Aix*Marseille université

Domaine Sciences et Technologies
DIU "Enseigner l'informatique au lycée"

Bloc 3 : Systèmes d'exploitation
TD/TP 3

Année 2018-19

Synchronisation

1 Threads en Python

Nous allons tout d'abord nous familiariser avec les threads (processus légers) en Python. Les threads sont implantés au sein du module `threading`.

Exercice 1 Nous allons essayer un petit programme pour commencer :

```
from threading import Thread
import time
from random import *


class MonThread(Thread):
    """La classe MonThread"""
    def run(self):
        time.sleep(randint(0,10))
        print("le thread secondaire")


"""Programme principal"""

#Création d'un thread
monthread = MonThread()

#lancement du thread
monthread.start()
time.sleep(randint(0,10))
print("le thread principal")
monthread.join()
```

Question 1.1 Lancer à plusieurs reprises ce programme. Que constatez-vous ?

Essayons maintenant de faire que l'affichage du thread principal soit toujours avant celui du thread secondaire.

Pour cela, nous allons faire partager au thread secondaire et au programme (thread) principal, un variable (un objet).

- Nous allons créer une classe `Declencheur` qui ne contient qu'un booléen. Le constructeur de cette classe positionne ce booléen à faux (le déclencheur est initialement inactif). La classe possède deux autres méthodes `active()` qui positionne le booléen à vrai et la méthode `est_actif()` qui retourne la valeur du booléen.
- Nous allons créer une instance `declencheur` de la classe `Declencheur`. C'est cet objet qui va être partagé. Nous allons étendre la classe `MonThread` afin qu'elle possède comme un attribut valant `declencheur`.
- Pour le thread secondaire, celui-ci va réaliser son affichage et cela fait, va activer le déclencheur.
- Pour le programme principal, une fois le thread secondaire lancé, il va se mettre en attente active (boucle) tant que `declencheur` n'est pas activé. Une fois que cela est fait, le programme principal affiche son message.

Question 1.2 Modifiez votre programme pour réaliser cette implémentation

Notre programme partage une variable : un thread en modifie la valeur (écriture) tandis qu'un autre la lit. Le partage peut s'avérer problématique si plusieurs threads ont pour objectif d'écrire simultanément dans une variable partagée.

Considérons le programme suivant qui utilise un compteur marseillais. Il s'agit d'un compteur qu'on peut incrémenter, dont on peut consulter la valeur et qu'on peut remettre à zéro. Il est marseillais car l'incrémentation se fait de manière "tranquille" (il prend du/son temps).

Le programme principal crée et lance deux threads `Incrementeur` qui, comme leur nom l'indique, incréments le compteur chacun 100 fois puis affiche la valeur du compteur.

```
from threading import Thread

class Compteur():
    def __init__(self):
        self.compteur = 0

    def valeur(self):
        return self.compteur

    def inc(self):
        delai = 0
        borne = 50000
        v = self.compteur
        for i in range(borne):
            delai = delai + 1
        v = v+1
        self.compteur = v

    def raz(self):
        self.compteur = 0



class Incrementeur(Thread):

    def __init__(self,c):
        Thread.__init__(self)
        self.counter = c

    def run(self):
        self.counter.inc()



"""Programme principal"""
moncompteur = Compteur()


for i in range(100):
    monthread1 = Incrementeur(moncompteur)
    monthread2 = Incrementeur(moncompteur)

    monthread1.start()
    monthread2.start()
    monthread1.join()
```

```
monthread2.join()
print(moncompteur.valeur())
```

Question 1.3 Lancer plusieurs fois de suite le programme en changeant ou non la valeur de la variable `borne`. Comment expliquez vous les résultats obtenus ?

Afin d'éviter ce phénomène, nous allons tâcher de faire que l'écriture dans le compteur (son incrémentation) se fasse en section critique. Pour cela, nous allons utiliser un verrou. Ils sont fournis dans Python dans le module `threading`

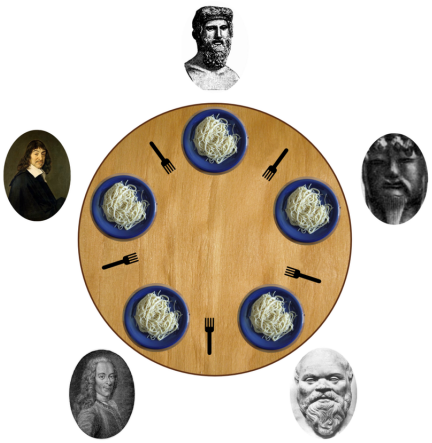
```
from threading import Lock
```

Les verrous possèdent essentiellement deux méthodes :
- `acquire` qui permet d'acquérir le verrou si il est disponible et d'être mis en attente sinon
- `release` qui relâche le verrou

Question 1.4 En utilisant un verrou, rendez atomique l'ensemble de l'opération d'incrémentation et vérifiez en exécutant le programme que le bon nombre d'incrémentation est réalisé.

2 Le diner des philosophes

Autour d'une table se retrouve 5 philosophes [1]. Face à chaque philosophe se trouve un plat de pâtes et chaque philosophe possède une fourchette sur sa droite. Il y a donc 5 fourchettes en tout sur la table. Ceci est illustré ci-dessous :



Chaque philosophe est dans l'un des trois états suivants :
- le philosophe est en pleine réflexion
- le philosophe a faim
- le philosophe mange

La faim va faire sortir le philosophe de sa réflexion. Pour manger le philosophe a besoin de deux fourchettes, celle se trouvant à sa droite et celle se trouvant à sa gauche. Lorsqu'il réfléchit, il n'utilise donc pas de fourchette. Lorsque le philosophe a faim, il va tenter de saisir successivement les deux fourchettes dont il a besoin pour manger. S'il y parvient alors il mange dans le cas contraire, il reste affamé tant qu'il ne peut accéder à ses deux fourchettes. Chaque philosophe reste en réflexion et mange pendant des temps non-déterminés a priori mais fini (un philosophe ne peut réfléchir ou manger indéfiniment). .

Exercice 2

1. Il peut y en avoir plus le nombre importe peu.

Question 2.1 Dans le cas de 5 philosophes, combien au maximum de philosophes peuvent manger en même temps ? Même question lorsqu'il y a n philosophes.

Question 2.2 Existe-t-il une situation d'interblocage, c'est-à-dire une situation dans laquelle plus aucun philosophe ne peut changer d'état ?

Question 2.3 Existe-t-il une possibilité de famine ? Un philosophe peut-il rester indéfiniment dans l'état affamé ? Si oui, donnez un scénario illustrant la situation.