

9.5

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.

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 du 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 × 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 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()
```

9.5.4 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")

```

9.6

Exercices type bac

NSI TERMINALE - JB DUTHOIT

Exercice type BAC 9.9

► Exercice tiré du sujet Métropole 2021 (candidats libres)

La commande UNIX ps présente un cliché instantané des processus en cours d'exécution. Avec l'option `-eo pid,ppid,stat,command`, cette commande affiche dans l'ordre : l'identifiant du processus PID (process identifier), le PPID (parent process identifier), l'état STAT et le nom de la commande à l'origine du processus.

Les valeurs du champ STAT indique l'état des processus :

R : processus en cours d'exécution S : processus endormi

Sur un ordinateur, on exécute la commande `ps -eo pid,ppid,stat,command` et on obtient un affichage dont on donne ci-dessous un extrait.

```
$ ps -eo pid,ppid,stat,command
PID  PPID  STAT  COMMAND
1   0   Ss   /sbin/init
...
1912 1908 Ss   Bash
2014 1912 Ss   Bash
1920 1747 S1   Gedit
2013 1912 Ss   Bash
2091 1593 S1   /usr/lib/firefox/firefox
5437 1912 S1   python programme1.py
5440 2013 R    python programme2.py
5450 1912 R+   ps -eo pid,ppid,stat,command
```

À l'aide de cet affichage, répondre aux questions ci-dessous.

- Quel est le nom de la première commande exécutée par le système d'exploitation lors du démarrage ?