Verrous et interblocage en Python

Introduction des threads

Dans ce pseudo-TD, on ne travaillera pas les processus mais sur les threads (thread = fil d'exécution) en Python (pseudo-TD car il s'agît plus ici d'observer et de comprendre que d'agir). Un thread est un processus simplifié. Notamment l'espace mémoire est partagé entre plusieurs thread provenant d'un même programme, ce qui n'est pas le cas entre plusieurs processus.
Sans lancer le programme ci-dessous, quelle doit être la valeur de `compteur` en fin d'exécution ?

In [ ]:
```
from time import sleep

compteur = 0 # Variable globale

def calcul(limite = 100000):
    global compteur
    for i in range(limite):
        temp = compteur
        # simule un traitement nécessitant des calculs
        #sleep(0.000000001)
        compteur = temp + 1

compteur = 0
calcul()
compteur
```

On va tester le code précédent exécuté simultanement par plusieurs threads. Avant exécution du code, **comprenez-le bien et donnez la valeur attendue de `compteur` en fin d'exécution**, suivant le nombre de threads lancés.
Vérifiez en lançant le programme, de préférence sous **idle**, en effet le résultat est moins intéressant sous Jupyter. Ce code est assez lent. S'il l'est trop, redémarrez le noyau et modifiez la ligne `time.sleep...` afin qu'il soit plus rapide. Que constatez-vous ?

In [ ]:
```
#!/usr/bin/env python
# coding: utf-8

"""
    module   : deadlock.py
    projet   : illustration des interblocages

    version  : 1.0
    auteur   : profs NSI
    creation : 18/04/20
    modif    :
"""

import random, time
import threading

def calcul(limite):
    global compteur
    for i in range(limite):
        tmp = compteur
        # simule un traitement nécessitant des calculs
        time.sleep(random.randint(1, 1000) / 10000)
        compteur = tmp + 1


def workflow(nb) :
    global compteur
    mes_threads=[]
    limite = 100
    for i in range(nb):     # Lance en parallèle nb fils d'exécutio
n
        mes_threads.append(threading.Thread(name=i, target=calcul,
args=[limite]))
        #mes_threads[i].setDaemon(True)
        mes_threads[i].start()
        print("thread numéro ",i," ",mes_threads[i]," vivant : ",
              mes_threads[i].is_alive())
    for i in range(nb):
        mes_threads[i].join()         # attend la fin du traitement
        print("thread numéro ",i," ",mes_threads[i]," vivant : ",
              mes_threads[i].is_alive())

    return compteur


compteur = 0     # Variable globale
print(workflow(1))

compteur = 0     # Variable globale
print(workflow(8))
```

Introduction des verrous

Comme on a pu le voir précédemment, le résultat attendu n'est pas retourné avec le multithreading. Pourquoi ? Il se trouve que les threads calculant "en même temps", certains calculs se "recouvrent" les uns les autres. Par exemple, si la variable `compteur` vaut 5, qu'elle est appelé simultanément par les quatre threads pour l'augmenter, chacun des thread va renvoyer 6. Le résultat final des quatre incrémentation sera alors 6 et non 9.

De nombreux développeurs déconseillent leur usage : "threads are evil, don't use them". Cependant ils sont indispensables, par exemple pour gérer les connexions par milliers à un serveur.

Pour pallier au problème précédent, on introduit des verrous (lock), afin de bloquer les ressources utilisées dans la section critique, c'est-à-dire la partie du programme qui exécute les calculs sur les ressources partagées.

In [ ]:
```
#!/usr/bin/env python
# coding: utf-8

"""
    module   : verrous.py
    projet   : illustration des verrous

    version  : 1.0
    auteur   : profs NSI
    creation : 18/04/20
    modif    :
"""

import random, time
import threading

mon_verrou = threading.Lock()

def calcul(limite, verrou = None):
    global compteur
    for i in range(limite):
        mon_verrou.acquire()    # Début de la section critique

        tmp = compteur
        # simule un traitement nécessitant des calculs
        time.sleep(random.randint(1, 100) / 10000)
        if verrou is not None :
            verrou.acquire()
        compteur = tmp + 1
        if verrou is not None :
            verrou.release()

        mon_verrou.release()    # Fin de la section critique

def workflow(nb, limite = 100) :
    global compteur

    verrous_internes = [threading.Lock() for i in range(nb)]

    mes_threads=[]
    for i in range(nb):     # Lance en parallèle nb fils d'exécutio
n
        mes_threads.append(threading.Thread(name=i, target=calcul,
args=[limite]))
        #p[i].setDaemon(True)
        mes_threads[i].start()
        print("thread numéro ",i," ",mes_threads[i]," vivant : ",
              mes_threads[i].is_alive())
    for i in range(nb):
        mes_threads[i].join()         # attend la fin du traitement
        print("thread numéro ",i," ",mes_threads[i]," vivant : ",
              mes_threads[i].is_alive())

    return compteur
```

```
compteur = 0     # Variable globale
print(workflow(4))
```

Interblocage

Le robot Persévérance a "amarsi" le 18 février 2021 (sur Mars bien évidemment). Il est de connaissance commune que son objectif est de trouver des martiens et de communiquer avec eux.
Ce robot dispose de plusieurs modules indépendants, mais partageant des ressources communes. Chaque module utilise les deux ressources.

- Le module de Communication utilise la carte graphique ou la carte son. Il utilise la carte graphique pour reconnaître les martiens, et la carte son pour tenter de communiquer avec eux.
- Le module Riposte utilise la carte son et la carte moteurs. En effet, les martiens de type 1 (cf. remarques ci-après) explosent lorsqu'ils entendent la chanson "Indian Love Call" de Slim Whitman. La carte son permet de diffuser la chanson, la carte moteur permet de faire tourner le pavillon du gramophone.
- Le module de Déplacement utilise la carte graphique et la carte moteurs. la carte graphique pour observer le terrain, et les moteurs pour se déplacer proprement dit.

Faire tourner le programme ci-dessous. Que constatez-vous ? Expliquer.

In [ ]:
```
# Librairie utilisées
from threading import Thread
from threading import Lock
import time
import random as rd

# Les ressources utilisées par le robot
R1_image = Lock()
R2_son = Lock()
R3_moteurs = Lock()
verrous = [(R1_image, "Image") , (R2_son, "Son"), (R3_moteurs, "Mot
eurs")]


def donnees_ress_th(indice, tableau):
    """
    Renvoie le verrou/le thread et son nom
    """
    return tableau[indice][0], tableau[indice][1]

def duTravail(thread, ressource1, ressource2):
    global verrous, fils
    """...encore du travail...
    Simulation de calcul
    @param thread, ressource1/2 : entiers entre 0 et 2, indices des
tableaux
                                verrous et fils. ressource1 != ressourc
e2
    """
    verrou1, nom_verrou1 = donnees_ress_th(ressource1, verrous)
    verrou2, nom_verrou2 = donnees_ress_th(ressource2, verrous)
    fil, nom_fil = donnees_ress_th(thread, fils)

    chaine = nom_fil + " " + str(thread)+ \
          " demande de la ressource " +str(ressource1)+\
          " " + nom_verrou1 + "\n"

    print(chaine)
    verrou1.acquire()
    # Commenter les lignes suivantes pour le print et constate
r...(décommenter le ligne
    # suivante pour cela)
    #"""
    chaine = nom_fil + " " + str(thread)+ \
          " demande de la ressource " +str(ressource2)+\
          " " + nom_verrou2 + "\n"

    print(chaine)
    #"""
    #print fonctionne ou une autre instruction, par exemple :
    #time.sleep(rd.random()/100.0)
    verrou2.acquire()
    time.sleep(rd.random()/100.0)
    verrou1.release()
    verrou2.release()
```

```
    chaine = nom_fil +  " " + str(thread) + " fin , et libération d
es ressources " +\
             str(ressource1) + " " + nom_verrou1 + " et " + str(res
source2) +\
              " " + nom_verrou2 + "\n"
    print(chaine)

def P1_communication():
    """Fonction de communication, embarquée dans le thread P1"""
    while True :
        print("Communication\n")
        duTravail(0, 0, 1)


def P2_riposte():
    """Fonction de déplacement, embarquée dans le thread P2"""
    while True :
        print("Riposte !\n")
        duTravail(1, 1, 2)


def P3_deplacement():
    """Fonction de deplacement, embarquée dans le thread P3"""
    while True :
        print("Déplacement\n" )
        duTravail(2, 2, 0)


# Création des threads
t1 = Thread(target=P1_communication)
t2 = Thread(target=P2_riposte)
t3 = Thread(target=P3_deplacement)
fils = [(t1, "Communication"), (t2, "Déplacement"), (t3, "Ripost
e")]

# Lancement des threads
t1.start()
t2.start()
t3.start()

# Attente de la fin du travail
t1.join()
t2.join()
t3.join()
print("fin de travail\n")
```

Notes pour les vacances sur Mars :

- Les martiens de type 1 sont très intelligents (ils ont un très gros cerveau), méchants et dangeureux. Juste avant utilisation de leur désintégrateur, ils crient "Ne fuyez pas nous sommes vos amis". Source scientifique : Mars Attacks ! - Tim Burton, 1996.
- Les martiens de type 2 sont insupportables, immatériels, peuvent se téléporter n'importe où, parlent toutes les langues, mais uniquement pour faire des commentaires très désagréables. Source scientifique : Martians go Home - Fredric Brown, 1955 (au CDI du lycée).
- Les martiens sont des petits hommes verts, ce n'est pas pour cela :
  - qu'il faut les confondre avec des cactus ;
  - qu'il faut leur faire remarquer qu'en vert sur le fond rouge de Mars, ils ressemblent à un pull de Noël de mauvais goût.
  - Ces deux erreurs peuvent avoir des conséquences très néfastes sur votre santé.

<hr style="color:black; height:1px />

Sources liste NSI : David Salle - Olivier Lécluse - et plus généralement les contributeurs de la liste
frederic.mandon @ ac-montpellier.fr, Lycée Jean Jaurès - Saint Clément de Rivière - France