Introduction à la programmation concurrente

Programmation Concurrente en Python (Cours 2)
Multi Processing + Projets

Alexander Saidi
ECL - LIRIS - CPE

Nov 2022

Rappel du cours 1

- Processus ou Thread, cas de Python (GIL)
- Mesures du temps (pour comparaisons des performances temporelles)
- États d'un processus
- Illustration de la nécessité la synchronisation et de l'exclusion mutuelle
  - ➛ via l'étude d'une instruction d'incrémentation
  - ➛ Interruptions et commutations de contextes
- Exclusion mutuelle dans une section critique avec *lock*
- Exclusion mutuelle dans une section critique avec *sémaphores*
- Exemple d'échange de données avec mp.Queue
- Exemple d'échange de données avec mp.Pipe

Passage à multiprocessing

Nous verrons (à l'aide du Package multiprocessing):

- Mémoire partage avec Shared Array
- Exercice d'allocation de ressources
- Variable de condition
- Pool de Processus
- Manager et échange de données, barrière, ...
- Event, Variable de condition, barrière, RDV, ...
- Schéma client-serveur simplifié (opérations arithmétiques)
- Série d'exercices

Exclusion mutuelle

- Exclusion mutuelle et **race condition**
- Section critique
  - Une section du code (code sensible à protéger) devient une **section critique** protégée par différents mécanismes
  - Mise en oeuvre par
    1. verrou (**lock**)
    2. sémaphores
    3. variable de condition
    4. etc.

Rappel Package multiprocessing

- Remplacer

```
import os
.....
    pid = os. fork()          # création et lancement implicite du fils
    if pid == 0 :             # Copie fils
        travail_du_fils(arg1, arg2, ...argn)
    else :                    # Copie père
        # Le pere a une activite'
        wait()  # ou _,_ = os.waitpid(pid, 0)
```

- Par :

```
import multiprocessing as mp
    ....
    pid = mp.Process(target=travail_du_fils, args=(arg1, arg2, ...argn,))   # Creation
    pid.start()                              # Lancement explicite du fils
    # Le pere a une activite ....
    pid.join() # Le pere attend la fin du fils via son id
```

Rappel : sémaphore

- Les sémaphores généralisent la notion de verrou
  - déclaré par **S=multiprocessing.Semaphore(n)**, $n \geq 0$
- Un *Lock* est un **sémaphore** avec un seul jeton ($\equiv$ un sémaphore *binaire*).
  - *verrou=mp.Lock()* équivaut à *verrou=mp.Semaphore(1)*
- Les primitives d'accès : **release()** et **acquire()** c.f. *Lock*.
- Avec les sémaphores :
  - On peut fixer le nombre de jetons à une valeur quelconque
  - y compris 0 (appelé sémaphore *privé*).
  - la tâche qui exécute *S.release()* peut ne pas être celle qui exécute *S.acquire()* On dira qu'on a un *sémaphore privé*.
- Ci-dessous, le code de l' ex. d'incrémentation avec un sémaphore.

Rappel : sémaphore (suite)

Rappel de la solution à la section critique d'incrémentation avec un sémaphore :

```
import multiprocessing as mp

def incrementer_protection_avec_Sem(variable_partagee, verrou):
    """ Chacun écrit à son rythme (non protégée)"""

    for i in range(nb_iterations):  # nb_iterations est globale en lecture (définie dans main)
        with verrou :       # «< Remarquer "with"
            variable_partagee.value += 1
```

Rappel : sémaphore (suite)

```
if __name__ == "__main__" :

    nb_iterations = 5000

    verrou = mp.Semaphore() # Val init=1 par défaut

    variable_partagee = mp.Value('i', 0) # ce sera un entier initialisé à 0

    print("la valeur de variable_partagee AVANT les incrémentations : ", variable_partagee.value)

    # On crée 2 process
    pid1 = mp.Process(target=incrementer_protection_avec_Sem, args=(variable_partagee, verrou,))
    pid1.start()
    pid2 = mp.Process(target=incrementer_protection_avec_Sem, args=(variable_partagee, verrou,))
    pid2.start()

    pid1.join()
    pid2.join()

    print("la valeur de variable_partagee APRES les incrémentations %d (attendu %d) " \
          % (variable_partagee.value, nb_iterations * 2))
```

Rappel : sémaphore (suite)

- Trace : on teste plusieurs fois ! Tout va bien cette fois.

```
la valeur de variable_partagee AVANT les incrémentations :  0
la valeur de variable_partagee APRES les incrémentations 10000 (attendu 10000):
la valeur de variable_partagee AVANT les incrémentations :  0
la valeur de variable_partagee APRES les incrémentations 10000 (attendu 10000):
...
```

Rappel : l'expression équivalente (avec le gestionnaire de **with xxx**)

```
with Sem :   # Sem est du type       mp.Semaphore()
      variable_partagee.value += 1
```

N.B. : expression avec "with" est équivalente à :

```
Sem.acquire()
try:
   variable_partagee.value += 1
finally:
   Sem.release()
```

Rappel : : Échange avec Queue (get / put)

Un exemple simple d'échange par file d'attente (Queue) :

```
from multiprocessing import Process, Queue

def travailleur(queue_pour_echanger_des_choses):
    queue_pour_echanger_des_choses.put([42, None, 'hello'])

if __name__ == '__main__':
    queue_pour_echanger_des_choses = Queue()

    p = Process(target=travailleur, args=(queue_pour_echanger_des_choses,))
    p.start()

    print(queue_pour_echanger_des_choses.get())
    p.join()
```

- Trace : "[42, None, 'hello']"

Rappel : Echange avec Pipe

Un exemple d'échange par Pipe anonyme :

- ➛ Le même principe qu'avec os.pipe() mais plus simple
- ➛ Avec les primitives **send()** et **recv()** (recv() est bloquant).

```
from multiprocessing import Process, Pipe

def worker(num, cote_fils):
    if num==1 :  cote_fils.send([42,"is","the", "best"])
    else: cote_fils.send('Hello')
    cote_fils.close()   # Optionnel

if __name__ == '__main__':
     cote_pere, cote_fils= Pipe()

    p1 = Process(target=worker, args=(1, cote_fils))
    p1.start()
    p2 = Process(target=worker, args=(2, cote_fils))
    p2.start()
    print(cote_pere.recv())  " On fait 2 "recv()" car les 2 fils ont fait 2 "send()"
    print(cote_pere.recv())
    p1.join(); p2.join()
```

Rappel : Echange avec Pipe (suite)

- Trace :

  [42, 'is','the', 'best']

  Hello

N.B :

☞ Un **Pipe** (tube) est utilisé pour faire communiquer deux processus;

➛ chacun son "coté" (son point de connexion)

Une **Queue** (file) est utilisée pour la communication entre de multiples processus; n'importe qui envoie / reçoit !

Rappel : Somme

• Rappel de l'exemple de somme : chaque fils envoie son résultat via un pipe (et 'array')

version parallèle comparée à la séquentielle.

```
import array
import os, time
import multiprocessing as mp # pour Value, Pipe

# La fonction qui fait une somme partielle (pour les fils)
def somme(num_process, table, debut, fin_exclue, pour_fils_to_send) :
    print("Je suis le fils num ", num_process, "et je fais la somme du tableau de taille ", fin_exclue-debut)

    S_local=0
    for i in range(debut, fin_exclue) :
        S_local += tableau[i]

    pour_fils_to_send.send(S_local) # Non bloquant
    #pour_fils_to_send.close()
    print(f"le pour_fils_to_send num {num_process}, envoie par send {S_local}")
```

Rappel : Somme (suite)

```
if __name__ == "__main__" :
    taille = 10**6

    # Plus efficace que les listes
    tableau = array.array('i',[i for i in range(taille)])

    pour_pere_to_receive, pour_fils_to_send=mp.Pipe()

    deb=time.time()
    process1 = mp.Process(target=somme,args=(1, tableau, 0, taille // 2, pour_fils_to_send,))
    process2 = mp.Process(target=somme,args=(2, tableau, taille // 2, taille,pour_fils_to_send,))

    process1.start(); process2.start()

    moitie1=pour_pere_to_receive.recv() # Bloquant
    moitie2=pour_pere_to_receive.recv()

    # On laisser "join" mais inutile dans ce cas car "recv()" est bloquant et les fils terminent avec send
    process1.join(); process2.join()

    fin=time.time()
    print("La somme totale du tableau obtenue : ", moitie1+moitie2, " en ", (fin-deb)*1000000)
    print(f"On vérifie que la somme par Python : {sum(tableau)}")
```

Rappel : Somme (suite)

- La version séquentielle (suite de "main") :

```
#-------------------------------------------------------------------
print('-'*50)
print("Version séquentielle (avec pipe): ")
deb=time.time()

# Exécution séquentielle
somme(1, tableau, 0, taille // 2,  pour_fils_to_send)
somme(2, tableau, taille // 2, taille, pour_fils_to_send)

moitie1=pour_pere_to_receive.recv() # bloquant
moitie2=pour_pere_to_receive.recv()

fin=time.time()
print("La somme totale du tableau en version séquentielle : ", moitie1+moitie2, " en ",  (fin-deb)*1000000)
```

Rappel : Somme (suite)

- Trace :

```
Je suis le fils num  1 et je fais la somme du tableau de taille  500000
Je suis le fils num  2 et je fais la somme du tableau de taille  500000
le fils num 1, envoie par send 124999750000
le fils num 2, envoie par send 374999750000
La somme totale du tableau obtenue :  499999500000  en   252648.11515808105
On vérifie que la somme par Python : 499999500000
-----------------------------------------------
Version séquentielle (avec pipe):
Je suis le fils num  1 et je fais la somme du tableau de taille  500000
le fils num 1, envoie par send 124999750000
Je suis le fils num  2 et je fais la somme du tableau de taille  500000
le fils num 2, envoie par send 374999750000
La somme totale du tableau en version séquentielle :  499999500000  en   465073.1086730957
```

☞ On remarque le gain : le temps est divisé $\simeq$ par deux !

Lock à la créarion de Value, Array

Exemple : l'écriture :

```
...
un_entier = mp.Value('i', 7)        # Par défaut =  je gere le SC moi–même
mutex=mp.Lock()

# Je modife la variable dans une SC
with mutex :
    un_entier += 42
```

Devient :

```
...
mutex=mp.Lock()
un_entier = mp.Value('i', 7, lock = mutex) # si "lock=True"," on accède au verrou par "un_entier.get_lock()"

# Je modife la variable dans une SC à l'aide du (sans expliciter mutex)
un_entier += 42
```

Un exemple : ../..

Lock à la créarion de Value, Array (suite)

```
from multiprocessing import Process, Lock, Semaphore
from multiprocessing.sharedctypes import Value, Array
from ctypes import  c_double

def modifier_les_vars(un_entier_protege, un_reel_NON_protege, un_str_protégé_par_lock):
    with un_entier_protege.get_lock() :
        un_entier_protege.value **= 2  # Lever au carre
    un_reel_NON_protege.value **= 2
    un_str_protégé_par_lock.value = un_str_protégé_par_lock.value.upper()  # Mettre en majuscule

if __name__ == '__main__':
    lock = Lock()  # On peut remplacer ceci par son equivalent : lock=Semaphore(1)

    un_entier_protege = Value('i', 7)  # Par défut, lock=True
    un_reel_NON_protege = Value(c_double, 1.0/3.0, lock=False)  # «– remarquer l'absence de verrou
    un_str_protégé_par_lock = Array('c', b'hello CPE', lock=lock)     # «– remarquer le verrou explicite

    p = Process(target=modifier_les_vars, args=(un_entier_protege, un_reel_NON_protege, un_str_protégé_par_lock, ))
    p.start()
    p.join()

    print(un_entier_non_protege.value, un_reel_NON_protege.value, un_str_protégé_par_lock.value)
```

Lock à la créarion de Value, Array (suite)

- Trace :

```
49
0.1111111111111111
HELLO CPE
```

☞ Manipulation d'une variable partage (Array/Value) via un verrou explicite.

- Voir aussi https://docs.python.org/3/library/multiprocessing.html
  ➛ (p.ex. BaseManager, Listeners and Clients)

☞ Lire également en bas de la même page "Programming guidelines"

Mémoire partagée : SharedArray

- On reprend le problème de somme d'un tableau
- Cette fois, le tableau est un *Shared Array*
  ➛ Plus efficace qu'avec **list / array / mp.Array**
- L'exemple compare une version séquentielle avec celle avec des Processus
  ➛ on utilise une mémoire partagée (*SharedArray*) dans une SC.

- Code de la somme d'une tranche

```
from multiprocessing import Process, Value, Lock
import os, time
from array import array  # Attention : différent des 'Array' des Process

import SharedArray as sa

def somme_d_un_tableau(processName, tableau, lock, cumul, deb, fin) :
    print ("%s: calcule la somme sur la tranche [%s .. %s[" % ( processName, deb,fin))
    Somme_local=0;
    for i in range(deb, fin) :
        Somme_local += tableau[i]
    with lock :
        cumul.value += Somme_local # section critique (mm passé en arg de cette fonc)
```

Mémoire partagée : SharedArray (suite)

- La partie principale

```
if __name__ == "__main__" :
    N=10000000
    tableau=array('i',[1 for i in range(N)]) # Initialisation

    lock=Lock() # création verou
    print("------------ Sans process")
    t_start = time.time()
    somme1 = Value('d', 0)
    somme_d_un_tableau('Sans Process ', tableau, lock, somme1, 0, N)
    t_end = time.time()
    save_temps_mono=(t_end - t_start)*1000 # Pour comparer plus bas
    print('temps Sans process : la somme = ' , somme1.value , " en " , save_temps_mono , "ms.")


    print("------------ Avec process")
    Nb_process=os.cpu_count()
    debut_tranche=0
    une_part = N // Nb_process
    mes_process=[0 for i in range(Nb_process)]

    somme2 = Value('d', 0)
    fin_tranches=[une_part*i+1 for i in range(1,Nb_process+1) ]
    fin_tranches[-1]=N  # On prend en charge tout le reste

    # Création de la mémoire partagée
```

Mémoire partagée : SharedArray (suite)

```
try: s_tableau = sa.create("shm://test", N, dtype=int)
except FileExistsError:
    sa.delete("test")
    s_tableau = sa.create("shm://test", N, dtype=int)

for i in range(N) : s_tableau[i]=1

t_start = time.time()
for i in range(Nb_process) : # Lancer     Nb_process  processus
    mes_process[i] = Process(target=somme_d_un_tableau, args=\
        ("Avec Process"+str(i+1), s_tableau, lock, somme2, debut_tranche,  fin_tranches[i],))
    mes_process[i].start()
    debut_tranche= une_part*(i+1)+1

for i in range(Nb_process) : mes_process[i].join()

t_end = time.time()

print("Somme Avec process", somme2.value)

print('Pour %d process, le temps  = %d %s %d'% (Nb_process, (t_end – t_start)*1000 , "ms. comparez à mono :",
save_temps_mono))
```

Mémoire partagée : SharedArray (suite)

La trace (comparer les temps de calculs)

```
---------- Sans process
Sans Process : calcule la somme sur la tranche [0 .. 10000000[
temps Sans process : la somme =  10000000.0  en  788.7670993804932 ms.
---------- Avec process
Avec Process1: calcule la somme sur la tranche [0 .. 1250001[
Avec Process2: calcule la somme sur la tranche [1250001 .. 2500001[
Avec Process3: calcule la somme sur la tranche [2500001 .. 3750001[
Avec Process5: calcule la somme sur la tranche [5000001 .. 6250001[
Avec Process7: calcule la somme sur la tranche [7500001 .. 8750001[
Avec Process4: calcule la somme sur la tranche [3750001 .. 5000001[
Avec Process6: calcule la somme sur la tranche [6250001 .. 7500001[
Avec Process8: calcule la somme sur la tranche [8750001 .. 10000000[
Somme Avec process 10000000.0
Pour 8 process, le temps = 548 ms. comparez à mono : 788
```

☞ Il n'est pas possible de spécifier un lock explicite lors de la création d'un SharedArray !

Retour d'une fonction

Remarques sur la valeur de retour d'un processus

- Pour obtenir une valeur calculée par une fonction exécutée par un process, Python propose différentes solutions.
  - Value
  - Array
  - SharedArray
- Mais si on souhaite pouvoir utiliser **return(.)** :
  - Gestion des retours via un Pool de processus
  - Gestion via un Manager (voir l'exemple suivant)
- Les exemples suivants montre l'utilisation d'un Pool puis d'un Manager

Pool : un exemple simple

On crée un Pool de 2 processus et on demande l'exécution de la fonction *travailleur* avec un paramètre x.
Ces paramètres sont regroupés dans une liste (la liste [1,5,3]).

```
from multiprocessing import Pool

def travailleur(x):
    """ On recoit un parametre que l'on multiplie par lui-meme """
    return x*x

if __name__ == '__main__':
    with Pool(2) as p:
        print(p.map(travailleur, [1, 5, 3]))

"""
[1, 25, 9]
"""
```

Pool : un exemple simple (suite)

• Un **Pool** de k processus (ici 2) est un réservoir de k processus gérée par un *exécuteur*.

• L'exécuteur décide de répartir les travaux à réaliser (la fonction *travailleur*) entre les k processus.

➛ Il peut récupérer les résultats renvoyés par les processus.

☞ **A noter** : *Pool* exige la présence d'une section __*main*__ qui sera importée par les fils.

➛ C-à-d. les exemples utilisant *multiprocessing.Pool* ne fonctionnent pas dans l'interpréteur interactif de Python (où il n'y a pas de __*main*__).

➛ Voir aussi la page "conseil de programmation" de Python.

Pool : exemple somme

• Ci-dessous, une solution avec un Pool de processus pour faire la somme des éléments d'un vecteur, tranche par tranche.

- Pour simplifier, le tableau dont on veut la somme (en lecture) est placé en global
- Chaque processus s'occupe de faire la somme d'une tranche.
- Un processus récupère les indices de début / fin de sa tranche qui lui donnent accès à 2 listes dédiées (d'indices) *debut_tranches* et *fin_tranches*,
- Les processus sont ici organisés dans un pool.
- Chaque processus termine avec la clause "return" et renvoie son résultat

➛ "return" Possible car Pool !

➛ Le **Pool** est capable, par son mécanisme interne, de récupérer ces résultats.

Pool : exemple somme (suite)

- Le code de chaque processus qui fait la somme d'une tranche (slice) :
  - Le tableau à "sommer" est une variable globale et remplie de 10^6 de 1s.
  - On a l'intention d'utiliser 6 processus (sur un Intel I7).

  ➛ Ce paramètre s'adapte à votre plateforme (p.ex. 2 pour un Intel I3).

```
from multiprocessing import  Value, Pool
import os, time
from array import array # Attention : different des 'Array' des Process

N=1000000
L=array('i',[1 for i in range(N)]) # Initialisation

debut_tranches=fin_tranches=[]

# Definir une founction pour ls processs
def somme(ind) : #ind est un indice dans les listes "Tranches" et permet de retrouver sa "tranche
    deb=debut_tranches[ind]; fin=fin_tranches[ind]
    print ("On calcule la somme sur la tranche [%s .. %s[" % (deb,fin ))
    S_local=0;
    for i in range(deb, fin) :
        S_local += L[i]
    return S_local
```

Pool : exemple somme (suite)

- Le Main :

```
if __name__ == "__main__" :
    Nb_process = os.cpu_count() – 2

    une_part = N // Nb_process  # Chaque process devra "sommer" tant de valeurs

    # Préparation des indices des tranches
    fin_tranches = [une_part * i + 1 for i in range(1, Nb_process + 1)]
    fin_tranches[–1] = N  # On prend en charge tout le reste

    debut_tranches=[fin_tranches[i–1] for i  in range(1,Nb_process)]
    debut_tranches.insert(0,0) # mettre (0,0) en tête de la liste.

    indices=[i for i in range(Nb_process)]

    print("Controle : debut_tranches :", debut_tranches, "\nfin_tranches :", fin_tranches, "\n")
    t_start = time.time()
    with  Pool(Nb_process) as poupoule : # Les processus recevront un paramère = un indice qui leur est propre
        liste_de_sommes_partielles = list(poupoule.map(somme, indices))

    print("res = ", sum(liste_de_sommes_partielles))

    t_end = time.time()
    print('Pour %d process, le temps  = %d %s'% (Nb_process, (t_end – t_start)*1000 , "ms. "))
```

Pool : exemple somme (suite)

☞ Expliquer "map"

- Trace :

```
"""
Controle : debut_tranches : [0, 166667, 333333, 499999, 666665, 833331]
fin_tranches : [166667, 333333, 499999, 666665, 833331, 1000000]

On calcule la somme sur la tranche [0 .. 166667[
On calcule la somme sur la tranche [166667 .. 333333[
On calcule la somme sur la tranche [333333 .. 499999[
On calcule la somme sur la tranche [499999 .. 666665[
On calcule la somme sur la tranche [666665 .. 833331[
On calcule la somme sur la tranche [833331 .. 1000000[
res =  1000000
Pour 6 process, le temps  = 240 ms.
"""
```

N.B. : Le mécanisme équivalent à Pool (*concurrent.futures.ProcessPoolExecutor*) englobe et utilise un Pool de processus. Il simplifie les écritures mais souffre de quelques limitations.

➛ Voir la doc Python sur ce package.

Exemple avec Manager

Dans cet exemple, les compte rendus des actions des processus sont placés dans un dico Python (Expliquer Dico ?).

• Chaque processus $P_{i=0..4}$ reçoit les coordonnées d'un point $(x, y) \in [0.0, 1.0]^2$ et vérifie si ce point est dans un cercle de rayon 1.

➛ Le résultat de ce test est placé dans un dico pour la clé i

• Le code des processus :

```
import multiprocessing, time, random

def verifier_si_x_y_dans_cercle_R(my_num, x,y,R, dico_des_comptes_rendus):
    """verifier si (x,y) est  dans le cercle de rayon R """
    print("je suis le processus " + str(my_num) +  " et je verifie si ", (x,y), "dans le cercle de rayon", R)
    time.sleep(1)
    resultat = x*x + y*y <= R*R
    print(my_num , " :  je renvoie la réponse")

    dico_des_comptes_rendus[my_num] = resultat
```

Exemple avec Manager (suite)

- Le code du "MAIN" :

```
if __name__ == "__main__":
    manager = multiprocessing.Manager()
    dico_des_comptes_rendus = manager.dict()
    ids = []
    Rayon_du_cercle=1
    for i in range(5):
        x,y=random.random(),random.random()
        p = multiprocessing.Process(target=verifier_si_x_y_dans_cercle_R, \
                args=(i, x,y,Rayon_du_cercle, dico_des_comptes_rendus))
        ids.append(p)
        p.start()

    for proc in ids:
        proc.join()

    print(dico_des_comptes_rendus.values())
```

☞ Ici, le Manager a géré (pour nous) le partage et la protection du dico

Exemple avec Manager (suite)

- Trace :

```
"""
TRACE :
je suis le processus 0 et je verifie si  (0.5575801575536624, 0.472181451084017) dans le cercle de rayon 1
je suis le processus 1 et je verifie si  (0.8963054538237742, 0.6835018754810716) dans le cercle de rayon 1
je suis le processus 2 et je verifie si  (0.46470068334128767, 0.3245305760146546) dans le cercle de rayon 1
je suis le processus 3 et je verifie si  (0.5882645789778974, 0.8398554207179544) dans le cercle de rayon 1
je suis le processus 4 et je verifie si  (0.8209510929670091, 0.07184267175641679) dans le cercle de rayon 1
0  :  je renvoie la réponse
1  :  je renvoie la réponse
2  :  je renvoie la réponse
3  :  je renvoie la réponse
4  :  je renvoie la réponse
[True, False, True, False, True]
"""
```

Un exemple de Event

- Python gère également des événements (Event).
- Un simple mécanisme "attendre-signaler" via **set() / wait()**.
- Opérateurs : **set(), is_set(), wait(time_out), clear()** .
- *set()* réveille tous les processus en attente; ce réveil n'efface pas le drapeau (mis par *set()*)
- Un *set()* suivi immédiatement de *clear()* n?empêchera pas les réveils !

```
import multiprocessing as mp
import time

def travailleur(event, timeout):
    print("Process démarré qui attend  un Event...")

    # Faire attendre jsq'à l'arrivée de l'evt avec time-out
    evt_arrive = event.wait(timeout)
    if evt_arrive:
        print("Event reçu, ça libère le process...")
    else:
        print("Time out : on avance quand même ! ")
```

Un exemple de Event (suite)

```
if __name__ == '__main__':
    # Création / initialisation de l'Event
    e = mp.Event()

    # démarrage du processus fils
    process1 = mp.Process(name='Attente-sur-Event', target=travailleur, args=(e,4))
    process1.start()

    # la main attend  3 secondes
    time.sleep(3)

    # puis génère l'Event
    e.set()
    print("Event est signalé.")
"""
Process démarré qui attend  un Event...
Event reçu, ça libère le process...
Event est signalé.
"""
```

☞ Event permet très simplement de réaliser des RDV.

☞ Il est possible d'implanter *Event* à l'aide de *Condition*.

Variable de condition

- Similaire aux **Events** mais multi-processus
- Création par **mp.Condition(Lock=None)**
- Par défaut, pas de verrou explicite nécessaire
- Si pas de verrou explicite, un verrou (*RLock*) est fourni
- Mécanisme de synchronisation avec les primitives :

  wait() / wait_for() / notify() / notify_all()

  ➛ Mais si Lock spécifié, un appel préalable à *acquire()* est requis

Variable de condition (suite)

- A propos des primitives :
  - wait() / wait_for() : il faut au préalable un appel à *acquire()* sur le verrou explicite

    Un appel à wait() libère (automatiquement) le verrou (appelé par *acquire()*)

    Après le réveil, ce verrou est (automatiquement) libéré
  - *notify() / notify_all()* : il faut au préalable un appel à *acquire()* sur le verrou explicite

- Les variables de Condition peuvent être utilisées pour synchroniser des étapes d'un travail tels que certaines parties seront en parallèle mais d'autres séquentielles, même dans des processus séparés.

Exemple de Var de cond.

Exemple :

Ici, on lance 3 processus P_1, P_{21} et P_{22} où P_{21} et P_{22} se chargeront de l'étape 2 d'un projet mais doivent attendre que le processus P_1 termine l'étape 1 d'abord avant que ces deux processus parallèles effectuent l'étape 2.

```
import multiprocessing
import time

def etape_1(cond):
    """Première étape d'un travail puis notification à etape_2 de continuer"""
    name = multiprocessing.current_process().name
    print ('Lancement étape_1', name)
    with cond: # Nous dispense de prendre le verrou
        print ("%s : étape_1 terminée; les process de l'étape 2 peuvent commencer" % name)
        cond.notify_all()

def etape_2(cond):
    """Attendre la  condition qui dit :  etape_1 terminée"""
    name = multiprocessing.current_process().name
    print ('Lancement étape_2', name)
    with cond:
        cond.wait()
        print ('%s en cours' % name)
```

Exemple de Var de cond. (suite)

La partie main :

```
if __name__ == '__main__':
    condition = multiprocessing.Condition()
    s1 = multiprocessing.Process(name='étape_1', target=etape_1, args=(condition,))
    s2_clients = [multiprocessing.Process(name='etape_2[%d]' % i, target=etape_2, args=(condition,))
                  for i in range(1, 3)]

    for c in s2_clients:
        c.start()
        time.sleep(1)
    s1.start()

    s1.join()
    for c in s2_clients:
        c.join()

"""
Lancement étape_2 etape_2[1]
Lancement étape_2 etape_2[2]
Lancement étape_1 étape_1
étape_1 : étape_1 terminée; les process de l'étape 2 peuvent commencer
etape_2[1] en cours
etape_2[2] en cours
"""
```

Condition et RDV

Un cas d'utilisation des Conditions :

Après un premier traitement (pour chacun), K processus se donnent RDV pour se synchroniser et reprendre ensuite la suite de leurs travaux.
Chaque processus répète cette séquence indéfiniment.

- ➛ Comment réaliser ce RDV ?
  - Solution 1 : à l'aide des sémaphies / verrous
  - Solution 2 : à l'aide d'une variable de *condition*
  - Solution 3 : à l'aide d'un *Event*
  - ➛ La 3e solution est très similaire à la 2e

Condition et RDV (suite)

- Solution 1 : avec verrou et sémaphore

```
k = nombre de processus
nb_process_arrivés_au_RDV : variable partagée = 0  # valeur initiale
verrou :  Lock pour protéger la variable partagée nb_process_arrivés_au_RDV
sem = Sémaphore(0)  # pour gérer les processus en attente de la réalisation du RDV

def travail_de_chaque_processus(num_processus, paramètres...) :
    while True :
        # traitement de la première partie
        RDV()
        # suite des traitements

def RDV() :
    with verrou : # consulter rapidement la variable nb_process_arrivés_au_RDV
        nb_process_arrivés_au_RDV += 1
        on_est_combien = nb_process_arrivés_au_RDV
    if on_est_combien == k : # Tout le monde est arrivé au RDV
        with verrou : # réinitialiser la variable nb_process_arrivés_au_RDV
            nb_process_arrivés_au_RDV = 0 # Pour le prochain RDV
        # Libérer tout le monde (sauf le processus actuel) en attente sur sem
        for i in range(k-1) : # Le processus actuel ne s'était pas mis en attente
            sem.release()
    else :
        sem.acquire() # On se bloque sur ce sémaphore
```

Condition et RDV (suite)

- Solution 2 : avec verrou et Condition

```
k = nombre de processus
nb_process_arrivés_au_RDV : variable partagée = 0  # valeur initiale
verrou :  Lock pour protéger la variable partagée nb_process_arrivés_au_RDV
tous_là : Condition

def travail_de_chacun(num_processus, paramètres...) :
    while True :
        # traitement de la première partie
        RDV()
        # suite des traitements

def RDV() :
    with verrou : # consulter rapidement la variable nb_process_arrivés_au_RDV
        nb_process_arrivés_au_RDV += 1
        on_est_combien = nb_process_arrivés_au_RDV
    if on_est_combien == k : # Tout le monde est arrivé au RDV
        with verrou : # réinitialiser la variable nb_process_arrivés_au_RDV
            nb_process_arrivés_au_RDV = 0 # Pour le prochain RDV
        # Libérer tout le monde (sauf le processus actuel) en attente sur sem
        tous_là.notify_all()
    else :
        tous_là.wait() # On se bloque sur cette condition
```

Condition et RDV (suite)

- Solution 3 : avec verrou et Event ($\simeq$ Solution 2)

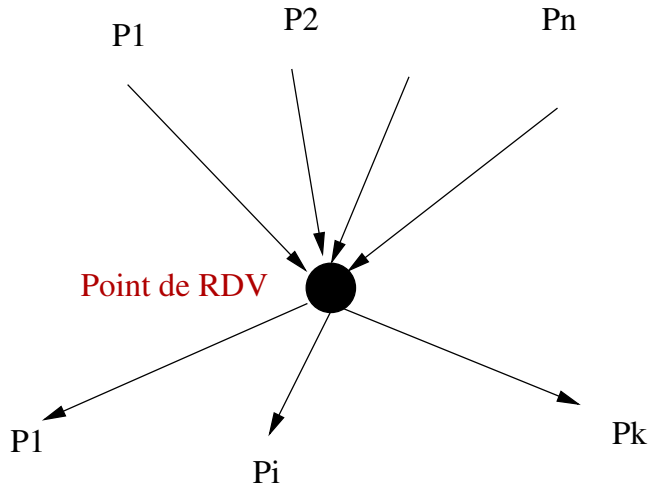
```
k = nombre de processus
nb_process_arrivés_au_RDV : variable partagée = 0  # valeur initiale
verrou :  Lock pour protéger la variable partagée nb_process_arrivés_au_RDV
tous_là : Event

def travail_de_chacun(num_processus, paramètres...) :
    while True :
        # traitement de la première partie
        RDV()
        # suite des traitements

def RDV() :
    with verrou : # consulter rapidement la variable nb_process_arrivés_au_RDV
        nb_process_arrivés_au_RDV += 1
        on_est_combien = nb_process_arrivés_au_RDV
    if on_est_combien == k : # Tout le monde est arrivé au RDV
        with verrou : # réinitialiser la variable nb_process_arrivés_au_RDV
            nb_process_arrivés_au_RDV = 0 # Pour le prochain RDV
        # Libérer tout le monde (sauf le processus actuel) en attente sur sem
        tous_là.set()
        tous_là.clear() # N'empêche pas le réveil en chaine
    else :
        tous_là.wait() # On se bloque sur cette condition
```

Barrières

Les Barrières simplifient le mécanisme de RDV des processus. *k* processus se mettront en attente de l'arrivée de l'évènement "tous les processus se sont présentés à un point de RDV". Ils seront débloqués lorsque les *k* processus seront là !



Exemple de Barrière

Exemple : un groupe d'élèves prépare une (grosse) fête. Ils vont à Carrouf acheter des bières et des chips.
Le magasin risque de ne pas avoir tout ce qui est nécessaire.
Les élèves se scindent en deux : quelques uns iront au rayon des bières et les autres les chips.
Pour se synchroniser, ils décident de se donner RDV à un point de rencontre; faire le point sur ce qui a été trouvé pour ensuite décider de la suite.
De plus, une fois tous à ce RDV, l'un d'eux ira chercher sa voiture pour transporter les achats.
☞ On aura également un processus "surveillant" qui en permanence rend compte de l'état du RDV.

Exemple de Barrière (suite)

Algorithme de principe

Le travail des chargés des bières / chips est similaire.

Ce travail consistera en :

1. Prendre quelques bières (Chips) : aléatoirement dans le code
2. Se présenter au point de RDV (*wait* bloquant)
3. A la sortie du RDV :
   - ➛ l'un ira chercher la voiture (il arrête les courses)
4. Les autres prendront le reste des bières (chips)

Le "surveillant" surveille le nombre de processus en attente du RDV et informe également du nombre restant de Bières (Chips).

Exemple de Barrière (suite)

```
# Ex RDV avec Barrière
# K processus font part1 de leur travail puis RDV puis leur PART2
# Après le RDV, l'un ira chercher la voiture.

import multiprocessing  as mp
import time, random

def faire_des_courses_ensemble(mon_num, barrier):
    mon_nom=mp.current_process().name   # extraire le nom du processus
    print(f"Le processus du nom {mon_nom} commence Part1 de ses courses")
    if mon_nom.split('-')[0]=="biere" : # Un processus Biere
        # PART1 : de mes courses : je prend des bieres

        with nb_bouteilles_bieres_necessaires.get_lock() :
            nb_bouteilles_bieres_restants_a_acheter=nb_bouteilles_bieres_necessaires.value

        # On va mettre qq bouteilles dans le caddy   (aléatoire)
        nb_bouteilles_prises=random.randint(1,nb_bouteilles_bieres_restants_a_acheter)
                if nb_bouteilles_bieres_restants_a_acheter >0 else 0

        # MAJ du nbr de bieres
        with nb_bouteilles_bieres_necessaires.get_lock() :
            nb_bouteilles_bieres_necessaires.value -= nb_bouteilles_prises

        print(f"\t(Biere) Le process {mon_nom} a prise {nb_bouteilles_prises} bieres")
        time.sleep(random.randint(1,5)*0.8)
```

Exemple de Barrière (suite)

```
# On se met en attente du RDV
res = barrier.wait() # The return value is an integer in the range 0 to parties ? 1, different for each thread.

if res==0 : # Un seul recevera 0 (chacun recoit un eval diff entre 0 et nb_process-1)
    print("$"*10,f"Le process {mon_nom} ira chercher la voirure (il quitte les courses)", "$"*10)
    return

# PART2 : A la sortie du RDV, s'il faut encore des bieres, je les prend
time.sleep(random.randint(1,5)*0.2)
with nb_bouteilles_bieres_necessaires.get_lock() :
    deuxieme_paquet_de_bieres=nb_bouteilles_bieres_necessaires.value
    nb_bouteilles_bieres_necessaires.value =0

print(f"\t(Biere) Le process {mon_nom} a complété avec {deuxieme_paquet_de_bieres} bieres")
      if deuxieme_paquet_de_bieres >0 else ''
```

Exemple de Barrière (suite)

De même pour les chargés de Chips !

```
else : # mon_nom.split('-')[0]=="chips" :
    # PART1 : de mes courses : je prend des chips

    with nb_paquets_chips_necessaires.get_lock() :
        nb_paquets_chips_restants_a_acheter=nb_paquets_chips_necessaires.value

    # On va mettre qq paquest de chips dans le caddy   (aléatoire)
    nb_bouteilles_prises=random.randint(1,nb_paquets_chips_restants_a_acheter)
            if nb_paquets_chips_restants_a_acheter >0 else 0
    with nb_paquets_chips_necessaires.get_lock() :
        nb_paquets_chips_necessaires.value -= nb_bouteilles_prises

    print(f"\t(Chips) Le process {mon_nom} a prise {nb_bouteilles_prises} chips")
    time.sleep(random.randint(1,7)*0.2)

    # On se met en attente du RDV
    res = barrier.wait()
    if res==0 : # Un seul recevera 0 (chacun recoit un eval diff entre 0 et nb_process-1)
        print("$"*10,f"Le process {mon_nom} ira chercher la voirure(il quitte les courses)", "$"*10)
        return

    # PART2 : A la sortie du RDV, s'il faut encore des chips, je les prend
    time.sleep(random.randint(1,7)*0.5)
    with nb_paquets_chips_necessaires.get_lock() :
        deuxieme_paquet_de_chips=nb_paquets_chips_necessaires.value
```

Exemple de Barrière (suite)

```
            nb_paquets_chips_necessaires.value =0

        print(f"\t(Chips) Le process {mon_nom} a complété avec {deuxieme_paquet_de_chips} chips")
                if deuxieme_paquet_de_chips >0 else ''
```

Le processus surveillant :

```
#-------------------------------------------------------------------------------
def surveillant(bar) :
    while True : # On l'arretera dans main
        with nb_bouteilles_bieres_necessaires.get_lock() :
            print(f"(Surveillant) il faut encore {nb_bouteilles_bieres_necessaires.value} bouteilles de Bieres", end=' ')
        time.sleep(0.1)
        with nb_paquets_chips_necessaires.get_lock() :
            print(f" et {nb_paquets_chips_necessaires.value} paquets de chips", end=' ')
        print(f", Il reste encore {bar.n_waiting} processus en attente du RDV")
        time.sleep(0.5)
```

Exemple de Barrière (suite)

La partie Main

```
if __name__ == '__main__':

    Nb_Process = 4
    nbloops = 10
    nb_bouteilles_bieres_necessaires = mp.Value('i', 10,lock=True) # par défaut, c'est True
    nb_paquets_chips_necessaires = mp.Value('i', 15)

    lst_processes = []
    bar = mp.Barrier(Nb_Process, action=lambda : print("*"*10+"Tous là !" + "*"*10)) # timeout=None

    for i in range(0, Nb_Process//2):
        lst_processes.append(mp.Process(name="biere-"+str(i), target=faire_des_courses_ensemble, args=(i,bar,) ) )

    for i in range(0, Nb_Process//2):
        lst_processes.append(mp.Process(name="chips-"+str(i), target=faire_des_courses_ensemble, args=(i,bar,) ) )

    pid_surveillant=mp.Process(target=surveillant, args=(bar,) )
    pid_surveillant.start()

    for process in lst_processes: process.start()

    for process in lst_processes: process.join()

    pid_surveillant.terminate()
```

Exemple de Barrière (suite)

La trace

```
"""
 Ex-Barriere-RDV.py
Le processus du nom biere-0 commence Part1 de ses courses
      (Biere) Le process biere-0 a prise 1 bieres
Le processus du nom biere-1 commence Part1 de ses courses
      (Biere) Le process biere-1 a prise 4 bieres
Le processus du nom chips-0 commence Part1 de ses courses
      (Chips) Le process chips-0 a prise 14 chips
Le processus du nom chips-1 commence Part1 de ses courses
      (Chips) Le process chips-1 a prise 1 chips
(Surveillant) il faut encore 10 bouteilles de Bieres  et 0 paquets de chips , Il reste encore 0 processus en attente du RDV
(Surveillant) il faut encore 5 bouteilles de Bieres  et 0 paquets de chips , Il reste encore 1 processus en attente du RDV
**********Tous là !**********
$$$$$$$$$$ Le process chips-1 ira chercher la voirure (il quitte les courses) $$$$$$$$$$
      (Biere) Le process biere-1 a complété avec 5 bieres
(Surveillant) il faut encore 5 bouteilles de Bieres  et 0 paquets de chips , Il reste encore 0 processus en attente du RDV
(Surveillant) il faut encore 0 bouteilles de Bieres  et 0 paquets de chips , Il reste encore 0 processus en attente du RDV
(Surveillant) il faut encore 0 bouteilles de Bieres  et 0 paquets de chips , Il reste encore 0 processus en attente du RDV
"""
```

TabMat

TabMat (suite)