Exercices de TP autour des threads en Python

Nicolas Louvet & Fabien Rico

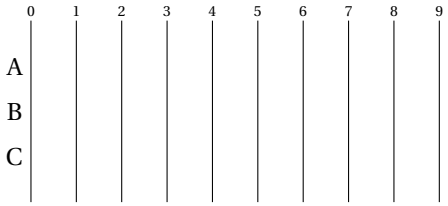
1 Ordonancement

2 Ordonnancement avec liste de priorité

— Le système gère des listes de priorités.
— On se place en mode préemptif c'est à dire que lorsqu'une tâche plus prioritaire arrive, le système lui donne immédiatement accès au processeur.
— Pour des tâches de même priorité, le système utilise le `Round Robin` avec un quantum de 2.
— *Précision :* Pour Round Robin, une tâche qui perd la main perd son tour même si son quantum d'exécution n'était pas terminé.

Tâche	Date(s) d'arrivée(s)	Priorité	Durée	Remarque
A	0	4	3	
B	1	5	2	
C	1	4	4	

1) Dans la figure suivante, faites l'ordonancement sur 8 unités de temps.



3 Programmation avec des threads

Le but est d'expérimenter avec les possibilités proposées par le module standard de Python 3 pour programmer avec des threads. Les exercices proposées sont, comme avec les processus, très scolaires, et permettent de rappeler quelques problèmes classiques : exécution concurrente, accès concurrents à la mémoire, verrous... On utilise le module `os` [1].

On commence par un petit exemple dans lequel on crée deux threads à l'aide du constructeur `Thread()` du module `threading` pour lancer deux threads qui exécuteront chacun la fonction `printer()` pour effectuer des affichages sur la sortie standard. Noter que le constructeur `threading.Thread()` s'utilise essentiellement de la même manière que le constructeur `multiprocessing.Process()` que l'on a utilisé dans le TP sur les processus. Ensuite, les deux threads sont lancés à l'aide de leur méthode `start()`, puis attendus par le processus principal avec leur méthode `join()` : il faut bien entendu à chaque fois un appel par thread.

```
import threading

def printer(id, msg):
    print("thread {}: starting".format(id))
    for i in range(10):
        print(msg)
    print("thread {}: finishing".format(id))

if __name__ == "__main__":
    th1 = threading.Thread(target=printer, args=(1,"pif"))
    th2 = threading.Thread(target=printer, args=(2,"paf"))
    th1.start()
    th2.start()
    th1.join()
    th2.join()
```

Une autre manière de manipuler des threads est de les « emballer » dans une classe dérivant de la classe `threading.Thread()`, et dans ce cas c'est la méthode `run()` de la classe qui sera invoquée quand on lancera le thread. Voici un exemple.

```
import threading, time

class Printer(threading.Thread):
    def __init__(self, id, msg):
        threading.Thread.__init__(self)
        self.id = id
        self.msg = msg

    def run(self):
        print("thread {}: starting".format(self.id))
        for i in range(10):
            print(self.msg)
        print("thread {}: finishing".format(self.id))

if __name__ == "__main__":
    th1 = Printer(1, "pif")
    th2 = Printer(2, "paf")
    th1.start()
    th2.start()
    th1.join()
    th2.join()
```

Notez que, dans le TP sur les processus, on aurait déjà pu travailler de cette manière, en « emballant [1] » les processus dans des classes dérivant de la classe `multiprocessing.Process()` ; ça ne peut pas vous faire de mal de voir plusieurs façons de travailler !

4 Des exercices !

EXERCICE 1 ▶ **Un compteur**
Vouz devez écrire un programme dans lequel un compteur initialisé à 0 est partagé entre deux threads ; ce programme doit mettre en évidence le problème des lectures-écritures concurrentes d'une variable partagée entre plusieurs threads. Une fois lancé, chaque thread incrémente dix fois le compteur, et après que les threads soient terminés, le processus principal affiche la valeur du compteur.

1) Précisément, quel problème d'accès concurrent peut se poser ? Dans quel cas détecterez vous qu'il y a bien eu des accès concurrents au compteur ?

2) Ecrivez votre programme. Il est très possible que vous n'observiez pas le problème d'accès concurrent avec votre première version. Voici des suggestions pour pouvoir mettre en évidence le problème :
- Assurez vous que votre compteur est bien global ou de type mutable !
- Décomposer l'incrémentation en 3 phases : copie de la valeur du compteur dans une variable locale, incrémentation de la variable locale, copie du résultat dans le compteur.
- Obliger les threads à passer « un peu de temps » avant d'écrire leur résultat dans la variable partagée, par exemple en leur faisant afficher le résultat de l'incrémentation.

3) Maintenant que l'on a bien mis en évidence le problème de l'accès concurrent à une variable partagée entre deux threads, on veut le régler ! Construisez une solution à l'aide de la classe `threading.Lock` (voir la documentation ici ; jetez un oeil aussi au *context management protocol* par là)

EXERCICE 2 ▶ **Gestion de comptes banquaires** Un programmeur veut mettre au point un système de gestion de compte bancaire distribué : dans un tel système, différentes agences doivent pouvoir effectuer des opérations de transfert d'argent simultanément. Par exemple, l'agence de Nancy effectue

```
N1 : cur = get_account(1867A)
N2 : new = cur + 1000
N3 : set_account(1867A, new)
```

pendant que l'agence de Karlsruhe fait

```
K1 : cur = get_account(1527B)
K2 : new = cur - 1000
K3 : set_account(1527B, cur)
```

1) En supposant qu'aucune mesure n'est prise pour protéger l'accès concurrent aux données, indiquez les différentes valeurs possibles pour les comptes `1867A` et `1527B` au terme des deux opérations précédentes.

1. Il faut parler plutot d'encapsulation pour briller en société…

2) Heureusement, notre programmeur est parfaitement au courant qu'il est possible d'utiliser des verrous pour protéger l'accès aux données. Afin de tester sa solution avant de la déployer, il décide de la programmer en python en utilisant des threads pour simuler une opération de vas-et-vient entre deux agences. Il utilise pour cela le programme suivant :

```
import threading, random, time

class Account():
    def __init__(self, val = 0):
        self.value = val
        self.account_lock = threading.Lock()

    def lock(self):
        self.account_lock.acquire()

    def unlock(self):
        self.account_lock.release()

    def get_value(self):
        return self.value

    def set_value(self, val):
        self.value = val

def transfert(src, dst, amount):
    src.lock()
    t = src.get_value();
    t -= amount
    src.set_value(t)
    print("New src value =", src.get_value())
    src.unlock()

    dst.lock()
    t = dst.get_value();
    t += amount
    dst.set_value(t)
    print("New dst value =", dst.get_value())
    dst.unlock()

class TransfertTest(threading.Thread):
    def __init__(self, id, src, dst):
        threading.Thread.__init__(self)
        self.id = id
        self.src = src
        self.dst = dst

    def run(self):
        print("thread {}: starting".format(self.id))
        for i in range(10):
            transfert(self.src, self.dst, 500)
        print("thread {}: finishing".format(self.id))

if __name__ == "__main__":
    accountA = Account(10000)
    accountB = Account(10000)
    threads = [TransfertTest(1, accountA, accountB), TransfertTest(2, accountB, accountA)]
    for th in threads: th.start()
    for th in threads: th.join()
    print("accountA -> {}, accountB -> {}".format(accountA.get_value(), accountB.get_value()))
```

En supposant que l'exécution du programme se termine, quelle sera la valeur finale des comptes `accountA` et `accountB` ?

3) Le programmeur se rend compte que l'exécution du programme reste généralement bloquée : expliquez pourquoi, et donnez une solution.

5 Trois flocons !

Le petit Nicolas a écrit un programme utilisant le module `turtle` fourni avec Python pour dessiner trois flocons de Koch. Il a aussi utilisé le module `tkinter` pour mettre ces dessins dans une fenêtre graphique.

```
import tkinter, turtle

def floc(t, l):
    if l<3:
        t.fd(l)
        return
    floc(t, l / 3)
    t.lt(60)
    floc(t, l / 3)
    t.rt(120)
    floc(t, l / 3)
    t.lt(60)
    floc(t, l / 3)

def flocon(t, l):
    t.speed(0)
    t.color('#0000ff', '#55ffff')
    t.begin_fill()
    for i in range(3):
        floc(t, l)
        t.rt(120)
    t.end_fill()

root = tkinter.Tk()
canvas = tkinter.Canvas(root, width=800, height=600)
canvas.pack()
screen = turtle.TurtleScreen(canvas)
screen.bgcolor("#b0ffff")

t1 = turtle.RawTurtle(screen)
t1.up(); t1.goto(-300, -50); t1.down()
flocon(t1, 100)

t2 = turtle.RawTurtle(screen)
t2.up(); t2.goto(200, -50); t2.down()
flocon(t2, 100)

t3 = turtle.RawTurtle(screen)
t3.up(); t3.goto(-50, 150); t3.down()
flocon(t3, 100)

tkinter.mainloop()
```

Nicolas est très satisfait par l'effet produit, mais il trouve un peu dommage que les trois tortues mobilisées pour dessiner les trois flocons ne travaillent pas en même temps à l'écran… Pourriez-vous l'aider ?

Vous commencerez par tenter de faire travailler les trois tortues indépendamment à l'aide de trois threads : vous devriez vous apercevoir assez rapidement que l'approche n'est pas viable ; pourquoi ?

Qu'à cela ne tienne ! Vous adapterez ensuite votre programme de façon à ce que vos trois threads envoient les commandes des trois tortues dans une file partagée, et que ce soit le processus principal qui ordonne les mouvements des trois tortues. Vous vous intéresserez pour cela à la documentation du module `queue`.

Références

[1] `https://docs.python.org/3.8/library/threading.html`