

18.4

Programmation concurrente en Python

NSI TERMINALE - JB DUTHOIT

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.

18.4.1 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}")
    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...

18.4.2 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
print(compteur)
```

- ☛ Sans surprise, on a $\text{compteur} = 1000000\dots$

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 de 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()

```

- On pourrait s'attendre à avoir `compteur = 4 × 1000000`, soit 4000000..mais non !! On obtient des résultats inférieurs à 4000000 et différents en fonction des exécutions...

Pour expliquer ce phénomène , supposons que t_0 soit en exécution avec `compteur = 42`. si t_0 est interrompu juste après avoir exécuté `c = compteur`, alors sa variable local `v` contient la valeur 42. C'est t_1 qui prend maintenant la main en exécutant tout d'abord `v = compteur` suivi de `compteur = v + 1`. La variable local `v` contient 42 et `compteur` contient donc 43. Si le thread t_0 reprend la main, il va continuer là où il s'est arrêté! t_0 exécute donc `compteur = v + 1`, où la valeur de `v` est 42 ! Le compteur repasse donc à 42 (au lieu de 44 ici) !

18.4.3 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 de 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()

```

18.4.4 Interblocage

```

from threading import *

verrou1 = Lock()
verrou2 = Lock()

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

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

```