Interblocage

Nous avons vu dans la partie précédente que les *threads* peuvent être une solution pour accélérer le traitement de données. Cependant cette pratique n'est pas sans risque, en particulier nous risquons un **interblocage** (*deadlock* en anglais).

1. Un exemple de situation problématique avec les threads

Copiez-collez le code suivant dans un programme `deadlock.py` :

```
import threading, time
from random import randint

verrou1 = threading.Lock()
verrou2 = threading.Lock()

def f1() :
    time.sleep(randint(0,100)/100)
    verrou1.acquire()
    print("Zone risquée f1.1")
    verrou2.acquire()
    print("Zone risquée f1.2")
    verrou2.release()
    verrou1.release()

def f2() :
    verrou2.acquire()
    time.sleep(randint(0,100)/100)
    print("Zone risquée f2.1")
    verrou1.acquire()

    print("Zone risquée f2.2")
    verrou1.release()
    verrou2.release()


t1 = threading.Thread(target=f1)
t2 = threading.Thread(target=f2)

t1.start()
t2.start()
t1.join()
t2.join()
```

En analysant le code :

- deux threads `t1` et `t2` utilisent deux ressources `f1` et `f2`, contrôlées par des verrous 1 et 2
- `f1` verrouille (`acquire`) `verrou1`, puis verrouille `verrou2`, avant de les relâcher (`release`) dans l'ordre inverse de leur verrouillage ;
- `f2` verrouille (`acquire`) `verrou2`, puis verrouille `verrou1`, avant de les relâcher (`release`) dans l'ordre inverse de leur verrouillage ;
- l'exécution des codes de `f1` et de `f2` sont conditionné par un déclenchement aléatoire dans le temps.

Quatre cas peuvent alors se produire :

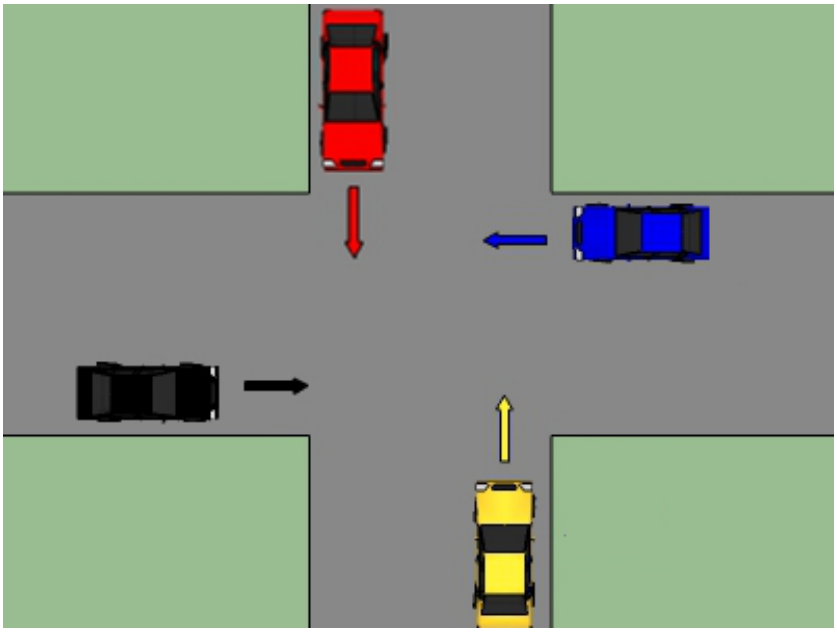
- `t1` démarre en premier, acquiert `verrou1` puis `verrou2`, `t2` est mis en attente de la libération, et peut agir dès que `verrou2` est libéré.
- `t2` démarre en premier, acquiert `verrou2` puis `verrou1`, `t1` est mis en attente de la libération, et peut agir dès que `verrou1` est libéré.
- `t1` démarre en premier et acquiert `verrou1`, `t2` démarre et acquiert `verrou2`. `t1` est alors bloqué, car il ne peut acquérir `verrou2`, et `t2` est dans la même situation avec `verrou1`. Les deux processus sont en attente, et ne pourront pas libérer leurs verrous.

- Il s'agit de la même situation que précédemment, mais `t2` démarre avant `t1`.

Dans les deux derniers cas, nous sommes dans une situation dite d'**interblocage**, les deux *threads* ne pouvant continuer puisqu'ils se bloquent l'un l'autre.

2. Notion d'Interblocage

Les interblocages sont des situations de la vie quotidienne. Un exemple est celui du carrefour avec priorité à droite où chaque véhicule est bloqué car il doit laisser le passage au véhicule à sa droite.



En informatique également, l'interblocage peut se produire lorsque **des processus concurrents s'attendent mutuellement**. Les processus bloqués dans cet état le sont définitivement. Ce scénario catastrophe peut se produire dans un environnement où des ressources sont partagées entre plusieurs processus et l'un d'entre eux détient indéfiniement une ressource nécessaire pour l'autre.

Cette situation d'interblocage a été théorisée par l'informaticien Edward Coffman (1934-) qui a énoncé quatre conditions (appelées conditions de Coffman) menant à l'interblocage :

- **Exclusion mutuelle** : au moins une des ressources du système doit être en accès exclusif.
- **Rétention des ressources** : un processus détient au moins une ressource et requiert une autre ressource détenue par un autre processus
- **Non-préemption** : Seul le détenteur d'une ressource peut la libérer.
- **Attente circulaire** : Chaque processus attend une ressource détenue par un autre processus. P_1 attend une ressource détenue par P_2 qui à son tour attend une ressource détenue par P_3 etc... qui attend une ressource détenue par P_1 ce qui clos la boucle.

Il existe heureusement des stratégies pour éviter ces situations. Nous ne rentrerons pas ici dans ces considérations qui dépassent le cadre du programme.

3. Un exemple historique

Le travail ayant été plus que bien fait par d'autres : le problème Pathfinder