Problèmes liés à l'accès concurrent aux ressources

Les processus se partagent souvent une ou plusieurs ressources, et cela peut poser des problèmes.

Problèmes de synchronisation : illustration avec Python

Exemple d'une variable partagée

Prenons l'exemple d'une variable (= ressource logicielle) partagée entre plusieurs processus. Plus précisément, considérons un programme de jeu multi-joueur dans lequel une variable `nb_pions` représente le nombre de pions disponibles pour tous les joueurs.

Une fonction `prendre_un_pion()` permet de prendre un pion dans le tas commun de pions disponibles, s'il reste au moins un pion évidemment.

On va se mettre dans la situation où il ne reste plus qu'un pion dans le tas commun et on suppose que deux joueurs utilisent la fonction `prendre_un_pion()`, ce qui conduit à la création de deux processus `p1` et `p2`, chacun correspondant à un joueur.

Avec Python, on peut utiliser le module `multiprocessing` pour créer des processus. Le programme Python `pions.py` suivant permet de réaliser la situation de jeu décrite :

```
from multiprocessing import Process, Value
import time

def prendre_un_pion(nombre):
    if nombre.value >= 1:
        time.sleep(0.0005)  # pour simuler un traitement avec des calculs
        temp = nombre.value
        nombre.value = temp - 1  # on décrémente le nombre de pions

if __name__ == '__main__':
    # création de la variable partagée initialisée à 1
    nb_pions = Value('i', 1)
    # on crée deux processus
    p1 = Process(target=prendre_un_pion, args=[nb_pions])
    p2 = Process(target=prendre_un_pion, args=[nb_pions])
    # on démarre les deux processus
    p1.start()
    p2.start()
    # on attend la fin des deux processus
    p1.join()
    p2.join()
    print("nombre final de pions :", nb_pions.value)
```

Explications :

- Le `if __name__ = '__main'__:` permet de ne créer qu'une seule fois les processus `p1` et `p2` qui suivent (c'est nécessaire sous Windows, pas sous GNU/Linux car la création des processus ne se fait pas de la même manière, mais cela reste conseillé ne serait-ce que pour des raisons de compatibilité), on n'en dira pas davantage ici car cela dépasse le niveau de ce cours.
- On a utilisé la classe `Process` du module `multiprocessing` pour instancier deux processus `p1` et `p2`.
    - L'argument `target` est le nom de la fonction qui sera exécutée par le processus : ici les deux processus doivent exécuter la même fonction `prendre_un_pion()`

    - L'argument `args` est une liste des arguments passés à la fonction cible : ici il s'agit de la variable `nb_pions` qui est partagée par les deux processus.
- Par défaut, deux processus ne partagent pas de données en mémoire : on ne peut pas donc pas utiliser `nb_pions` comme une variable globale. Il faut utiliser la classe `Value` du module `multiprocessing` pour créer `nb_pions` dans une mémoire partagée entre les processus. L'argument `'i'` indique que `nb_pions` est un entier (signé) et le deuxième argument est la valeur initiale de la variable, ici 1.
- La fonction `prendre_un_pion()` prend un nombre en paramètre et décrémente sa valeur d'une unité si le nombre est au moins égal à 1.
    - Lors de l'exécution de la fonction par les deux processus, l'argument en question sera l'objet `nb_pions` de la classe `Value` et on accède à sa valeur avec l'attribut `value`.
    - On a ajouté une temporisation permettant de simuler d'autres calculs qui pourraient avoir lieu (par exemple, des instructions de mise à jour du nombre de pions des joueurs)
- Les dernières lignes permettent de démarrer les deux processus et attendre qu'ils soient terminés pour afficher la valeur finale de `nb_pions`.

Si on exécute ce programme, les deux processus `p1` et `p2` sont exécutés et on s'attend au comportement suivant (en supposant qu'il ne reste qu'un seule pion dans le tas commun) :

- l'un des deux est élu en premier, par exemple `p1`, et exécute la fonction `prendre_un_pion()`, le nombre de pions est égal à 1 donc `nb_pions` est décrémenté d'une unité et prend donc la valeur 0, le processus `p1` est terminé ;
- le processus `p2`, qui était en attente, est ensuite élu, et comme le nombre de pions est désormais égal à 0 rien ne se passe et `p2` termine.

Ainsi, le premier joueur a pu prendre le pion restant et le second s'est retrouvé coincé, et la valeur finale de `nb_pions` vaut 0.

Et pourtant, il est tout à fait possible que les choses ne se passent pas ainsi ! En effet, en exécutant plusieurs fois le programme `pions.py` dans un terminal, on obtient parfois une valeur finale égale à 0 et parfois égale à -1 :

```
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions.py
nombre final de pions : -1
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions.py
nombre final de pions : -1
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions.py
nombre final de pions : 0
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions.py
nombre final de pions : -1
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions.py
nombre final de pions : -1
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions.py
nombre final de pions : -1
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions.py
nombre final de pions : 0
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions.py
nombre final de pions : -1
```

C'est un résultat très perturbant non ? Expliquons pourquoi !

Pour cela, on peut ajouter quelques instructions d'affichage pour suivre ce qu'il se passe. On obtient le script `pions_v2.py` suivant :

```
from multiprocessing import Process, Value
import time

def prendre_un_pion(nombre, numero_processus):
    print(f"début du processus {numero_processus}")
    if nombre.value >= 1:
        print(f"processus {numero_processus} : étape A")
        time.sleep(0.0005)  # pour simuler un traitement avec des calculs
        print(f"processus {numero_processus} : étape B")
        temp = nombre.value
        nombre.value = temp - 1  # on décrémente le nombre de pions
    print(f"nb de pions restants à la fin du processus {numero_processus} : {nombre.value}")

if __name__ == '__main__':
    # création de la variable partagée initialisée à 1
    nb_pions = Value('i', 1)
    # on crée deux processus
    p1 = Process(target=prendre_un_pion, args=[nb_pions, 1])
    p2 = Process(target=prendre_un_pion, args=[nb_pions, 2])
    # on démarre les deux processus
    p1.start()
    p2.start()
    # on attend la fin des deux processus
    p1.join()
    p2.join()
    print("nombre final de pions :", nb_pions.value)
```

Explications :

- Lors de la création des processus, on passe un deuxième argument à la fonction `prendre_un_pion()`, le numéro du processus : 1 pour `p1` et 2 pour `p2`.
- Cela permet d'afficher dans cette fonction le numéro du processus à des endroits stratégiques : au début, à l'entrée dans le `if`, juste avant de décrémenter le nombre de pions et à la fin du processus.

En exécutant `pions_v2.py` dans un terminal, on obtient ce genre de choses :

```
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions_v2.py
début du processus 1
début du processus 2
processus 1 : étape A
processus 2 : étape A
processus 1 : étape B
processus 2 : étape B
nombre de pions restants à la fin du processus 2 : -1
nombre de pions restants à la fin du processus 1 : 0
nombre final de pions : -1
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions_v2.py
début du processus 1
processus 1 : étape A
processus 1 : étape B
début du processus 2
nombre de pions restants à la fin du processus 1 : 0
nombre de pions restants à la fin du processus 2 : 0
nombre final de pions : 0
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions_v2.py
début du processus 1
processus 1 : étape A
début du processus 2
processus 1 : étape B
processus 2 : étape A
nombre de pions restants à la fin du processus 1 : 0
processus 2 : étape B
nombre de pions restants à la fin du processus 2 : -1
nombre final de pions : -1
```

Analysons la première exécution du programme.

Heureusement, on peut éviter le problème mis en évidence dans l'exemple précédent.

Comment éviter les problèmes de synchronisation ?

On va utiliser ce qu'on appelle un **verrou** : un verrou est objet partagé entre plusieurs processus mais qui garantit qu'un seul processus accède à une ressource à un instant donné.

Concrètement, un verrou peut être acquis par les différents processus, et le premier à faire la demande acquiert le verrou. Si le verrou est détenu par un autre processus, alors tout autre processus souhaitant l'obtenir est bloqué jusqu'à ce qu'il soit libéré.

Le module `multiprocessing` de Python propose un objet `Lock()` correspondant à un verrou. Deux méthodes sont utilisées :

- la méthode `.acquire()` permet de demander le verrou (le processus faisant la demande est bloqué tant qu'il ne l'a pas obtenu)
- la méthode `.release()` permet de libérer le verrou (il pourra alors être obtenu par un autre processus qui en fait la demande)

On peut alors régler le problème de l'exemple précédent avec le script `pions_v3.py` suivant dans lequel on a laissé les affichages pour bien suivre :

```python
from multiprocessing import Process, Value, Lock
import time

def prendre_un_pion(v, nombre, num_processus):
    print(f"début du processus {num_processus}")
    v.acquire()  # acquisition du verrou
    if nombre.value >= 1:
        print(f"processus {num_processus} : étape A")
        time.sleep(0.0005)
        print(f"processus {num_processus} : étape B")
        temp = nombre.value
        nombre.value = temp - 1
    v.release()  # verrou libéré
    print(f"nb de pions restants à la fin du processus {num_processus} : {nombre.value}")

if __name__ == '__main__':
    # création de la variable partagée initialisée à 1
    nb_pions = Value('i', 1)
    # verrou partagé par les deux processus
    verrou = Lock()
    # on crée deux processus
    p1 = Process(target=prendre_un_pion, args=[verrou, nb_pions, 1])
    p2 = Process(target=prendre_un_pion, args=[verrou, nb_pions, 2])
    # on démarre les deux processus
    p1.start()
    p2.start()
    # on attend la fin des deux processus
    p1.join()
    p2.join()
    print("nombre final de pions :", nb_pions.value)
```

En exécutant (plusieurs fois) ce script dans un terminal on constate que le nombre final de pions est toujours égal à 0.

```
(base) terminale@mounier-01:~/Documents/TNSI/processus$ python pions_v3.py
début du processus 1
processus 1 : étape A
début du processus 2
processus 1 : étape B
nombre de pions restants à la fin du processus 1 : 0
nombre de pions restants à la fin du processus 2 : 0
nombre final de pions : 0
```

Avant de faire le test du `if`, le processus essaye d'acquérir le verrou avec `v.acquire()`. Dès qu'il est acquis, le processus a la garantie qu'il est le seul à pouvoir exécuter le code jusqu'à l'instruction `v.release()`. Cette portion de code protégée s'appelle une *section critique*. Cela ne veut pas dire que le processus détenant le verrou ne peut pas être interrompu (même en section critique), mais les autres processus seront bloqués lorsqu'ils essaieront d'acquérir le même verrou.

Interblocage (deadlock)

Les interblocages (*deadlock* en anglais) sont des situations de la vie quotidienne. L'exemple classique 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 l'**interblocage** peut également se produire lorsque plusieurs processus concurrents s'attendent mutuellement. Ce scénario peut se produire lorsque plusieurs ressources sont partagées par plusieurs processus et l'un d'entre eux possède indéfiniment une ressource nécessaire pour un autre.

Ce phénomène d'*attente circulaire*, où chaque processus attend une ressource détenue par un autre processus, peut être provoquée par l'utilisation de *plusieurs* verrous.

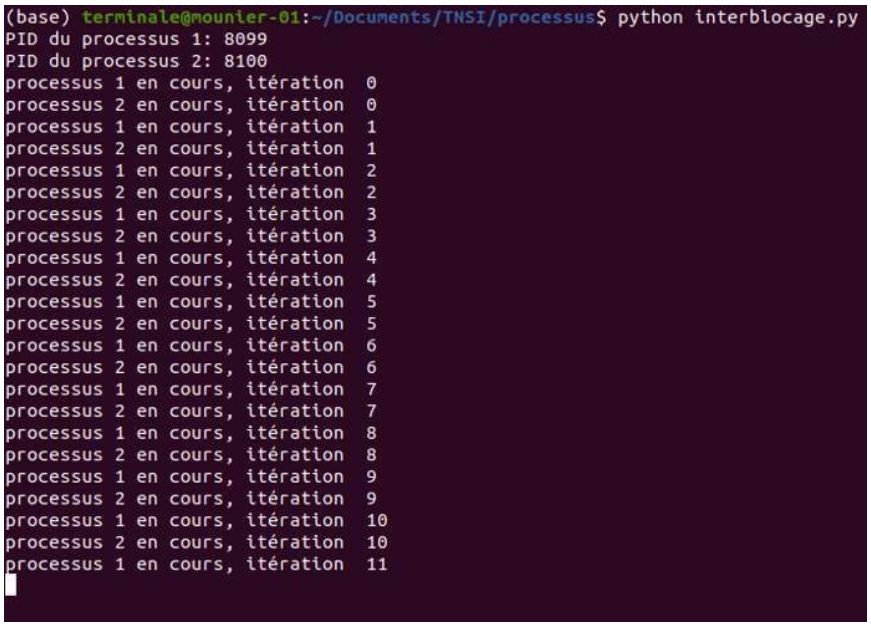
Considérons le script `interblocage.py` suivant dans lequel on a créé deux verrous `v1` et `v2` utilisés par deux fonctions `f1` et `f2` exécutées respectivement par deux processus `p1` et `p2`. Le processus `p1` essaie d'acquérir d'abord `v1` puis `v2` tandis que le processus `p2` essaie de les acquérir dans l'ordre inverse.

```
from multiprocessing import Process, Lock
import time
import os

def f1(v1, v2):
    print("PID du processus 1:", os.getpid())
    for i in range(100):
        time.sleep(0.001)
        v1.acquire()
        v2.acquire()
        print("processus 1 en cours, itération ", i)
        v2.release()
        v1.release()

def f2(v1, v2):
    print("PID du processus 2:", os.getpid())
    for i in range(100):
        time.sleep(0.001)
        v2.acquire()
        v1.acquire()
        print("processus 2 en cours, itération ", i)
        v1.release()
        v2.release()

if __name__ == '__main__':
    # création de deux verrous
    v1 = Lock()
    v2 = Lock()
    # création de deux processus
    p1 = Process(target=f1, args=[v1, v2])
    p2 = Process(target=f2, args=[v1, v2])
    # on démarre les deux processus
    p1.start()
    p2.start()
    # on attend la fin des deux processus
    p1.join()
    p2.join()
```

Si on exécute ce programme, il y a de grandes chances de se retrouver bloqué. Par exemple, dans le cas de l'exécution suivante :

- le processus `p1` est élu : il s'exécute jusqu'à l'acquisition de `v1` mais avant la tentative d'acquisition de `v2`, puis est interrompu
- le processus `p2` est à son tour élu : il s'exécute et acquiert `v2` qui est toujours libre, puis bloque sur l'acquisition de `v1` (qui est détenu par `p1`).
- le processus `p1` reprend la main et bloque sur l'acquisition de `v2` (détenu par `p2`).

Chaque processus détient un verrou et attend l'autre : ils sont en interblocage et l'attente est infinie.



Il n'y a alors pas d'autres choix que d'interrompre les processus en interblocage, par exemple avec la commande `kill`.

Cependant, ce problème a lieu ici car les deux processus essaie d'acquérir les verrous dans l'ordre contraire. Si l'ordre d'acquisition est le même pour les processus, le problème n'a plus lieu (n'hésitez pas à tester !).

De manière générale, dans des problèmes complexes les situations d'interblocage sont difficiles à détecter et il se peut très bien que le programme se comporte bien pendant toute une phase de tests mais bloque lors d'une exécution ultérieure puisque l'on ne peut pas prévoir l'ordonnancement des processus.

> Autre application : lien

Et pour les systèmes multiprocesseurs ?

Les ordinateurs actuels possèdent généralement plusieurs processeurs, ce qui permet à plusieurs processus d'être exécutés parallèlement : un par processeur. Ce parallélisme permet bien évidemment une plus grande puissance de calcul.

Pour répartir les différents processus entre les différents processeurs, on distingue deux approches :

- l'approche *partitionnée* : chaque processeur possède un ordonnanceur particulier et les processus sont répartis entre les différents ordonnanceurs
- l'approche *globale* : un ordonnanceur global est chargé de déterminer la répartition des processus entre les différents processeurs

L'ordonnancement des processus des systèmes d'exploitation actuels est bien plus complexe que les quelques algorithmes évoqués dans ce cours, et cela dépasse largement le cadre du programme de NSI. Si vous souhaitez en savoir plus, voici néanmoins une vidéo intéressante (en français) sur l'ordonnancement du noyau Linux de la chaîne Vitonimal : https://youtu.be/uCGe5WWd1OI.

Bilan

- Un programme en cours d'exécution s'appelle un *processus*. Les systèmes d'exploitation récents permettent d'exécuter plusieurs processus simultanément.
- En réalité, ces processus sont exécutés *à tour de rôle* par le système d'exploitation qui est chargé d'allouer à chacun d'eux les ressources dont il a besoin en termes de mémoire, entrées-sorties ou temps d'accès au processeur, et de s'assurer que les processus ne se gênent pas les uns les autres.
- Au cours de leur vie, les processus varient entre trois états : *élu* si le processus est exécuté par le processeur, *prêt* si le processus est prêt à être exécuté, et *bloqué* si le processus est en attente d'une ressource.
- C'est l'*ordonnanceur* qui est chargé de définir l'ordre dans lequel les processus doivent être exécutés par le processeur. Ce choix se fait grâce à des algorithmes d'ordonnancement.
- Les processus se partagent les différentes ressources, on parle d'*accès concurrent* aux ressources. Ce partage des ressources n'est pas sans risque et peut conduire à des problèmes de synchronisation. Ces problèmes peuvent être évités en utilisant un *verrou*, qui permet à un processus de ne pas être interrompu dans sa section critique par un autre processus demandant le même verrou.
- L'utilisation de plusieurs verrous peut entraîner des *interblocages*, c'est-à-dire des situations où chaque processus attend une ressource détenue par un autre, conduisant à une attente cyclique infinie. L'ordre d'acquisition des verrous est important mais pas toujours évident à écrire dans le cas de problèmes complexes.