Introduction à la programmation concurrente

Programmation Concurrente en Python (Cours 2)

Multi Processing + Projets

Alexander Saidi
ECL - LIRIS - CPE

Nov 2020

Rappel du cours 1

1. Processus ou Thread, cas de Python (GIL)
2. Différents exemples d'illustration de fork(), **pipe anonyme et nommé**, **signal()**, **execlp()**, **wait()**, **waitpid()**

   ➛ Voir ci-après la liste des exemples
3. Mesures du temps (pour comparaisons des performances temporelles)
4. États d'un processus
5. Illustration de la nécessité la synchronisation et de l'exclusion mutuelle via l'étude d'une instruction d'incrémentation

   ➛ Interruptions et commutations de contextes

Cours 1 : exemples étudiés

1. Somme des des éléments d'un tableau :

   Versions séquentielle et parallèle

   Création de processus par *fork()*

   Récupération des résultats via une variables partagée par *mp.Value()* et *value*

   Attente avec *os.wait()*

   Version avec *array* (ne pas confondre avec *mp.Array*) et mp.Pipe()

2. Calcule de la valeur approchée de π par la méthode *Arctg*

   Versions séquentielle et parallèle

   Utilisation de *mp.Value()*, *value*, *os.waitpid(pid, 0)*

3. Demande du nombre de processus parallèles possible sur une plateforme donnée.

Cours 1 : exemples étudiés (suite)

4. Échange entre processus via un pipe anonyme : client demandeur, serveur calculette.
5. Différentes mesures du temps (pour comparaisons)
6. Échange entre processus via un pipe nommé

   Utilisation de *fork()*, *execlp()*, *signal()*
7. Échange entre processus via un pipe nommé pour réaliser des opérations arithmétiques

   Utilisation de *fork()*, *execlp()*, *signal()*
8. Schéma client-serveur simplifié (opérations arithmétiques)
9. Exemple de la commande Linux avec pipe() : "ps -ef | wc -l"
10. Exemple d'incrémentation et illustration des interruption / commutation de contextes

Passage à multiprocessing

Nous verrons (dans le Package multiprocessing):

1. Exclusion mutuelle dans une section critique avec *lock*
2. Exclusion mutuelle dans une section critique avec *sémaphores*
3. Exemple d'échange de données avec mp.Queue
4. Exemple d'échange de données avec mp.Pipe
5. Mémoire partage avec Shared Array
6. Exercice d'allocation de ressources
7. Pool de Processus
8. Manager et échange de données
9. Rappel de quelques détails sur fork()
10. Série d'exercices

Exclusion mutuelle

- Exclusion mutuelle et **race condition**
- Section critique
  - Une section à protéger devient une **section critique** protégée par différents mécanismes
  - Mis en oeuvre par

1. verrou (**lock**)
2. sémaphores
3. variable de condition
4.

Exemple de race condition

On reprend l'exemple vu en cours 1.

- 2 processus incrémentent N fois et concurremment une variable globale.
- Qu'a-t-on à la fin dans cette variable ?
- Bien remarquer les valeurs affichées par les processus.

- Le code simple de l'incrémentation (du cours 1, avec *fork()*):

```
import multiprocessing as mp
import os

# Incrementation sans proteger la variable partagee (non protegee)
def count1_on_se_marche_sur_les_pieds(nb_iterations):
    global variable_partagee
    for i in range(nb_iterations):
        variable_partagee.value += 1
    os._exit(0)
```

Exemple de race condition (suite)

- Le main :

```
if __name__ == '__main__' :
    nb_iterations = 10000

    variable_partagee = mp.Value('i',0)   # La variable épartage : un entier éinitialis à 0
    print("la variable_partagee AVANT les éincrmentations : ", variable_partagee.value)

    # On écre 2 process
    id1=os.fork()
    if id1 == 0 : # le fils No 1
        count1_on_se_marche_sur_les_pieds(nb_iterations)
    else : # Le pere
        id2 = os.fork()
        if id2 == 0 : # le fils No 2
            count1_on_se_marche_sur_les_pieds(nb_iterations)

    # On attend la fin  des fils
    pid, status = os.waitpid(id1, 0) ; print(pid, status)
    pid, status = os.waitpid(id2, 0); ; print(pid, status)

    print("la variable_partagee APRES les éincrmentations %d (attendu %d) "% (variable_partagee.value,nb_iterations*2))
```

Exemple de race condition (suite)

- Une trace (plusieurs exécutions) :

```
"""
QQ traces

la valeur de variable_partagee APRES les éincrmentations 10278 (attendu 20000)
la valeur de variable_partagee APRES les éincrmentations 10521 (attendu 20000)
la valeur de variable_partagee APRES les éincrmentations 12067 (attendu 20000)
la valeur de variable_partagee APRES les éincrmentations 10669 (attendu 20000)
la valeur de variable_partagee APRES les éincrmentations 10282 (attendu 20000)
"""
```

☞ Presque jamais on a le bon compte !

- Cause : voir (cours 1) l'anatomie d'une instruction d'incrémentation.

L'incrémentation `variable_partagee.value += 1` devient une *section critique* (SC).

Protection de la SC avec verrou

- On reprend l'exemple d'incrémentation d'une variable partagée.
  - On protège une SC qui est la zone où il y a un risque de *perturbation* (d'interruption).
  - Les modifications à apporter sont en gras et de plus grande taille.

```
import multiprocessing as mp

# éIncrmentation avec protection de la variable épartage
def count2_on_protege_la_section_critique(nb_iterations):
    """ Chacun éincrmente dans la section ééprotge """

    for i in range(nb_iterations):
        verrou.acquire()
        variable_partagee.value += 1
        verrou.release()
```

Protection de la SC avec verrou (suite)

Le reste du code :

```
#---------- PARTIE principale (le point d'éentre de cet exemple -------
# On recommence avec la version ééprotge par un verrou
if __name__ == '__main__' :
    nb_iterations = 10000

    # La variable épartage
    variable_partagee = mp.Value('i',0)  # ce sera un entier

     verrou=mp.Lock()

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

    # On écre 2 process
    pid1=mp.Process(target=count2_on_protege_la_section_critique, args=(nb_iterations,)); pid1.start()
    pid2=mp.Process(target=count2_on_protege_la_section_critique, args=(nb_iterations,)); pid2.start()
    pid1.join(); pid2.join()

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

Protection de la SC avec verrou (suite)

- Trace : on teste plusieurs fois !

```
la valeur de variable_partagee AVANT les éincrmentations :  0
la valeur de variable_partagee APRES les éincrmentations 20000 (attendu 20000):
la valeur de variable_partagee AVANT les éincrmentations :  0
la valeur de variable_partagee APRES les éincrmentations 20000 (attendu 20000):
...
```

- Le verrou :
  - Création : verrou=mp.Lock()
  - Demande bloquante d'entrée en SC : verrou.acquire()
  - Demande de sortie de la SC : verrou.release()
- Un verrou (*Lock*) est équivalent à un sémaphore binaire (1 seul jeton)
- Possible : RLock() (un Lock Ré-entrant = Récursif)

Protection de la SC avec verrou (suite)

Remarque : *With-Statement-Context Manager* :

- Il est possible de simplifier l'écriture :

```
def count2_on_protege_la_section_critique(nb_iterations):
    """ Chacun écrit dans la section ééprotge """
    global variable_partagee;  global verrou
    for i in range(nb_iterations):
        verrou.acquire()
        variable_partagee.value += 1
        verrou.release()
```

- Par un gestionnaire de with-expression (**with xxx**)

```
def count2_on_protege_la_section_critique(nb_iterations):
    """ Chacun écrit dans la section ééprotge """
    global variable_partagee;  global verrou
    for i in range(nb_iterations):
        with verrou :
            variable_partagee.value += 1
```

L'intérêt : écriture simplifiée, fin du risque d'oublier *release()*

→ cet oubli peut provoquer un **DeadLock**.

Solution avec un sémaphore

- Un *Lock* est un **sémaphore** avec un seul jeton (sémaphore *binaire*).
- Les sémaphores généralisent la notion de verrou
  - ➛ On peut fixer le nombre de jetons à une valeur quelconque
  - ➛ y compris 0 (appelé sémaphore *privé*).
  - ➛ déclaré par **S=multiprocessing.Semaphore(n)**, $n \geq 0$
  - ➛ la tâche qui exécute *S.release()* peut ne pas être celle qui exécute *S.acquire()*.
- Les primitives d'accès : **release()** et **acquire()**
  - ➛ les mêmes que pour un verrou *Lock*.

- Ci-dessous, le code du même exemple d'incrémentation avec un sémaphore.

Solution avec un sémaphore (suite)

☞ On en profite pour passer à multiprocessing !

```
import multiprocessing as mp

#Ici, le verrou est déclaré en global (par opp. à une déclaration dans "main")
#Pas propre mais évite le passage du sémaphore en paramètre
verrou = mp.Semaphore()  # Val init=1 par édfaut

#De même pour la "variable_partagee" (pas bien !)
variable_partagee = mp.Value('i', 0)  # ce sera un entier éinitialis à 0

def count2_SC_sem(nb_iterations):
    """ Chacun écrit à son rythme (non ééprotge)"""
    global variable_partagee
    for i in range(nb_iterations):
        with verrou :       # «< Remarquer "with"
            variable_partagee.value += 1
```

Solution avec un sémaphore (suite)

```
if __name__ == "__main__" :

    nb_iterations = 5000

    # La variable épartage : éplace hors cette fonction (sinon, la passr en param)
    # variable_partagee = mp.Value('i',0)  # ce sera un entier éinitialis à 0

    print("la valeur de variable_partagee AVANT les éincrmentations : ",
          variable_partagee.value)
    # On écre 2 process
    pid1 = mp.Process(target=count2_SC_sem, args=(nb_iterations,))
    pid1.start()
    pid2 = mp.Process(target=count2_SC_sem, args=(nb_iterations,))
    pid2.start()
    pid1.join()
    pid2.join()

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

Solution avec un sémaphore (suite)

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

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

Package multiprocessing

Éléments d'utilisation :

- Remplacer

```
import os
.....
    pid = os. fork()          # écration et lancement implicite du fils
    if pid == 0 :             # Copie fils
        travail_du_fils(arg1, arg2, ...argn)
    else :                    # Copie èpre
        # 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
```

Éviter la SC : Solution alternative

- Parfois il est possible d'éviter une SC.
- Ex somme : on s'arrange pour avoir chacun "sa case" pour y écrire sa "part" !
- On utilise le type **mp.Array** (notez le 'A' majuscule).
  - ➛ Le verrou disparaît (plus besoin dans ce cas précis).
  - ➛ A aucun moment la *race-condition* survient.

```
Le même exemple d'incrémentation d'une variable partagée
import multiprocessing as mp

# Ici, chaque process éincrmente la valeur de "SA case" (une case par processeur)
def count3_on_travaille_dans_un_array(nb_iterations):
    global tableau_partage
    for i in range(nb_iterations):
        mon_indice = mp.current_process().pid % 2  # donnera 0 / 1 selon le process
        tableau_partage[mon_indice]+=1

    var_local_a_moi_tout_seul=0
    for i in range(nb_iterations): var_local_a_moi_tout_seul+=1
     # Et on écrit UNE SEULE FOIS :
    mon_indice = mp.current_process().pid % 2  # <<-- Quelle est mon indice (0 ou 1)
    tableau_partage[mon_indice]+=1
```

Éviter la SC : Solution alternative (suite)

```
#------------------------- Avec Array ------------------------------
if __name__ == "__main__" :
    tableau_partage = mp.Array('i', 2)  # tableau de 2 entiers

    # Initialisation des array :
    tableau_partage[0]=0; tableau_partage[1]=0;   # Initialisation de l'arra
    # Ou via
    tableau_partage[:]= [0 for _ in range(2) ] # IL FAUT les [:] sinon, tableau_partage devient une liste !

    # ATTENTION : NE PAS INITIALISER comme ceci : tableau_partage= [0 for _ in range(2) ]
    # Cette écriture éredfinira  notre Array comme une liste ! (principe de la prog. fonctionnelle)
    # Egalement, sans [:], print dennera le type de l'Array, pas son contenu

    print("le contenu du tableau_partage AVANT les éincrmentations : ", tableau_partage[:])

    # On écre 2 process
    nb_iterations = 5000
    pid1=mp.Process(target=count3_on_travaille_dans_un_array, args=(nb_iterations,)); pid1.start()
    pid2=mp.Process(target=count3_on_travaille_dans_un_array, args=(nb_iterations,)); pid2.start()
    pid1.join(); pid2.join()

    print(tableau_partage[0], " et " , tableau_partage[1])
    print("la somme du tableau partage APRES les éincrmentations : %d (doit etre %d)"\
          %(sum(tableau_partage),nb_iterations*2) )
```

Éviter la SC : Solution alternative (suite)

- La trace :

```
"""
TRACE
le contenu du tableau_partage AVANT les éincrmentations :  [0, 0]
5000  et  5000
la somme du tableau partage APRES les éincrmentations : 10000 (doit etre 10000)
"""
```

☞ Rappel de la syntaxe "multiprocessing":

```
pid1=mp.Process(target=count3_on_travaille_dans_un_array, args=(nb_iterations,))
pid1.start()
pid2=mp.Process(target=count3_on_travaille_dans_un_array, args=(nb_iterations,))
pid2.start
# .... On les laisse bosser

#
pid1.join();
pid2.join()
```

Échange avec Queue (get / put)

Un exemple simple :

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())
    p.join()
#  Trace :  "[42, None, 'hello']"
```

Echange avec Pipe (send / recv)

Caractérisé avec send/recv

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())
    p.join()

# prints "[42, None, 'hello']"
```

A propos de Value, Array et lock à leur création

- Les variables partagées font souvent l'objet d'une SC
  - ➛ On les protège par un sémaphore / Lock
- Il est possible de déléguer cette protection
  - ➛ Lors de la création d'une variable partagée, on peut spécifier un Lock.

A propos de Value, Array et lock à leur création (suite)

Exemple : l'écriture :

```
...
un_entier = mp.Value('i', 7)        # Par édfaut =  je gere le SC moi-êmme
mutex=mp.Lock()

# Je modife la variable dans une SC : donc il faut un verrou mutex
with mutex :
    un_entier += 42
```

Devient :

```
...
mutex=mp.Lock()
un_entier = mp.Value('i', 7, lock = mutex)

# Je modife la variable dans une SC SANS M'OCCUPER de mutex
un_entier += 42
```

Un exemple : ../..

A propos de Value, Array et lock à leur création (suite)

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

def modifier_les_vars_partagees(un_entier, un_reel, un_str):
    un_entier.value **= 2  # Lever au carre
    un_reel.value **= 2     # Lever au carre
    un_str.value = un_str.value.upper()  # Mettre en majuscule

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

    un_entier = Value('i', 7)
    un_reel = Value(c_double, 1.0/3.0, lock=False)  # «– remarquer l'absence de verrou
    un_str = Array('c', b'hello CPE', lock=lock)     # «– remarquer le verrou

    p = Process(target=modifier_les_vars_partagees, args=(un_entier, un_reel, un_str, ))
    p.start()
    p.join()

    print(un_entier.value)
    print(un_reel.value)
    print(un_str.value)
```

A propos de Value, Array et lock à leur création (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

- On reprend le problème de somme d'un tableau
- Cette fois, le tableau est un *Shared Array*
  ➛ Plus efficace qu'avec **list / array / 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 : édiffrent 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 épass en arg de cette fonc)
```

Mémoire partagée (suite)

- La partie principale

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

    lock=Lock() # écration 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
```

Mémoire partagée (suite)

```
# éCration de la émmoire épartage
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 (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 Shared Array !

Exercice : allocation de ressources

On souhaite réaliser l'exemple suivant :

- N processus (p. ex. $N = 4$) ont besoin chacun d'un nombre k d'une ressource (p. ex. des Billes) pour avancer leur travail
- Cette ressource existe en un nombre limité : on ne peut satisfaire la demande de tout le monde en même temps.

  Par exemple, la demande de $Process_{i=1..4}$ est de $(4, 3, 5, 2)$ billes et on ne dispose que de $nb\_max\_billes = 9$ billes
- Chaque Processus répète la séquence (p. ex. m fois) :

  "demander k ressources, utiliser ressources, rendre k ressources"
- Le "main" crée les 4 processus.

  Il crée également un processus *controleur* qui vérifie en permanence si le nombre de Billes disponible est dans l'intervalle $[0..nb\_max\_billes]$

Exercice : allocation de ressources (suite)

- Pour chaque P_i, l'accès à la ressource se fait par une fonction "demander(k)" qui doit bloquer le demandeur tant que le nombre de billes disponible est inférieur à k
- P_i rend les k billes acquises après son travail et recommence sa séquence

Psoeudo algorithmes :

- Main :

```
MAIN :
    Creer 4 processus travailleurs (avec mp.Process)
    Lancer ces 4 processus
    Creer un processus controleur
    Lancer controleur
    ... tourner les pouces un peu ...
    Attendre les fin des 4 orocessus
    Terimer le processus controleur
```

Exercice : allocation de ressources (suite)

- Travailleur :

```
Travailleur(k_bills):
    Iterer m fois :
        demander k_bills
        simuler le travail avec un delai (sleep s sec.)
        renre k_bills
```

- Demander k billes :

```
Demander(k_bills):
    Consulter nbr_disponible_billes (dans une SC)
    Si nbr_disponible_billes < k_bills :
        se bloquer (sur un semaphore)
    nbr_disponible_billes = nbr_disponible_billes – k_bills
```

- rendre k billes :

```
rendre(k_bills):
    Dans une SC :
        nbr_disponible_billes = nbr_disponible_billes + k_bills
```

Exercice : allocation de ressources (suite)

☞ Noter que **Demander(k_bills)** sera équivalent à **sem.acquire(k_jetons)** (fonction qui n'existe pas dans le package multiprocessing) !

☞ De même pour **rendre** et **release()**

• Contrôleur :

```
Controleur(max_billes):
    Iterer toujours :
        Dans une SC :
            Verifier que 0 <= nbr_disponible_billes <= max_billes
        delai(1 sec)
```

Exemple Pool : somme d'un tableau

Remarques sur la valeur de retour d'un processus

• Pour obtenir une valeur de retour d'une fonction exécutée par un process, Python propose différentes solutions.

- Value
- Array
- Shared Array
- Gestion des retours via un Pool de processus
- Gestion via 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 est en mesure de récupérer les résultats renvoyés par les processus.

A noter : *Pool* exige la présence d'une section *__main__* qui sera importée par les fils.
Cela signifie que les exemples utilisant *multiprocessing.pool.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.

- Le tableau dont on veut la somme est global
- Chaque processus s'occupe de faire la somme d'une tranche.
  Le processus récupère les indices de début / fin de sa tranche qui lui donnent accès à deux listes dédiées *debut_tranches* et *fin_tranches*,
- Les processus sont ici organisés dans un pool.
- Chaque processus termine avec la clause "return" et renvoie son résultat
- Le **Pool** est capable, par son mécanisme interne, de récupérer ces résultats.

Pool : exemple somme (suite)

- Le code de chaque processus qui faire la somme d'une tranche (slice) :
  - Le tableau à "sommer" est une variable globale et remplie de 10^6 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 # section critique (mm épass en arg de cette fonc)
```

Pool : exemple somme (suite)

- Le Main :

```
if __name__ == "__main__" :
    print("------------ Avec Pool de process")
    Nb_process = os.cpu_count() - 2

    une_part = N // Nb_process

    t_start = time.time()
    somme_cas_multi_processus_avec_Pool = 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

    debut_tranches=[fin_tranches[i-1] for i  in range(1,Nb_process)]
    debut_tranches.insert(0,0)

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

    print("debut_tranches :", debut_tranches, "\nfin_tranches :", fin_tranches, "\n")

    with  Pool(Nb_process) as p:
        res = list(p.map(somme, indices))

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


    t_end = time.time()
```

Pool : exemple somme (suite)

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

NB : Le mécanisme *concurrent.futures.ProcessPoolExecutor* est équivalent à Pool et englobe et utilise un Pool de processus. Il a l'intérêt de mieux simplifier les écritures mais a 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 érponse")
    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())
```

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 érponse
1  :  je renvoie la érponse
2  :  je renvoie la érponse
3  :  je renvoie la érponse
4  :  je renvoie la érponse
[True, False, True, False, True]
"""
```

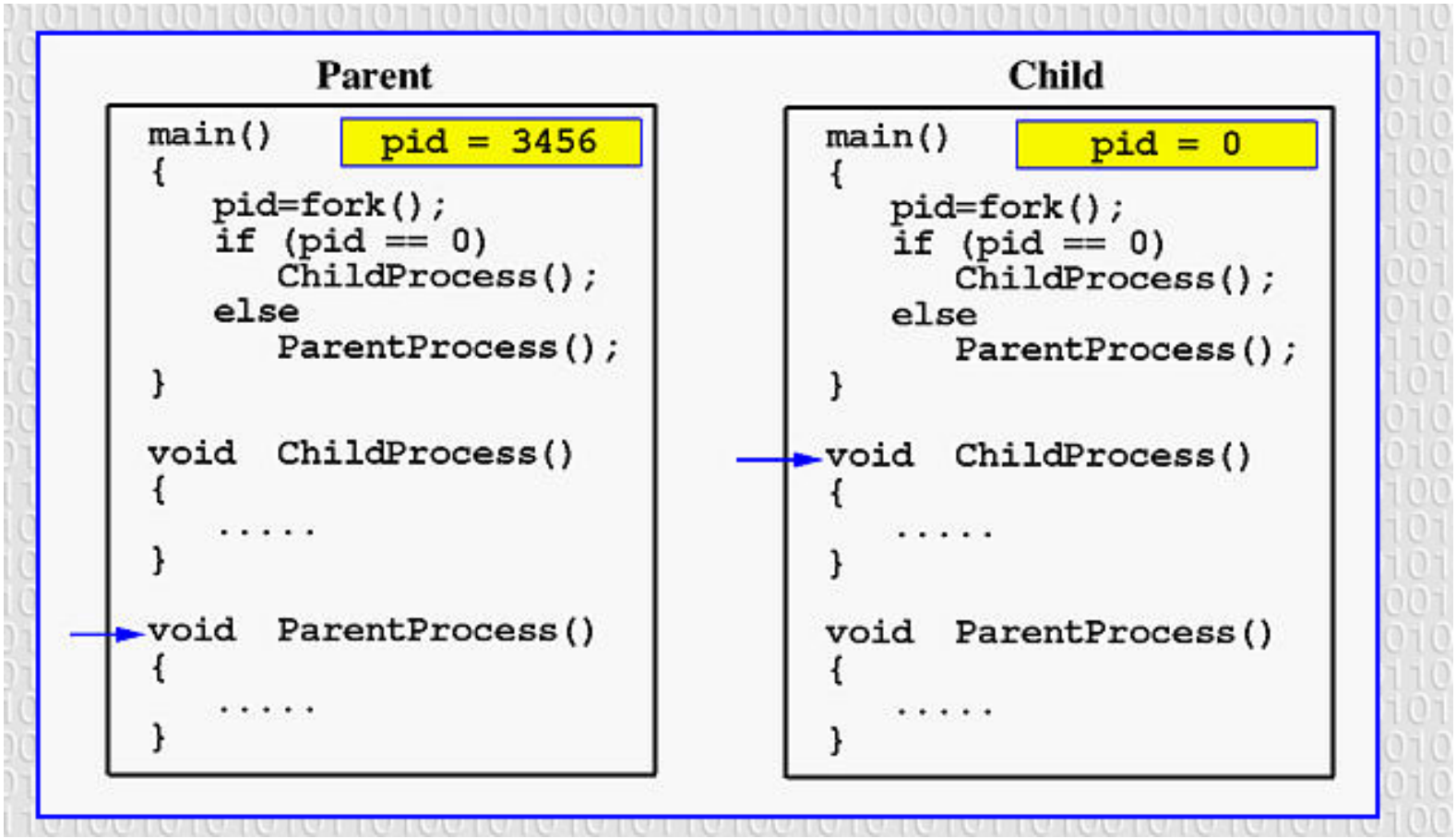
Rappel Détails du Fork

- Un processus peut être créé par un autre (père - fils)
- Le fils peut à son tour en créer d'autres
  - graphe du père et ses descendants = une structure partiellement ordonnée.
- On appelle parfois "job" l'ensemble du processus père et ses descendants

```
idfils = fork ();
if (idfils == 0)                    // en est dans le fils
    <Donner quelque chose à faire au fils>
else                                 // on est dans le père
    <continuer la travail du père>
```

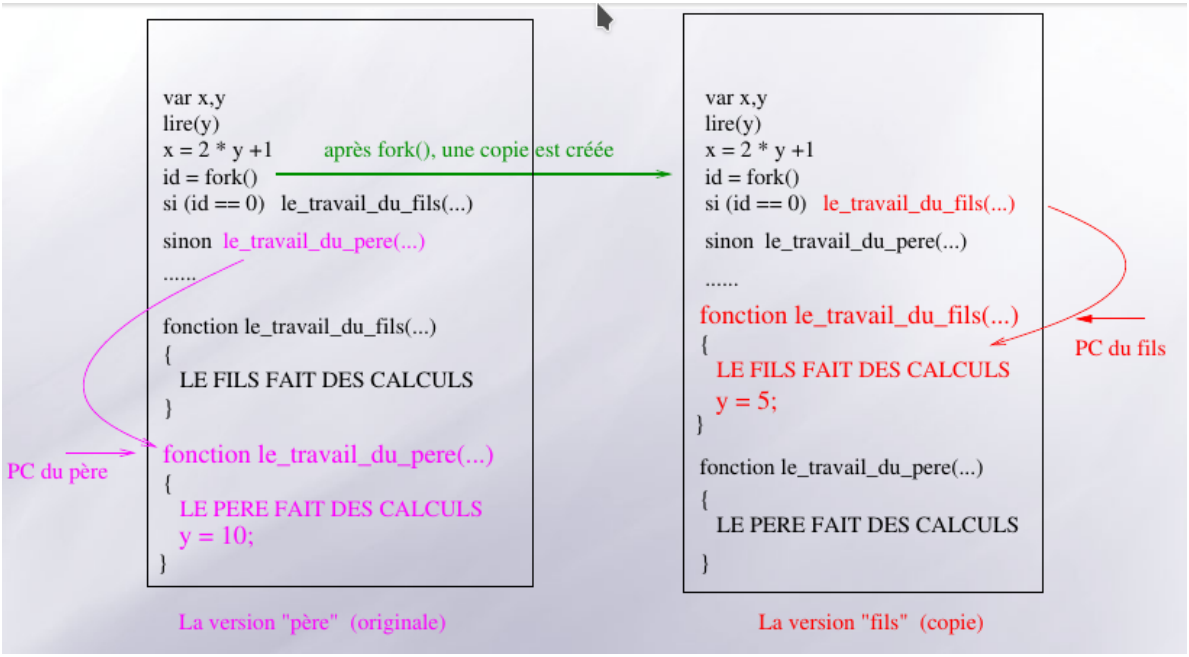
Rappel Détails du Fork (suite)

Comprendre le *Fork* :



Rappel Détails du Fork (suite)

- Fork recopie tout :



- En fait, le **code** n'a pas besoin d'être recopié (ne change pas !)
  - il suffit de donner un ptr d'instruction à chacun (program_counter=PC)
- Mais les **données** sont copiées : chacun pour soi (chacun son y).

Rappel Détails du Fork (suite)

Que fait-on au retour d'un *fork* (où le père s'est cloné !) ?

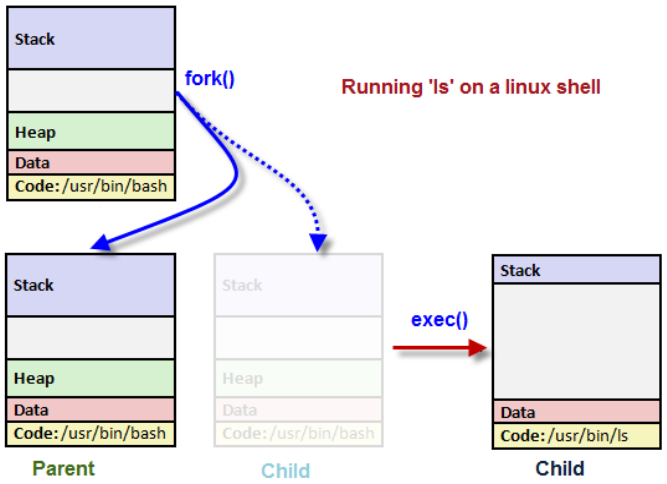
- On peut laisser les 2 codes se dérouler en parallèle faisant (une partie) de la même tâche (suivant des paramètres)
- Le fils peut s'enfermer dans une fonction du programme (e.g. surveillance d'un capteur)
- On peut utiliser la commande "**exec()**" pour exécuter un code différent

- Si **exec()** (où des paramètres d'appel peuvent être passés au fils) :
  - "exec" remplace le code + données du processus qui l'exécute par ceux du code chargé
  - le processus reste le même mais le programme exécuté change
  - "fork" + "exec" : exécution en parallèle (du père et du fils).

Rappel Détails du Fork (suite)

Exemple (d'exec)

```
idfils = fork ();
if (idfils == 0)                    // en est dans le fils
    exec(fichier disque)
else                                 // on est dans le père
    <continuer la travail du père>
```



Rappel Détails du Fork (suite)

- Selon les systèmes, on peut visualiser la liste des processus :
  - Unix (MacOs, Linux, etc.) : *ps*
  - Windows : passer par l'interface graphique
  - ➛ On peut en arrêter (*kill, signal*)
- Au **boot**, un processus spécial (*init*) est l'ancêtre des autres
  - Il peut lancer un processus spécialisé "lanceur de programmes"

Pseudo code du lanceur d'application P

```
Id : process-id;
if   demande de lancement d'un programme P
then  Id=fork();
      if (Id == 0)            % le fils
      then exec(P);       % attribuer P au demandeur (user)
endIf
```

  - La demande est traitée puis le lanceur reste dans son itération.
- Bien entendu, une application quelconque peut elle aussi faire des *fork*s.

TabMat