultat de la lecture et une autre variable pour contenir la ligne à analyser, ce qui permet à la lecture et à la recherche de se faire de façon concurrente.

```
PROCEDURE trouver_motif( fich, motif )
DEBUT
  ouvrir le fichier fich
  ligne1 ← lire une ligne du fichier fich
  TANTQUE ligne1 ≠ EOF FAIRE
    EN_PARALLELE
      ecrire ligne1 SI motif est present dans ligne1
      //
      ligne2 ← lire une ligne du fichier fich
    FIN
    ligne1 ← ligne2
  FIN
FIN
```

Exercice 3.4 Recherche parallèle de l'élément maximum d'un tableau

Soit le segment de code Ruby suivant qui trouve l'élément maximum parmi un tableau de nombres entiers positifs :

```
a = [10, 62, 173, 823, 32, 99, 9292, 0, 1]

m = 0
PRuby.pcall( 0...a.size,
  ->( i ) { m = a[i] if a[i] > m }
)

puts "maximum = #{m}"
```

1. Est-ce que ce programme est correct? Justifiez votre réponse.
2. S'il n'est pas correct, comment peut-on le rendre correct?

Solution :

1. Non, pas correct. Voici des résultats d'exécution obtenus en introduisant `jiggle` juste avant «`m = a[i]`» :

```
$ ruby maximum.rb
maximum = 9292
$ ruby maximum.rb
maximum = 9292
$ ruby maximum.rb
maximum = 99
$ ruby maximum.rb
maximum = 23
$ ruby maximum.rb
maximum = 12
$ ruby maximum.rb
maximum = 10
```

2. Première «solution» :

```
mutex = Mutex.new
PRuby.pcall (0...a.size),
  ->( i ) { mutex.synchronize { m = a[i] if a[i] > m } }
```

   Sauf que le programme devient alors tout à fait… **séquentiel** ☹

3. Deuxième solution, avec *double check* :

```
mutex = Mutex.new
PRuby.pcall( 0...a.size,
 ->( i ) { if a[i] > m
             mutex.synchronize { m = a[i] if a[i] > m }
           end
         }
)
```

Exercice 3.5 Qu'est-ce qui sera imprimé par ce programme?

```
#!/usr/bin/env ruby
#
# Petit programme illustrant certaines
# interactions entre threads et reallocation
# dynamique de la taille d'un tableau.
#

NB = 20

loop do
  a = Array.new

  futures = []
  (0...NB).each do |i|
    futures << PRuby.future { a[i] = 0 }
  end

  futures.map(&:value)

  puts a.reduce(&:+)
end
```

Solution :

```
0
0
0
.
.
.
0
0
0
NoMethodError: undefined method `+' for nil:NilClass
    each at org/jruby/RubyArray.java:1613
  inject at org/jruby/RubyEnumerable.java:860
  (root) at bogue-array-threads.rb:28
    loop at org/jruby/RubyKernel.java:1501
  (root) at bogue-array-threads.rb:16
```
