Université Claude Bernard Lyon 1

Programmation parallèle

Nicolas Pronost

LIRIS

Principe

- On utilise la programmation parallèle lorsque l'on veut exécuter plusieurs programmes « en même temps »
- Il n'y a jamais vraiment de « en même temps » dans un ordinateur
- D'où « en parallèle »
- C'est-à-dire que les programmes seront exécutés par bouts, chacun son tour

Notre exemple

- Nous prendrons l'exemple du code suivant, qui s'exécute pour l'instant de façon « linéaire »

```
import random
import time

i = 0
while i < 10:
    print("1")
    attente = 0.2 + random.randint(1, 60) / 100
    time.sleep(attente)
    i += 1
```

- ce code imprime une dizaine de « 1 » à des intervalles de temps entre 0.2 et 0.8 secondes

Les threads

- Les threads sont les morceaux de programme qui peuvent s'exécuter en parallèle
- Notre but est de créer deux threads qui s'exécuteront en parallèle
  - un qui affichera des « 1 »
  - un qui affichera des « 2 »

Classe et thread

- Pour qu'un morceau de programme puisse s'exécuter en parallèle d'un autre, il faut l'inclure dans une classe qui hérite de la classe Python **Thread** du module **threading**
- Le **constructeur** de cette classe initialise les informations nécessaire à l'exécution parallèle
- La procédure membre **run** contient le morceau de programme à exécuter

Classe et thread

- Sur notre exemple, nous créons une classe **AfficheurLettre**
  - le constructeur stocke la lettre à écrire
  - la procédure **run** contient le code précédent

```
import random
import time
from threading import Thread

class AfficheurLettre (Thread):

   def __init__(self,lettre):
      Thread.__init__(self)
      self.lettre = lettre

   def run(self):
      i = 0
      while i < 10:
         print(self.lettre)
         attente = 0.2 + random.randint(1, 60) / 100
         time.sleep(attente)
         i += 1
```

Lancement des threads

- Il ne reste plus qu'à lancer l'exécution de deux instances de cette classe, une avec la lettre « 1 » et l'autre avec la lettre « 2 »
- D'abord on créé les instances de classe

```
afficheur1 = AfficheurLettre("1")
afficheur2 = AfficheurLettre("2")
```

- Rien ne se passe tant qu'on ne demande pas explicitement de les exécuter

Lancement des threads

- La procédure membre **start()** de la classe Thread appelle les procédures membres **run** de chaque classe fille

```
afficheur1.start()
afficheur2.start()
```

- L'appel à **start** sur afficheur1 n'attend pas la fin de l'exécution de sa procédure **run** pour lancer **start** sur afficheur2 ☺ !
- L'appel à **start** créé un nouveau thread système ayant comme code le code de la fonction **run**
- Mais de la même manière l'appel à **start** sur afficheur2 n'attend pas la fin de l'exécution de sa procédure **run** pour continuer d'exécuter le code du programme (ex. principal)

Attente de la fin des threads

- Mais on peut imaginer que la suite du programme suppose que les deux afficheurs ont fini leur affichage
- Il faut attendre qu'ils aient fini d'être exécutés pour continuer le programme
  - si le programme appelant se termine avant les threads qu'il a lancé, les threads seront interrompus brutalement
- Afin d'attendre la fin de l'exécution d'un thread on utilise la procédure **join**

```
afficheur1.join()
afficheur2.join()
```

  - ici, on attend la fin de afficheur1 avant celle de afficheur2
  - si afficheur1 se termine en premier, on attendra la fin de afficheur2 dans le deuxième **join**
  - si afficheur2 se termine en premier, on continue d'attendre la fin de afficheur1 d'abord et le second **join** n'attendra pas du tout

Résumé sur l'exemple

```
import random
import time
from threading import Thread

class AfficheurLettre (Thread):

   def __init__(self,lettre):
      Thread.__init__(self)
      self.lettre = lettre

   def run(self):
      i = 0
      while i < 10:
         print(self.lettre)
         attente = 0.2 + random.randint(1, 60) / 100
         time.sleep(attente)
         i += 1

afficheur1 = AfficheurLettre("1")
afficheur2 = AfficheurLettre("2")

afficheur1.start()
afficheur2.start()

afficheur1.join()
afficheur2.join()
```

- affiche une séquence de 1 et de 2 à peu près alternés

Opérations concurrentes

- Ici, les seules données nécessaires à chaque thread étaient contenues dans l'instance de classe spécifique au thread : **self.lettre**
- Mais les données peuvent être partagées entre thread
- Et parfois l'ordre d'exécution des instructions est important
- Imaginez une donnée de classe **nbLettresAffichees** qui est incrémentée à chaque affichage

```
class AfficheurLettre (Thread):
   nbLettresAffichees = 0
   def __init__(self,lettre):
      Thread.__init__(self)
      self.lettre = lettre
   def run(self):
      i = 0
      while i < 10:
         print(self.lettre)
         attente = 0.2 + random.randint(1, 60) / 100
         time.sleep(attente)
         i += 1
         nbLettresAffichee += 1
```

Opérations concurrentes

- L'instruction d'incrément réalise trois opérations élémentaires
  1. Récupérer la valeur de **nbLettresAffichees**
  2. Ajouter 1
  3. Affecter la valeur incrémentée à **nbLettresAffichees**
- Comme les threads sont exécutés en parallèle, et qu'une instruction peut être 'interrompue' pour exécuter des opérations d'un autre thread, voici ce qui peut se passer
  - le thread afficheur1 commence à exécuter une première fois l'incrément : il récupère la valeur 0 et ajoute 1 (étapes 1 et 2)
  - le thread afficheur1 est interrompu, et le thread afficheur2 exécute pour sa première fois l'incrément en entier (étapes 1,2,3) : la donnée vaut maintenant 0+1=1
  - le thread afficheur2 est interrompu, afficheur1 reprend et exécute l'étape 3, la donnée vaut 1 au lieu de 2....!

Accès simultané

- Autre exemple de problème, si l'afficheur affiche un texte lettre par lettre au lieu d'une seule lettre

```
class AfficheurTexte (Thread):

   def __init__(self,texte):
      Thread.__init__(self)
      self.texte = texte

   def run(self):
      i = 0
      for lettre in self.texte:
         print(lettre)
         attente = 0.2 + random.randint(1, 60) / 100
         time.sleep(attente)

afficheur1 = AfficheurTexte("Bonjour")
afficheur2 = AfficheurTexte("Au revoir")

afficheur1.start()
afficheur2.start()

afficheur1.join()
afficheur2.join()
```

- ce code affiche les deux textes de façon entremêlée

Synchronisation par verrou

- Pour éviter ces problèmes on peut faire en sorte qu'une partie du code ne s'exécute que si la 'ressource' est libre
  - ressource = donnée partagée ou bout de code non interruptible
- Au début des instructions manipulant une donnée partagée entre threads ou devant s'exécuter sans interruption, on verrouille l'accès aux autres threads
- Si un autre thread tente d'y accéder il est bloqué (mis en attente)
- A la fin des instructions la ressource est libérée
- En Python, la classe **RLock** du module **threading** est utilisée

Synchronisation par verrou

```
from threading import Thread, RLock

class AfficheurTexte (Thread):

   verrou = RLock()

   def __init__(self,texte):
      Thread.__init__(self)
      self.texte = texte

   def run(self):
      i = 0
      with AfficheurTexte.verrou:
         for lettre in self.texte:
            print(lettre)
            attente = 0.2 + random.randint(1, 60) / 100
            time.sleep(attente)

afficheur1 = AfficheurTexte("Bonjour")
afficheur2 = AfficheurTexte("Au revoir")

afficheur1.start()
afficheur2.start()

afficheur1.join()
afficheur2.join()
```

Synchronisation par verrou

```
from threading import Thread, RLock

class AfficheurTexte (Thread):

   verrou = RLock()

   def __init__(self,texte):
      Thread.__init__(self)
      self.texte = texte

   def run(self):
      i = 0
      with AfficheurTexte.verrou:
         for lettre in self.texte:
            print(lettre)
            attente = 0.2 + random.randint(1, 60) / 100
            time.sleep(attente)

afficheur1 = AfficheurTexte("Bonjour")
afficheur2 = AfficheurTexte("Au revoir")

afficheur1.start()
afficheur2.start()

afficheur1.join()
afficheur2.join()
```

Importe la classe RLock implémentant le mécanisme de verrou

Synchronisation par verrou

```
from threading import Thread, RLock

class AfficheurTexte (Thread):

   verrou = RLock()

   def __init__(self,texte):
      Thread.__init__(self)
      self.texte = texte

   def run(self):
      i = 0
      with AfficheurTexte.verrou:
         for lettre in self.texte:
            print(lettre)
            attente = 0.2 + random.randint(1, 60) / 100
            time.sleep(attente)

afficheur1 = AfficheurTexte("Bonjour")
afficheur2 = AfficheurTexte("Au revoir")

afficheur1.start()
afficheur2.start()

afficheur1.join()
afficheur2.join()
```

Créer une instance de la classe RLock comme donnée de la classe, i.e. commune à toutes les instances de la classe AfficheurTexte

Synchronisation par verrou

```
from threading import Thread, RLock

class AfficheurTexte (Thread):

   verrou = RLock()

   def __init__(self,texte):
      Thread.__init__(self)
      self.texte = texte

   def run(self):
      i = 0
      with AfficheurTexte.verrou:
         for lettre in self.texte:
            print(lettre)
            attente = 0.2 + random.randint(1, 60) / 100
            time.sleep(attente)

afficheur1 = AfficheurTexte("Bonjour")
afficheur2 = AfficheurTexte("Au revoir")

afficheur1.start()
afficheur2.start()

afficheur1.join()
afficheur2.join()
```

Ceci verrouille le code contenu dans le **with**

Synchronisation par verrou

```
from threading import Thread, RLock

class AfficheurTexte (Thread):

   verrou = RLock()

   def __init__(self,texte):
      Thread.__init__(self)
      self.texte = texte

   def run(self):
      i = 0
      with AfficheurTexte.verrou:
         for lettre in self.texte:
            print(lettre)
            attente = 0.2 + random.randint(1, 60) / 100
            time.sleep(attente)

afficheur1 = AfficheurTexte("Bonjour")
afficheur2 = AfficheurTexte("Au revoir")

afficheur1.start()
afficheur2.start()

afficheur1.join()
afficheur2.join()
```

afficheur1 est lancé, verrouille le code jusqu'à ce qu'il finisse d'afficher le texte « Bonjour ». Pendant ce temps afficheur2 est bloqué au **with**. Dès que afficheur1 sort du **with**, afficheur2 verrouille le code pour lui et affiche « Au revoir ».