Python et Unix : TP4

Préambule :

L'objectif de ce TP est d'exhiber les mécanismes permettant l'interaction entre threads. Nous allons développer un exemple simple : le tunnel.

Exercice 1 : mise en place

Ecrire une fonction `Voiture` qui appelle successivement `EntrerTunnel`, `EtatTunnel` et `SortirTunnel`, ou ces trois fonctions pourront à ce stade se contenter de faire un print.

Ajouter un temps d'attente aléatoire compris entre 0 et 3 secondes l'appel à `EtatTunnel` et celui à `SortirTunnel` représentant le temps de traversée du tunnel.
On utilisera les fonctions du module `random`, *i.e.* `random.random()` *qui rend un* `float` *entre 0 et1, et la fonction* `time.sleep(t)` *qui attends t secondes (t float)*

Créer une liste *tunnel* représentant l'état du tunnel : elle contiendra les identifiants des threads dans le tunnel. L'*id* d'un thread est obtenu avec `thread.get_ident()`.
Compléter le corps de `EntrerTunnel` pour ajouter (append) l'identifiant du thread qui l'appelle à la ressource `tunnel`. De même compléter le corps de `SortirTunnel` pour enlever l'identifiant du thread qui l'appelle de la liste. Enfin écrire une sortie associée a `EtatTunnel` qui affiche : l'identifiant du thread qui appelle cette méthode, le nombre de "voitures" dans le tunnel, et leurs identifiants.

Créer le corps du programme qui instancie 10 ou 20 nouveaux thread sur la fonction Voiture, attends 4 secondes et se termine.

Exécutez le plusieurs fois afin d'observer l' indéterminisme de l'exécution.
[ce code aura des résultats différents en fonction du système d'exploitation et de la machine utilisée]

Exercice 2 : verrous : lock

Le tunnel à une taille limitée, dans une copie du script de la question 1, nous allons ajouter un contrôle d'accès à la ressource partagée tunnel.

Dans un premier temps on interdira l'accès au tunnel si un autre thread s'y trouve. On peut implémenter ce mécanisme à l'aide d'un sémaphore binaire ou verrou (mutex).

Un tel verrou est crée à l'aide de :
`monVerrou = thread.allocate_lock()`

On acquiert le verrou (opération bloquante si le verrou est déjà pris par un autre thread !!) par:
`monVerrou.acquire(1)`

La valeur 1 précise qu'on veut attendre indéfiniment qu'il se libère. La valeur 0 signifie que l'on ne veut pas se bloquer si le verrou est indisponible, i.e :

```
if (monVerrou.acquire(0)) :
        # j'ai le verrou ici, traitement sur la variable partagée
else :
        # sortir une erreur, le verrou n'a pas pu être acquis
```

On relache le verrou à l'aide de :

```
monVerrou.release()
```

Ajouter la gestion du sémaphore pour acquérir le verrou avant de pénetrer dans le tunnel, et le relacher en ressortant.
On pourra réduire le temps de passage à 0-1 secondes et affecter à chaque thread un numéro correspondant à l'ordre de création. A l'affichage, précisez le couple (rang de creation, thread_ident), ou même simplement le rang de création. Prévoyez aussi un affichage ***avant*** de rentrer dans le tunnel (i.e. avant l'appel à acquire(1)) du type : « voiture numero XX veut entrer dans le tunnel »

Assurez vous à l'exécution qu'un seul thread à la fois accède à la ressource.

Exercice 3 : Semaphore

Nous allons à présent supposer que le tunnel est suffisamment grand pour contenir trois voitures simultanément. Pour cela nous allons utiliser un `threading.Semaphore`. A la différence d'un verrou, un sémaphore dispose d'un compteur qui est décrémenté de 1 à chaque acquisition et réincrémenté à chaque release. L'appel à acquire est bloquant si le compteur est à la valeur 0. On initialise la valeur du compteur du sémaphore à la création :

```
monSem = threading.Semaphore(5)
while (1) :
        if (monSem.acquire(0)) :
                print « Ok »
        else :
                print « fini »
                break
monSem.release() …
```

Implémentez cette limite à trois voitures dans le tunnel simultanément à l'aide d'un sémaphore.

Exercice 4 : wait et notify

Il existe également des mécanismes de synchronisations similaire au java avec `wait`, `notify` et `notifyAll`
On instancie un objet Condition :

```
monMonitor = threading.Condition() :
```

On peut acquire et release cet objet, mais l'on dispose aussi de primitives de plus haut niveau. Il faut acquérir l'objet Condition (ou *monitor* en java) avant tout appel aux primitives qui suivent :

monMonitor.wait()
Relache le moniteur, attends (sleep) qu'un autre thread notifie le moniteur que « la voie est libre », et réacquiert le moniteur avant de poursuivre l'éxecution.

nomMonitor.notify()
Reveille un processus (aléatoirement) en attente sur ce moniteur, qui poursuivra son éxecution.

nomMonitor.notifyAll()
Reveille tous les processus en attente sur ce moniteur. Permet de mettre en concurrence les threads pour une ressource, ou d'éviter l'attente active en cas de besoin de synchronisation entre threads.

Toutes ces méthodes ne peuvent être utilisées que si l'on a préalablement acquis (monMonitor.acquire() et monMonitor.release() l'exclusivité sur le moniteur)

Nous allons ajouter un camion au système: le camion est prioritaire sur les voitures, et il est interdit d'avoir des voitures en même temps que le camion dans le tunnel.

Il correspond à un écrivain : dans le modèle lecteurs/écrivains d'accès à une ressource, les lecteurs peuvent accéder simultanément à la ressource, mais l'écriture est en exclusion mutuelle de toute autre opération de lecture ou écriture.

Les voitures (lecteurs) avant d'entrer dans le tunnel (section critique) consultent une variable camionEnAttente booléenne ; si elle est vraie ils se mettent en atente sur un moniteur (Condition) Camion, avant de demander le sémaphore de l'exercice 3 qui limite toujours à trois le nombre de voitures dans le tunnel.
Le camion (écrivain) positionne la variable camionEnAttente booléenne à vrai quand il arrive au tunnel, et acquiert trois fois le sémaphore pour rentrer dans le tunnel.

Ajoutez un temps aléatoire avant l'appel à EntrerTunnel, de l'ordre de 0-3 pour les voitures et 0-1,5 pour le camion.

Adaptez l'affichage pour faire apparaître plus nettement le camion.

Question subsidiaire : (répondre en commentaire dans le source)

Que se passe-t-il s'il on introduit deux écrivains (camions) dans le système ?
Quels problèmes/cas de figure peut-on rencontrer ?
Proposez une solution qui permette l'accès à un seul écrivain ou à plusieurs lecteurs, dans un système ou les demandes de lecture et d'écriture arrivent de façon aléatoire.
Implémentez votre solution, en faisant boucler indéfiniment 12 threads dont deux écrivains, avec un sleep aléatoire avant de demander une lecture/ecriture.