9.5 Programmation concurrente en Python

NSI terminale - JB Duthoit

Définition

La programmation concurrente est un paradigme de programmation tenant compte, dans un programme, de plusieurs contextes d'exécution (threads, processus, tâches) matérialisés par une pile d'exécution (stack) et des données privées

Pourquoi la programmation concurrente ?

La concurrence est indispensable lorsque l'on souhaite écrire des programmes interagissant avec le monde réel ou tirant parti de multiples processeurs (multi-coeurs, clusters, cloud, ...)

Ainsi ,

- **Programmation séquentielle** (ou synchrone) : on attend jusqu'à ce que la tâche se termine pour passer à la suivante.
- **Programmation concurrente** (ou asynchrone) : on lance la tâche et on passe à la suivante avant qu'elle se termine.

Exemple

- Serveur web ou ftp : chaque client est géré par un thread
- Browser : un thread gère une connexion, un thread fait le rendu de la page, un thread décode une vidéo, un thread gère l'interaction avec l'interface (tout ceci pour un seul onglet)
- Jeu vidéo : un thread s'occupe du rendu graphique, un autre de l'IA, un autre du son, n threads gèrent n joypads, etc.

Définition

Un **tread** est un mini-processus démarré par un processus et s'exécutant de manière concurrente avec le reste du programme.

On utilise pour cela le module **threading** de la bibliothèque Python.

9.5.1 Un premier programme

Recopier et exécuter ce programme :

```
from time import *
from threading import *

def hello(n):
    for i in range(5):
```

```
        print(f"je suis le thread {n} et ma valeur est {i}")
        sleep(0.5)
    print(f'---fin du thread {n}')

th = []
for n in range(4):
    t = Thread(target = hello, args=[n])
    t.start() #On démarre le thread
    th.append(t)

for t in th:
    t.join() #On attend que le thread s'arrête
```

☛ Deux exécutions successives donnent deux affichages différents.
☛ L'ordre dans lequel sont démarré les threads ne donne pas d'indications sur l'ordre dans lequel ils se terminent...

9.5.2 Attention aux threads

Un premier programme sans surprise :

```
compteur = 0 # Variable globale
limite = 1000000

def calcul():
    global compteur
    for c in range(limite):
        temp = compteur
        # simule un traitement nécessitant des calculs
        compteur = temp + 1

calcul()
print(compteur)
```

☛ Sans surprise, on a compteur = 1000000...
Utilisons maintenant le threading :

```
compteur = 0 # Variable globale
limite = 1000000

def calcul():
    global compteur
    for c in range(limite):
        temp = compteur
        # simule un traitement nécessitant des calculs
        compteur = temp + 1


th = []
compteur = 0
for i in range(4): # Lance en parallèle 4 exécutions dd calcul
    p = Thread(target = calcul, args= [])
```

```
    p.start()       # Lance calcul dans un processus léger à part.
    th.append(p)

for t in th:
    t.join()


    print(compteur)
```

☛ On pourrait s'attendre à avoir *compteur* $= 4 \times 1000000$, soit 4000000..mais non!! On obtient des résultats inférieurs à 4000000 et différents en fonctions des exécutions...

Pour expliquer ce phénomène, imaginons une situation :

1. un processus t_0 est en exécution avec `compteur = 42`. t_0 exécute ensuite l'instruction `temp = compteur`, donc la variable locale `temp` de t_0 contient la valeur `42`. Juste après, le processus est préempté.

2. C'est un autre processus, t_1, qui prend maintenant la main en exécutant tout d'abord `temp= compteur` ; la variable local `temp` contient 42. t_1 exécute ensuite `compteur = temp + 1`, et `compteur` contient donc `43`. A ce moment, le processus est préempté

3. Si le thread t_0 reprend la main, il va continuer là où il s'est arrêté ! t_0 exécute donc `compteur = temp + 1`, où la valeur de `temp` est `42`. Le compteur repasse donc à `42` (au lieu de 44 ici) !

9.5.3 Les verrous

Pour corriger ce problème, on utilise des verrous pour garantir l'accès exclusif pour les deux lignes `v = compteur` et `compteur = v + 1`.

Un verrou est un objet que l'on peut essayer d'acquérir ! Pour l'acquérir, il suffit que personne ne détient le verrou.

⚠ Si quelqu'un d'autre détient le verrou, alors on est bloqué :-(

```
compteur = 0 # Variable globale
limite = 1000000

verrou = Lock()

def calcul():
    global compteur
    for c in range(limite):
        verrou.acquire()
        temp = compteur
        # simule un traitement nécessitant des calculs
        compteur = temp + 1
        verrou.release()


th = []
```

```
compteur = 0
for i in range(4): # Lance en parallèle 4 exécutions dd calcul
    p = Thread(target = calcul, args= [])
    p.start()      # Lance calcul dans un processus léger à part.
    th.append(p)

for t in th:
    t.join()
```

9.5.4 Interblocage

```
from threading import *

verrou1 = Lock()
verrou2 = Lock()

def f():
    for i in range(100):
        verrou1.acquire()
        print("section verouillée f1")
        verrou2.acquire()
        print("section verouillée f2")
        verrou2.release()
        verrou1.release()

def g():
    for i in range(100):
        verrou2.acquire()
        print("section verouillée g2")
        verrou1.acquire()
        print("section verouillée g1")
        verrou1.release()
        verrou2.release()

t1 = Thread(target = f, args=[])
t2 = Thread(target = g, args=[])
t1.start()
t2.start()
t1.join()
t2.join()

print("fin")
```