IUT Robert Schuman
Institut universitaire de technologie
Département informatique
Université de Strasbourg

S4x/ Programmation concurrente (en Python)
Série 3 : Exclusion mutuelle & synchronisation

Notions :
- Section critique, exclusion mutuelle, synchronisation
- Programmation objets : construction d'une classe par héritage de la classe *Thread* (du package *threading*)
- Constructeurs du module *threading* : *Lock*(), *Semaphore*(*n*), *BoundedSemaphore*(*n*), *Condition*([*mutex*])
- Le module *multiprocessing* versus le module *threading*

Ressource très instructive sur la synchronisation des *threads* :
`http://www.laurentluce.com/posts/python-threads-synchronization-locks-rlocks-semaphores-conditions-events-and-queues/`

Préambule

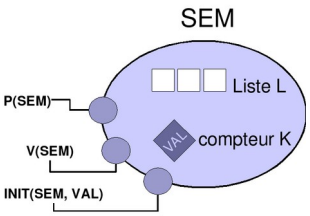
Maître Dijkstra !
Wouaah !

Extraits de l'ouvrage de JM Rifflet [Unix, programmation et communication / Dunod]

Les **sémaphores** dont le concept a été « inventé » par Edsger **Dijkstra** constituent avant tout un mécanisme de synchronisation des processus.
Un sémaphore *S* est une variable à valeurs entières non négatives accessibles au travers de deux opérations particulières :

$\mathcal{P}(S)$: *si* S est nul alors mettre le processus en attente *sinon* $S \leftarrow S - 1$
$\mathcal{V}(S) : S \leftarrow S + 1$; réveiller un (ou plusieurs) processus en attente.

La possibilité de synchroniser des processus en utilisant des sémaphores repose sur :
- l'atomicité des opérations $\mathcal{P}$ et $\mathcal{V}$ précédentes, c'est-à-dire que la suite des opérations les réalisant est non interruptible.
- l'existence d'un mécanisme permettant de mémoriser les demandes de réalisation d'opérations $\mathcal{P}$ non satisfaites afin de permettre le réveil de ces processus.



Les **mutex** : sémaphores binaires, c'est-à-dire ayant deux états (ou valeurs) possibles : un *mutex* peut être soit libre, soit verrouillé.

Les **conditions** : le concept de (variable de) condition, utilisé conjointement avec celui de mutex, permet l'implantation de moniteurs (notion introduite par Hoare). Le contexte général d'utilisation en est le suivant :
- Étant donné une variable *var* de type quelconque, on souhaite attendre que cette variable change d'état et satisfasse une *condition* particulière avant de poursuivre l'exécution du programme : il peut par exemple s'agir d'une file d'attente dont on souhaite attendre qu'elle soit non vide ou d'une variable entière dont on souhaite qu'elle ait une valeur non nulle ou supérieure à une valeur donnée.
- On associe « logiquement » à cette « condition réelle » d'une part un mutex *var_mutex* et d'autre part une variable de condition, *var_cond*. Le *mutex* est utilisé pour assurer la protection des opérations sur la variable *var* elle-même et la variable de condition permet d'en transmettre les changements d'état.

Partie I : Synchronisation dans une partie de *ping pong*, le retour

I.1. `ping_pong_sem.py`

Comme dans l'exercice de la série précédente, il s'agit d'écrire un script avec des threads qui doivent se synchroniser pour afficher *ping pong ping pong ping pong* ... un certain nombre de fois.
On lance N threads *ping* et N threads *pong* ; chaque thread *ping* fait **un** (seul) *ping* et chaque thread *pong* fait **un** (seul) *pong*.

Ici on vous **demande** de réaliser l'exclusion mutuelle et la synchronisation entre les deux sortes de threads en utilisant **deux sémaphores**, au lieu d'utiliser comme dans la série précédente, un *mutex* et une variable « bascule » (`Ping_OK`) qui pouvait prendre les valeurs *False* ou *True*.

ⓘ Sémaphores en *Python*

Dans le module `threading` il existe deux « variétés » de sémaphores : `BoundedSemaphore` et `Semaphore`.
Un `BoundedSemaphore` est un `Semaphore` **contraint** (borné) dont la valeur (un compteur associé au sémaphore) au cours de son existence ne peut pas dépasser sa valeur initiale (>0) ; le nombre de `acquire` doit toujours être ≥ au nombre de `release` !

Exemples illustrant la différence entre un *BoundedSemaphore* et un *Semaphore* :

```
from threading import Thread, BoundedSemaphore, Semaphore

sem1 = BoundedSemaphore(2)        # valeur initiale = 2
sem2 = Semaphore(1)
sem3 = Semaphore(0)               # peut être initialisé à 0 contrairement à un BoundedSemaphore !
```

On suppose qu'un thread 1 effectue les actions suivantes	2ième thread
`sem1.acquire()` # ok ; valeur passe à 1 `sem1.acquire()` # ok ; valeur passe à 0 `sem1.acquire()` # ici, le thread bloque !	
← Ce thread 1 est débloqué par l'action du 2ième thread et peut « poursuivre sa route » car `sem1.acquire()` a réussi ! La valeur du sémaphore redescend à 0 `sem1.release()` # ok `sem1.release()` # ok `sem1.release()` # erreur ! trop de *release* ! (on a un # message d'erreur le signalant) `sem2.acquire()` # ok `sem2.release()` # ok `sem2.release()` # ok `sem3.release()` # ok ! → La valeur init. qui était de 0 (donc sem3 était dans l'état verrouillé) est passée à 1 donc sem3 est maintenant dans l'état non verrouillé	`sem1.release()` # fait "grimper" la valeur de *sem1* à 1 `sem3.acquire()` # bloque ! Le compteur associé vaut 0 ... ... ok, 2ième thread débloqué

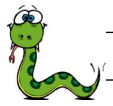
1ère version : codez le programme sans définir de classe, en faisant appel au constructeur de threads :

`Thread(target=fonction, args=(...) )`

2ième version : transformez le programme précédent en définissant deux **classes** `Ping` et `Pong`, sous-classes de la **classe** `Thread` ; donc les différents threads lancés seront des instances des classes `Ping` ou `Pong`.

Et **si vous avez le temps** ...
J'avais indiqué dans la série 2 (Partie III -3), qu'au lieu d'utiliser l'héritage, on pouvait associer une instance d'une classe à un thread en utilisant le principe de **relation** ou de **composition** comme dans les exemples *other_daemon_night_class.py* et *other_daemon_night_class1.py*. N'hésitez pas à coder une version utilisant ce principe !

I.2. ping_pong_cond.py



Je ne supporte plus ces *ping pong* !

Cette fois-ci, on réalise l'exclusion mutuelle et la synchronisation avec deux variables de type *Condition* (dotées du même *lock*/*mutex*) et une variable d'état « bascule » (*Switch*).

🛠 `Condition.notify()`
`Condition.wait()`

Principe : On change la valeur de *Switch* lorsqu'un *ping* (ou un *pong*) a été effectué ; le thread (*ping* ou *pong*) **notifie** alors, via la variable de *Condition*, ce changement d'état à l'un des threads en **attente** de l'autre catégorie.

Si vous êtes curieux, regardez sur le site de Laurent Luce comment sont implémentées les méthodes *put* et *get* de la classe *Queue*
`http://www.laurentluce.com/posts/python-threads-synchronization-locks-rlocks-semaphores-conditions-and-queues`

Ossature de script disponible sur *moodle*, plutôt « bien en chair » vu que cette solution est moins facile à mettre en oeuvre que la précédente ...

```
#!/usr/bin/python3
# ping_pong_cond.py
""" Une partie de ping pong avec des variables <Condition> """

# N objets (threads) Ping et N objets(threads) Pong
```

```
from threading import Thread, Lock, Condition
from random import random
from time import sleep
from sys import argv, stderr

# définition de 2 classes Ping et Pong
class Ping(Thread) :
    def run(self) :
        global Switch
        sleep(random())        # pour espacer aléatoirement les démarrages des threads
        Cond_ping.acquire()
        while Switch == 1 :
            Cond_pong.wait()   # en attente d'une notification de pong
        print("ping ...", end=' ')
        Switch = 1
        Cond_ping.notify()
        Cond_ping.release()

class Pong(Thread) :
    def run(self) :
        global Switch
        sleep(random())        # pour espacer les démarrages des threads
        ...

# main thread
if argv[1:] :    # si liste des paramètres non vide
    N = int(argv[1])
else :
    N = 100      # 100 ping pong par défaut
Switch = 0 # pour bloquer pong au départ
Mutex = Lock()  # Lock commun aux 2 sortes de threads, en paramètre des 2 variables Condition
Cond_ping = Condition(Mutex)
Cond_pong = Condition(Mutex)
...
print("\nFin de partie !")
```

I.3. ping_pong_sem1.py

Variante du script *ping_pong_sem.py* : déléguer l'affichage des *ping* et *pong* à un **thread dédié**, en mode standard (c.a.d. non *daemon*), qui affiche alternativement un *ping* puis un *pong* (avec une courte pause entre les affichages).
Les threads *ping* et *pong* communiquent avec le thread d'affichage via une structure *Queue* (cf série *Threads*).

🛠

```
class Affiche(Thread):
    def run(self):
        # global Fifo                  # Fifo pourrait aussi être en paramètre de la méthode __init__
        item = Fifo.get()      # get bloquant
        while not (item is None):      # while item is not None
            print(item)
            sleep(0.1)                 # courte pause (vous pouvez augmenter la durée si ça vous détend ;-)
            item = Fifo.get()  # get bloquant
```

Rappel (cf exercice II.1 de la série précédente) : ne pas oublier de déposer dans *Fifo* une valeur « sentinelle » (indicateur de fin) pour que le thread d'affichage puisse se terminer !
Mais qui (quel thread ou *main* thread) envoie cette valeur « sentinelle » dans *Fifo* ? Et où (dans le code) ? ...

Hum hum ...

I.4. ping_pong_sem2.py ... ou le retour de **Beastie** !

Encore une variante du script *ping_pong_sem.py* : cette fois-ci, on délègue l'affichage des *ping* et *pong* à un thread « tournant » en arrière-plan, c'est à dire en mode **daemon**, qui affiche alternativement un *ping* et un *pong* (avec une courte pause entre les affichages).

🛠


```
class Affiche(Thread):
    def __init__(self):
        Thread.__init__(self)
        self.setDaemon(True) # self.daemon=True

    def run(self):
        while True :           # boucle infinie (assez habituelle pour un thread daemon)
            ...                # lire dans la Queue (de nom Fifo) et afficher
            Fifo.task_done()
```

```
        sleep(0.1)
```

Rem. : D'une manière générale, il n'y a pas besoin de faire « sortir de la boucle infinie » un thread en arrière-plan..
Le *main* thread peut se terminer (n'est pas bloqué) même si un *daemon* « tourne » encore. Et la fin du programme entraîne automatiquement l'arrêt du *daemon* (cf les exemples à la fin de la série précédente).
Toutefois, ici, il faut trouver un moyen de faire attendre le *main*, le temps nécessaire au thread d'affichage de lister tous les résultats et donc que *Fifo* ne contienne plus d'item (cf la pause entre deux affichages qui le retarde)

Partie II : Série « L'Igloo Bar » (ou le IGlou Glou Bar si vous préférez ;-)

Dans cette partie, nous allons d'abord « affronter » un problème classique de **limitation des accès à une ressource critique** (ici en l'occurrence, un bar d'une certaine capacité C limitée par le nombre de sièges).
En effet, pour des raisons de confort, Anouk et Amarok Igloopuk (d'origine eskimo/inuit !) , les propriétaires du bar, souhaitent limiter le nombre de clients (threads) concurrents qui peuvent s'installer et consommer « en même temps ».

Présentation

Activité d'un (thread) client :
- met un certain temps (variable) pour arriver « sur les lieux »,
- s'installe dans le bar pour boire s'il y a une place libre, sinon attend

Un client peut boire seul ou à plusieurs. Le temps de consommation est variable suivant la personne.
Quand un client a fini de boire, il libère une place.

Contrainte (évidente) à respecter : Le nombre de clients installés ne doit pas dépasser la capacité du bar.

Immersion polaire : Avant de « se lancer » dans la programmation, faisons la connaissance de Maika et Jaani, des cousins éloignés d'Anouk et Amorok, et qui parlent en langue (dialecte) inuktut ...

Dialogue en inuktut, écriture romaine : [Source : https://tusaalanga.ca/fr/glossary/inuktitut]

Maika:
Nami iqqanaijaqqit Jaanii? [Où travailles-tu Jaani?]
Jaani:
Nunavut Tunngavikkunni iqqanaijaqtunga. Ivvilli, nami iqqanaijaqqit Maika?
[Je travaille à Nunavut Tunngavik. Et toi, où travailles-tu Maika ?]
Maika:
Haamalakkunni iqqainaijaqtunga. [Je travaille au hameau.]
Jaani:
Kisulirijiuvilli taikani? [Qu'est-ce que tu fais là-bas?]
Maika:
Kiinnaujalirijiujunga. [Je suis agente financière.]
Jaani:
Asukuluk. [Ha oui?]
Maika:
Ivvili, kisulirijiuvit Nunavut Tunngavikkunni? [Qu'est-ce que tu fais à Nunavut Tunngavik?]
Jaani:
Atuagalirijiujunga. [Je suis conseiller en politique.]
Maika:
Asukuluk. [Je vois.]

Le même dialogue, écriture syllabique :

Maika: ᓇᒥ ᐃᖃᓗᐃᔪᖃᑭᑦ, ᔭᓂ?
Jaani: ᓄᓇᕗᑦ ᑐᖕᒪᕆᐊᓂᓂ ᐃᖃᓗᐃᔪᖃᑐᖓ. ᐃᖏᕆᓪᓗ, ᓇᒥ ᐃᖃᓗᐃᔪᖃᑭᑦ, ᒪᐃᑲ?
Maika: ᕼᐊᐅᒪᓪᓚᐅᓂᓂ ᐃᖃᓗᐃᔪᖃᑐᖓ.
Jaani: ᑭᓯᓪᓕᔾᔨᕆᓪᓗ ᑕᐃᑲᓂ?
Maika: ᑲᓂᓇᐅᔭᓪᓕᔾᔨᐅᔪᖓ.
Jaani: ᐊᓯᔨᓗᑲ.
Maika: ᐃᖏᕆᓪᓗ, ᑭᓯᓪᓕᔾᔨᕆᑦ ᓄᓇᕗᑦ ᑐᖕᒪᕆᐊᓂᓂ?
Jaani: ᐊᑐᐊᒪᓪᓕᔾᔨᐅᔪᖓ.
Maika: ᐊᓯᔨᓗᑲ.

Après cette diversion, revenons « à nos moutons » : voici un **squelette** de programme, plutôt bien « en chair », *skel_igloo_bar1.py* , à votre disposition sur sur *moodle* /squelettes

```
#! /usr/bin/python3
# -*- coding: UTF-8 -*-
# igloo_bar1.py
# Contrainte : Nbre de clients en train de consommer <= nbre de sièges ds le bar

# Section des import
...

# __ var. globales __
# nb : Seats doit être connu avant de pouvoir initialiser la variable de
#      classe <enter_multiplex>
DefSeatsValue = 5 # valeur par défaut
try:
    Seats = input('Nbre de places ? (5 par defaut: taper Entree) ')
    if Seats == "":
        Seats = DefSeatsValue
    else:
        Seats = int(Seats)
    Nb_clients = int(input('Nbre de clients ? '))
    assert(Seats > 0)
    assert(Nb_clients > 0)
except ValueError: # si la conversion en 'int' échoue
    print('Nombre entier requis !')
    exit(1)
except AssertionError:
    print('Nbre (places et clients) positif requis !')
    exit(1)

class Client(Thread):
    # variables de classe partagées par tous les threads
    drinking = 0
    drink_mutex = Semaphore(1) # pour protéger les m.a.j. de 'drinking'
    enter_multiplex = BoundedSemaphore(...)

    def __init__(self, id, fifo):
        super().__init__() # Thread.__init__(self)
        self._id = id
        ...

    def run(self):
        self.to_arrive() # pour "espacer" les arrivées

        Client.enter_multiplex.acquire() # "zone" où il ne doit pas y avoir plus de
                                         # <SEATS> consommateurs
        ...
        Client.drinking += 1
        state = "s'installe (et boit)"
        item = "client {0:3d} {2:30s} compteur = {1:d}".format(self._id, ... , state)
        self._fifo.put(item)
        Client.drink_mutex.release()

        # phase de consommation
        self.drink()

        # Sortie d'un client
        Client.drink_mutex.acquire()
        ...
        state = "quitte sa place"
        item = ...
        self._fifo.put(item)
        ...

    def drink(self):
        sleep(uniform(1, 3))

    def to_arrive(self):
        sleep(uniform(0, 20))
        item = 'client {:3d} arrive au bar'.format(self._id)
        ...

class Affiche(Thread):
    def __init__(self, fifo):
        super().__init__() # Thread.__init__(self)
        ...

    ...
```

```
# main
Fifo = Queue()
Th_clients = [Client(i+1, Fifo) for i in range(Nb_clients)]
...
...
print("\nPlus de client !")
```

On vous demande de **compléter** le script donné et de l'appeler `igloo_bar1.py`.

Exemples d'exécution :

$ ./igloo_bar1.py

```
Nbre de sieges ? (5 par defaut: taper Entree) 4
Nbre de clients ? 9
client   2 arrive au bar
client   2 s'installe (et boit)       compteur = 1
client   6 arrive au bar
client   6 s'installe (et boit)       compteur = 2
client   5 arrive au bar
client   5 s'installe (et boit)       compteur = 3
client   2 quitte sa place            compteur = 2
client   5 quitte sa place            compteur = 1
client   6 quitte sa place            compteur = 0
client   1 arrive au bar
client   1 s'installe (et boit)       compteur = 1
client   7 arrive au bar
client   7 s'installe (et boit)       compteur = 2
client   1 quitte sa place            compteur = 1
client   9 arrive au bar
client   9 s'installe (et boit)       compteur = 2
client   8 arrive au bar
client   8 s'installe (et boit)       compteur = 3
client   7 quitte sa place)           compteur = 2
client   9 quitte sa place            compteur = 1
client   3 arrive au bar
client   3 s'installe (et boit)       compteur = 2
client   8 quitte sa place            compteur = 1
client   4 arrive au bar
client   4 s'installe (et boit)       compteur = 2
client   3 quitte sa place            compteur = 1
client   4 quitte sa place            compteur = 0
Plus de client !
```

Avertissement : l'abus d'alcool peut nuire à la qualité de la programmation !

$ ./igloo_bar1.py

```
Nbre de sieges ? (5 par defaut: taper Entree) 2
Nbre de clients ? 12
client   1 arrive au bar
client   1 s'installe (et boit)       compteur = 1
client   9 arrive au bar
client   9 s'installe (et boit)       compteur = 2
client   1 quitte sa place            compteur = 1
client   9 quitte sa place            compteur = 0
client   8 arrive au bar
client   8 s'installe (et boit)       compteur = 1
client  12 arrive au bar
client  12 s'installe (et boit)       compteur = 2
client   8 quitte sa place            compteur = 1
client   3 arrive au bar
client   3 s'installe (et boit)       compteur = 2      # le bar est plein (les 2 sièges sont occupés)
client   4 arrive au bar                                # le 4 est bloqué, en attente d'une place
client  12 quitte sa place            compteur = 1
client   4 s'installe (et boit)       compteur = 2      # le 4 peut entrer suite à la libération d'une place
client   7 arrive au bar                                # en attente de place
client   6 arrive au bar                                # en attente de place
client   4 quitte sa place            compteur = 1
client   7 s'installe (et boit)       compteur = 2
client   3 quitte sa place            compteur = 1
client   6 s'installe (et boit)       compteur = 2
client  10 arrive au bar
client   6 quitte sa place            compteur = 1
client  10 s'installe (et boit)       compteur = 2
client   5 arrive au bar
client   7 quitte sa place            compteur = 1
client   5 s'installe (et boit)       compteur = 2
client  10 quitte sa place            compteur = 1
client   5 quitte sa place            compteur = 0
client  11 arrive au bar
client  11 s'installe (et boit)       compteur = 1
client  11 quitte sa place            compteur = 0
```

```
client   2 arrive au bar
client   2 s'installe (et boit)       compteur = 1
client   2 quitte sa place            compteur = 0
Plus de client !
```

Rem. : On admet qu'un client puisse griller la priorité à un client qui attend avant lui devant le bar.
Cette situation peut se produire (très exceptionnellement !) si au moment même où un client quitte le bar, libérant ainsi une place, un client arrive et donc se retrouve devant le bar !
Pour l'instant, je ne vous demande pas d'éviter qu'un tel cas d'injustice se produise (car je souhaite que cette version reste simple)

Exemple :

```
# on suppose que le bar est plein
client   3 arrive au bar    # se met en attente
client   2 quitte sa place
client   7 arrive au bar
client   7 s'installe (et boit)     # oh le vilain ! Il profite de la « porte ouverte » au détriment du client 3 !
```

Une **pause** dans cette étude de cas qui va reprendre dans la série suivante … patience !

Aspect **important à connaître** sur les threads en Python (un regard critique et objectif)

Extrait de l'ouvrage, dans la langue de Shakespeare, de Brett Slatkin « Effective Python (59 specific ways to write better Python) » aux éditions Addison-Wesley :

Item 37 : Use Threads for Blocking I/O, Avoid for Parallelism

The standard implementation of Python is called CPython. CPython runs a Python program in two steps.
First, it parses and compiles the source text into bytecode. Then, it runs the bytecode using a stack-based interpreter.
The bytecode interpreter has state that must be maintained coherent while the Python program executes.
Python enforces coherence with a mechanism called the global interpreter lock (**GIL**).

Essentially, the GIL is a mutual-exclusion lock (mutex) that prevents CPython from being affected by preemptive multithreading, where one thread takes control of a program by interrupting another thread.
Such an interruption could corrupt the interpreter state if it comes at an unexpected time.
The GIL prevents these interruptions and ensures that every bytecode instruction works correctly with the Cpython implementation and its Cextension modules.

The GIL has an important **negative side effect**. With programs written in languages like C++ or Java, having multiple threads of execution means your program could utilize multiple CPU cores at the same time.
Although Python supports multiple threads of execution, the GIL causes only one of them to make forward progress at a time. This means that **when you reach for threads to do parallel computation and speed up your Python programs, you will be sorely disappointed**.

Illustration :
Téléchargez les trois scripts qui sont dans le dossier "Comparaison performances" sur *moodle* : *costlyWithoutThread.py*, *costlyThread.py* et *costlyMultiProcess.py*.
La fonction *CostlyFunction* qui est exécutée 100000 fois (dans chacun de ces trois exemples) est plutôt coûteuse en temps de calcul (« CPU Bound task »).

Lancez-les tour à tour sur l'un de nos serveurs et comparez leurs temps d'exécution respectifs.
Le programme *costlyMultiProcess.py* qui utilise le package `multiprocessing` est de loin le plus rapide ! (voir l'encadré ci-dessous)
Le script utilisant des threads (*costlyThread.py*) est même plus lent que *costlyWithoutThread.py* où les 100000 appels à la fonction se font séquentiellement (*costlyFunction*(*i*+1) est exécuté une fois que *costlyFunction*(*i*) est fini) !

Auriez-vous une **explication** satisfaisante à me proposer **?** …

Utilité du package `multiprocessing` :

multiprocessing is a package that supports spawning processes using an API similar to the threading module.
The multiprocessing package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads. Due to this, the multiprocessing module allows the programmer to fully leverage **multiple processors** on a given machine. It runs on both Unix and Windows.

Une dernière **question** :

Pourquoi l'utilisation de threads est-elle judicieuse et très intéressante du point de vue "performance" dans l'exemple de début de la série précédente (série *Threads*) ?

Rappel : le script en question envoie des requêtes *ping* à un certain nombre de machines/hôtes pour connaître leur « état » (vivante, défaillante ou inconnue).