TP Débloquez les tous !

1. Un bloquage à un carrefour

Dans cette situation, chaque voiture doit laisser la priorité à droite…
La situation est donc bloquée !

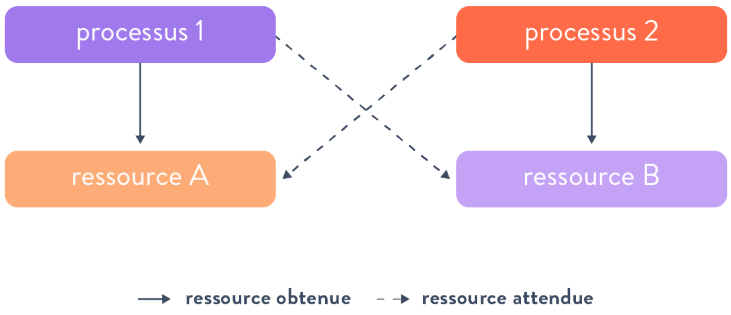
1) A l'aide des documents 1 et 2, représenter sous forme de diagramme situation. On notera **Pb** le processus permettant à la voiture bleu de rouler, **Pj** le processus pour la voiture jaune et ainsi de suite…
2) Proposer une solution pour résoudre le problème de blocage.

Document 1

Les processus d'un système s'exécutent de manière concurrente : leurs exécutions sont entrelacées. De plus, l'ordre dans lequel ils s'exécutent est hors de contrôle car elle est décidée par l'ordonnanceur de processus du système d'exploitation. (…)
Lorsqu'un processus est interrompu, il reprendra son exécution dans l'état où il s'était arrêté. Tant que ce processus manipule des objets lui appartenant à lui seul (par ex. des variables allouées), tout va bien. Mais lorsqu'il accède à des ressources partagées, comme l'accès à un fichier ou un périphérique matériel, alors des problèmes peuvent survenir. (…)
Ces accès concurrents sont problématiques car ils aboutissent à un blocage de l'exécution des processus.

D'après Numérique et Science Informatique, T. Balabonsky, Ellipse, 2020

Document 2



Dans cet exemple, il y a ce que l'on appelle *interblocage* :
Le processus P1 détient la ressource RA et attend la ressource RB pour continuer son exécution.
Le processus P2 détient la ressource RB et attend la ressource RA pour continuer son exécution.

2. Verrous et interblocage

2.1. Le threads en python.

Dans cette partie, on ne travaillera plus sur les processus mais sur les **threads** (*thread* = fil d'exécution) en Python. Il s'agît plus 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.

1) Lancer le programme *thread.py* dans un terminal.

```python
import threading

def hello(n):
    for i in range(5):
        print("je suis le thread", n, "et ma valeur est", i)
    print("--- Fin du thread ", n)

for n in range(5):
    th = threading.Thread(target=hello, args=[n])
    th.start()
```

thread.py

2) Expliquer les lignes :

```
th = threading.Thread(target=hello, args=[n]))
th.start()
```

3) Après plusieurs exécutions, que constate-t-on ?

On s'intéresse maintenant au programme *concurrence.py* qui utilise le *multithreading*. Pour chacun des 4 threads créés, on applique la méthode **join()** : elle permet d'attendre que le thread se termine, et donc s'assure que le compteur ne s'affiche qu'à la fin de tous les threads.

4) L'exécuter plusieurs fois et noter la valeur de COMPTEUR : ce n'est pas 400 000… ?

2.2. 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 appelée simultanément par 4 threads pour l'augmenter, chacun des thread va renvoyer 6. Le résultat final des 4 incrémentations 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.

- Au début du programme, on créé un verrou :

```
verrou = threading.Lock()
```

- Dans la fonction incrément, au début de la fonction on « acquiert » le verrou :

`verrou.acquire()` Le thread courant a la garanti de l'usage exclusif de la ressource.

- Et à la fin de la fonction on « relâche » le verrou : `verrou.release()`

5) Effectuer les modifications et exécuter à nouveau le programme : qu'observe-t-on ?

2.3. Interblocage

Maintenant, on utilise deux verrous dans le programme *`deadlock.py`*

On déclare 2 verrous utilisés de façon symétrique dans **fonction1( )** et **fonction2( )**. La fonction1 essaye d'acquérir *`verrou1`*, puis *`verrou2`*, alors que la fonction2 essaye de les acquérir dans l'ordre inverse.

6) Dans certains cas, il peut y avoir blocage (*interblocage*) : expliquer comment cela peut se produire.