Introduction à la programmation concurrente

Programmation Concurrente en Python (Cours 1)

Quelques notions sur les activités parallèles

Alexander Saidi
ECL - LIRIS - CPE

Nov. 2022

Introduction

- **Nécessité** de la programmation Concurrente
  liens avec le *Big-Data*
- **Dans quels cas utiliser la programmation parallèle ?**
  - Des applications où il y a de la concurrence d'activités
    (e.g. schéma *Prod / Conso* / Client-serveur)
  - Activités décomposables en *tâches* (quasi) indépendantes
    (traitement d'images / texte / signal, calculs divers e.g. PI, Fib, MCL, ...)
  - Des applications où les E/S ne doivent pas être bloquantes
    ➛ serveurs, requêtes sur sites , relevée de capteurs, ...

Introduction (suite)

- Notions : *Multi tâches* vs. *mono tache*
- La programmation parallèle réalisée par :
  - **Processus** et / ou
  - **Threads**

☞ Tout application (programme) lancée coûte 1 processus.

➛ Choix entre *threads* et *processus* (cas Python)

- Liens avec l'embarqué, temps réel, ...
- Formalisation :

  l'algèbre des tâches et l'indépendance des activités.

Introduction (suite)

- Processus Mono et multi-thread

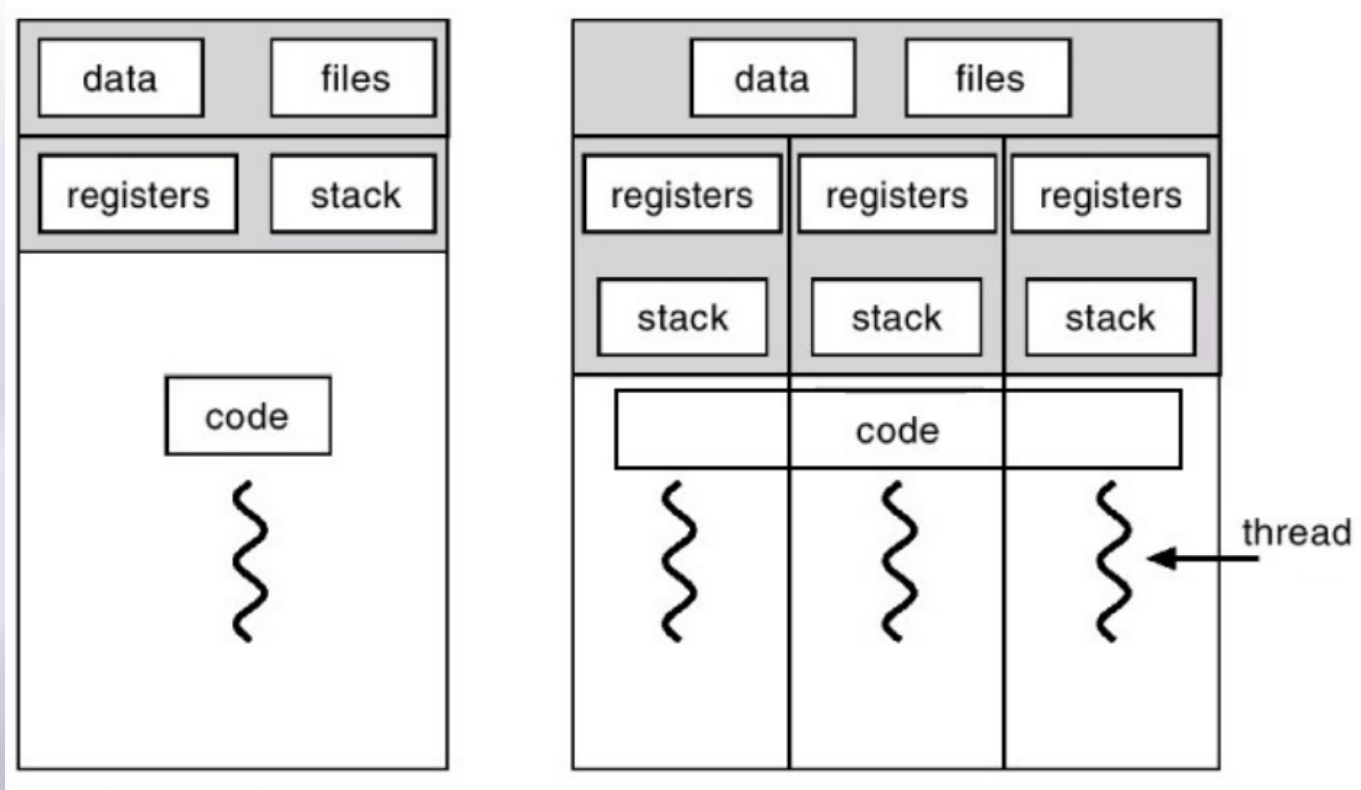
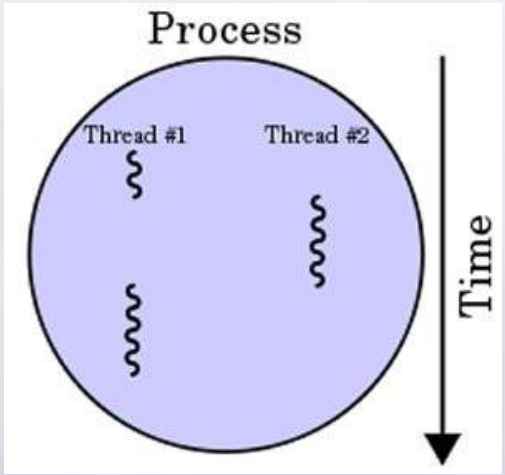


FIGURE 1 – Processus Mono / multi-threads

Introduction (suite)



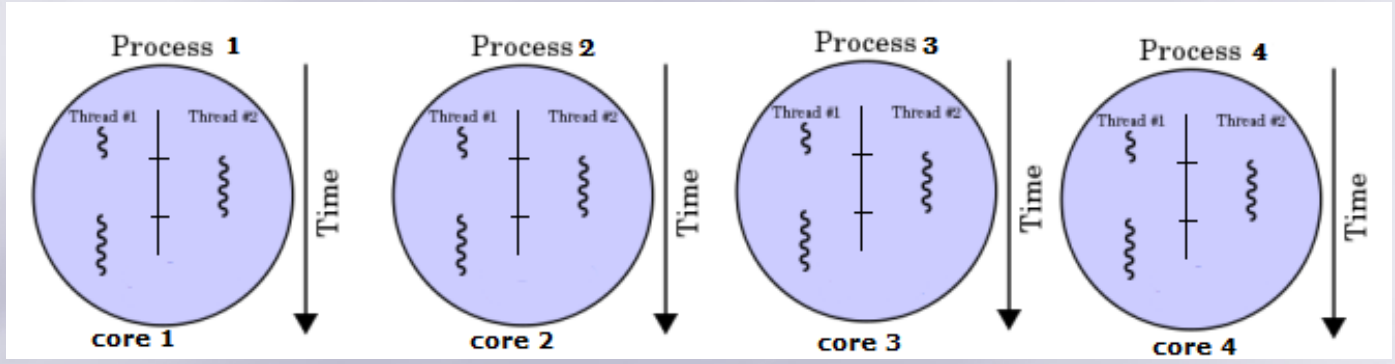


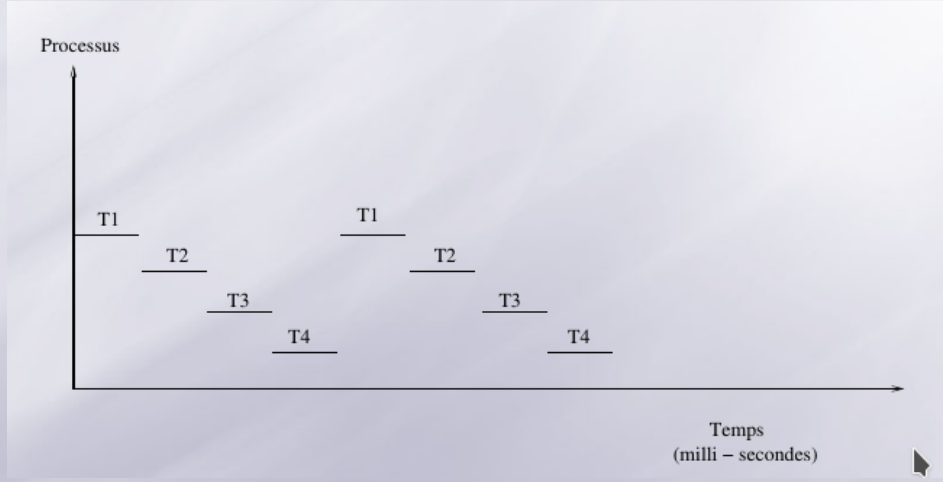
FIGURE 2 – Thread vs. Processus sur une machine Mono & multi-coeurs

Introduction (suite)

C'était comment (Monoprocesseur / Mono corps) ?

Comment fait-on si plusieurs processus tournent sur 1 processeur ?

Gestion multi-tâches sur un processeur (le pseudo parallélisme)



- Il peut y avoir différentes stratégies d'allocation du temps.
  - ➛ *Ordonnancements* divers (CPU, mémoire, Disques, autres ressources)

Introduction (suite)

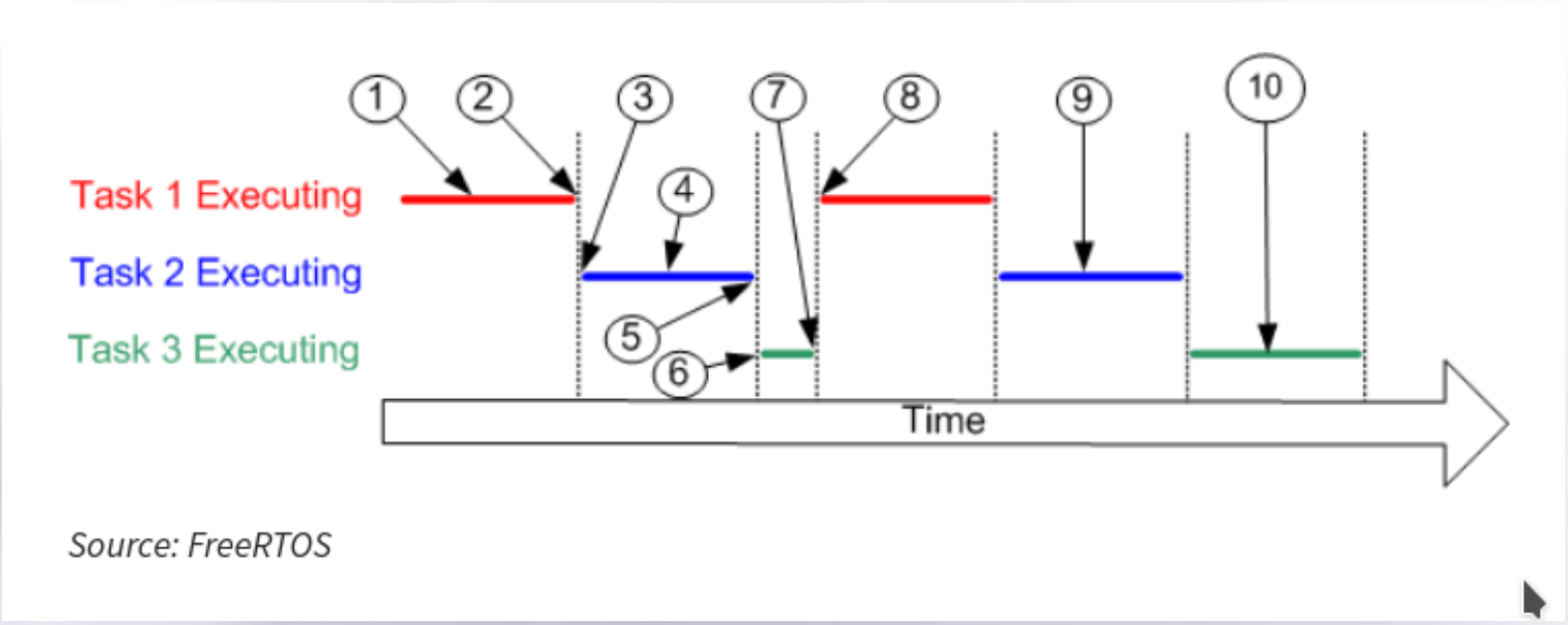


FIGURE 3 – Temps partagé sur un système Temps Réel mono processeur

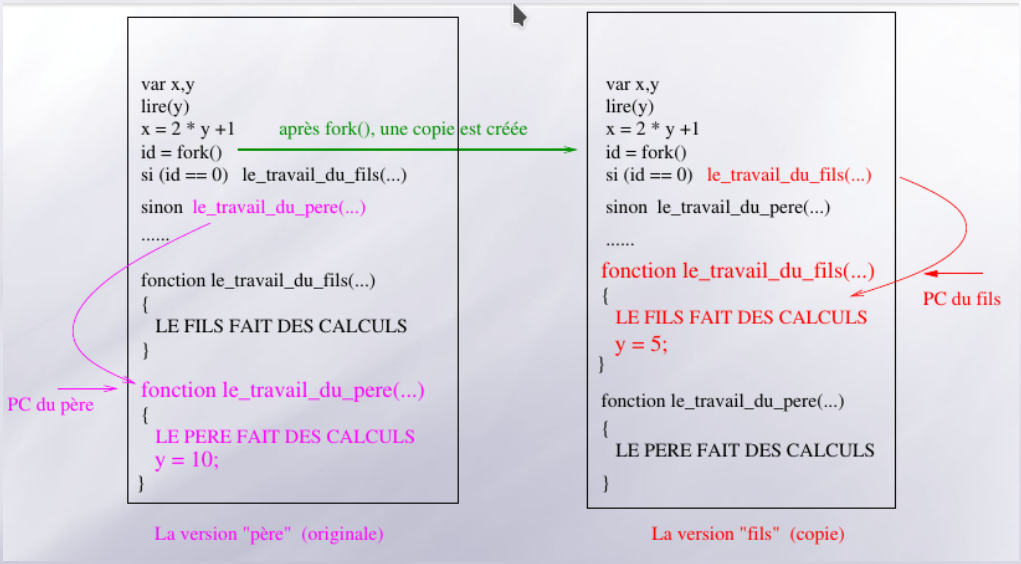
1,4,9,10 : tâches en exécution;

2,5,7 : commutation;

3,6, 8 : reprise/début de tâche

Rappel Détails du Fork

- La commande **fork()** recopie tout :



- En fait, le **code** n'a pas besoin d'être recopié (ne change pas !)
  - il suffit à chacun d'avoir une copie de son PC (program_counter)
- Mais les **données** sont copiées : chacun pour soi (chacun son x,y).
- Le graphe des processus = une structure partiellement ordonnée.

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 "**exec()**" pour exécuter un code différent

- Si **exec(prog)** (*execlp/execvp/...* + paramètres) :
  - "prog" remplace le code + données de l'appelant
  - le processus reste le même mais le programme exécuté change

Rappel Détails du Fork (suite)

Exemple (le fils exécute un code ordinaire)

```
idfils = fork ()
if (idfils == 0) : <squénce ordinaire de code> # (copie fils)
else : <continuer la travail du père> # (copie père)
```

Exemple (le fils appelle une fonction (dans le même fichier / module importé)

```
idfils = fork ()
if (idfils == 0) : surveiller_capteurs() # (copie fils)
else : <continuer la travail du père> # (copie père)
```

Exemple (le fils exécute execlp(..) et quitte ce code)

```
idfils = fork ()
if (idfils == 0) : execlp(...) # (copie fils)
else : <continuer la travail du père> # (copie père)
```

☞ **Que ce passe-t-il si** *execlp()* **relance ce même code ?**

Rappel Détails du Fork (suite)

L'évolution de la mémoire suite au *fork()* et *exec()*

- Duplication (des données) après *fork()*
- Remplacement du code du fils après *exec()*

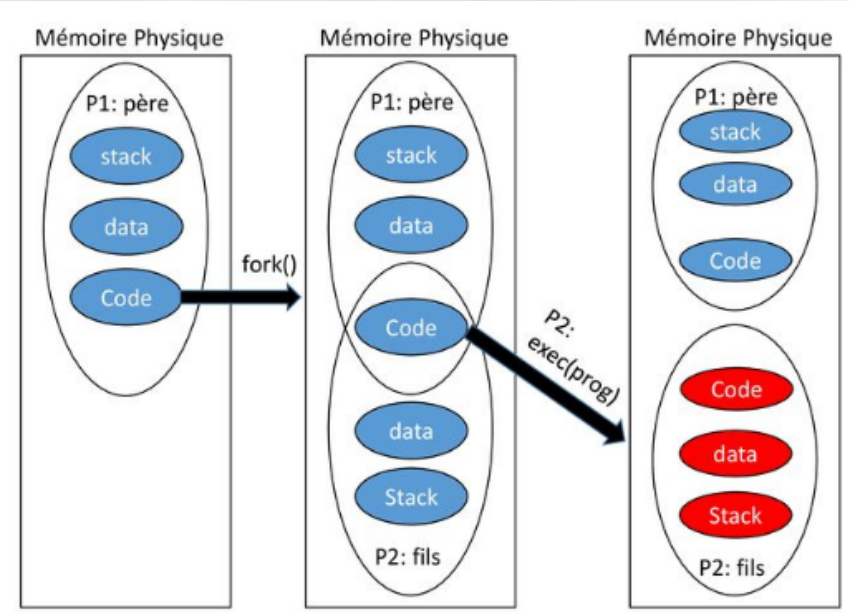


FIGURE 4 – Thanks to A. Talbi (ESME)

Exemple de graphe de fork

```
for i in range(4) :  fork();
# Equivalent à :
fork();
fork();
fork();
fork();
```

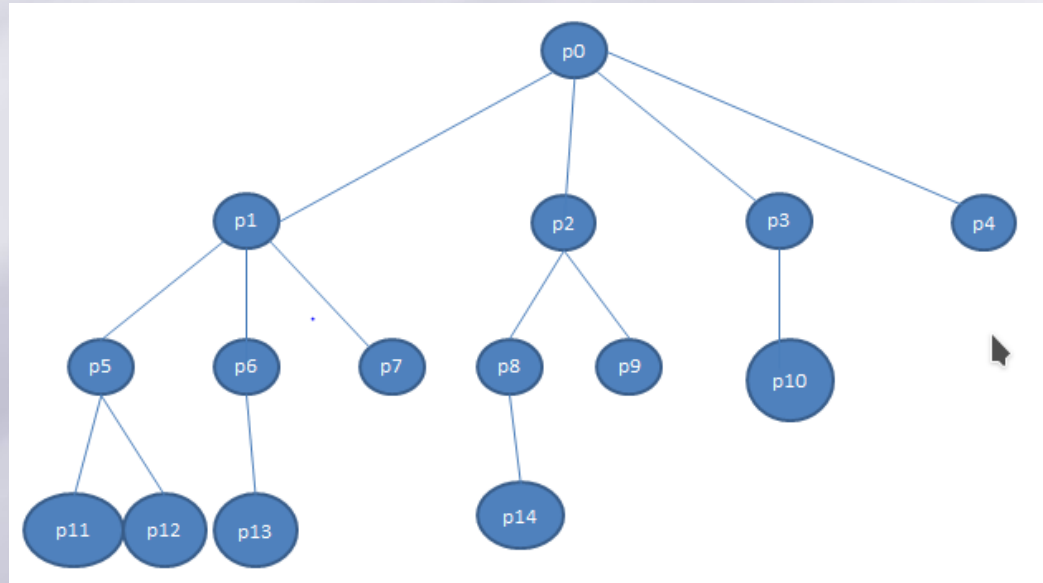
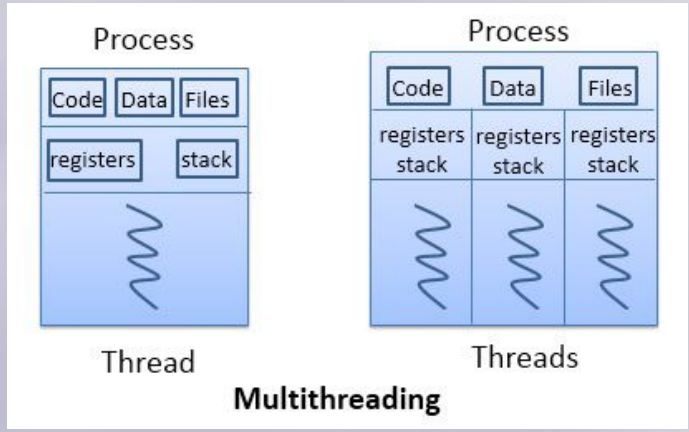


FIGURE 5 – graphe des forks par les pères et fils

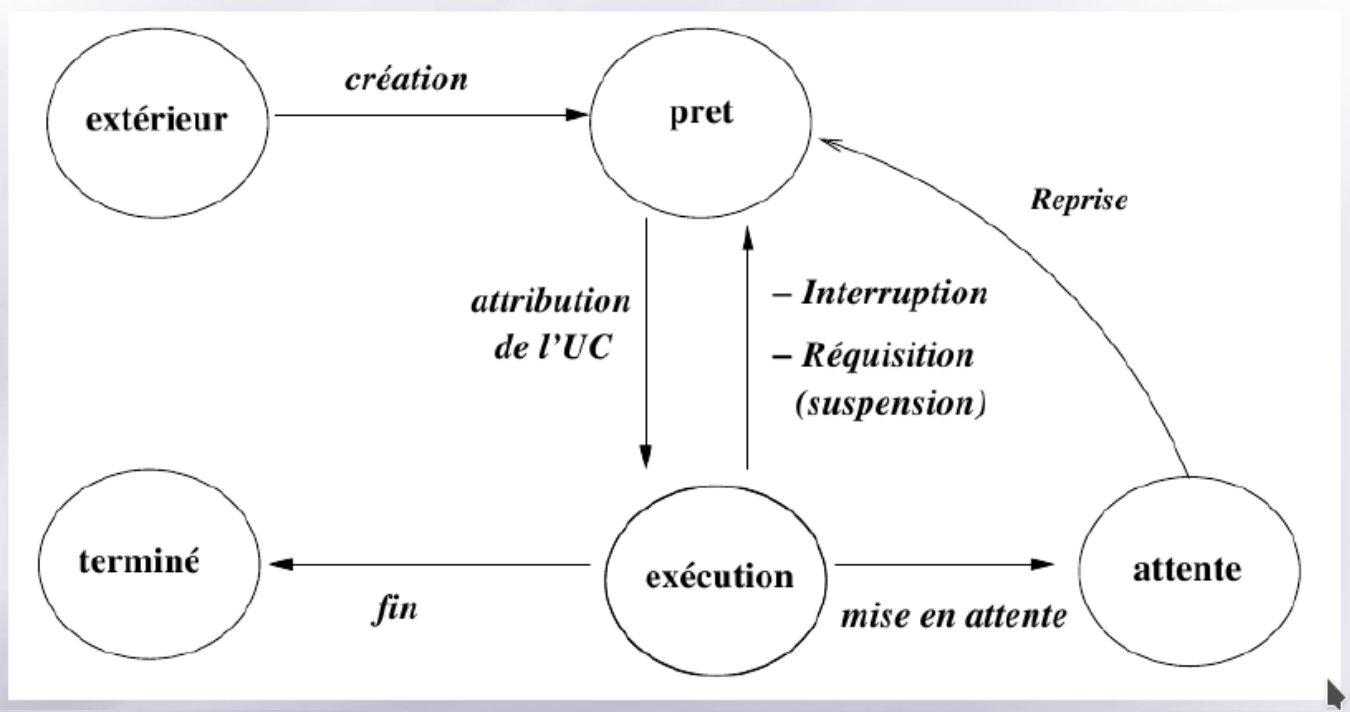
Python, Threads et GIL

- Les threads sont plus réactifs
- Le partage des ressources y est plus simple
- Mais Python n'a pas de Threads parallèles !
- Cas particulier de Python (GIL : Global Interpreter Lock)
- Néanmoins, ses Threads sont utiles dans "quelques" de cas !



Etats d'un processus / thread

- Diagramme simplifié



MP : Quelques exemples introductifs

Nous utiliseront le module multiprocessing de Python3.

- Somme des des éléments d'un tableau
- Calcule de la valeur approchée de π
- Incrémentation (section critique) avec *Lock* et *Semaphore*

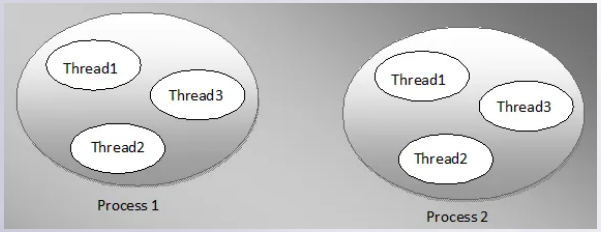
☞ NB : limite de création de processus : *cat /proc/sys/kernel/pid_max*

➛ 32768 sous Ubuntu 18/Intel I7

Ex1 : somme PIle d'un tableau

Somme des éléments d'un tableau

- On décide de faire la somme des éléments d'un grand tableau (liste) par deux processus.
- Les codes du fils et du père sont indépendants :
  - ➛ il n'y aura pas de **variable globale** !
- Il n'y a pas de **return** vers le père (non plus) !
- Comment faire pour assembler les deux sommes partielles ?
  - ➛ mp.Value()
  - ➛ Queue et Pipes (modules *mp* ou *os*)
  - ➛ Autre possibilités (voir plus loin)



L'OS voit 2 processus indépendants

(on n'aura pas de threads)

Ex1 : somme PIle d'un tableau (suite)

Somme : Version parallèle (reprise pour une comparaison plus loin).

- La fonction qui réalise la somme d'une tranche du tableau :

```
import os
import multiprocessing as mp # pour Value

def somme(num_process, Val, tableau) :
    print("Je suis le fils num ", num_process, "et je fais la somme du tableau ", tableau )
    S_local=0
    for i in range(len(tableau)) :
        S_local += tableau[i]
    Val.value += S_local
```

- Cette fonction fait la somme des éléments du paramètre *tableau*.
- Pour communiquer ses résultats, elle verse "sa" somme dans *Val*
- Si cette fonction fait *"return S_local"*, qui réceptionne la valeur ?
- Quel est le problème ?
  - ➛ la transmission d'un résultats ? par "return" ?

Ex1 : somme PIle d'un tableau (suite)

- La fonction principale crée un fils pour "sommer" $\frac{1}{2}$ du tableau
  - ➛ Elle-même fait l'autre moitié (car elle coûte un processus !)

```
if __name__ == "__main__" :
    taille = 15
    tableau = [i for i in range(taille)]  # remplissage du tableau

    somme_totale = mp.Value('i', 0)
    # Création et lancement du fils
    id_fils = mp.Process(target=somme,args=(1, somme_totale, tableau[:taille // 2],))
    id_fils.start()

    somme(0, somme_totale, tableau[taille // 2:])
    # Attente de la fin du fils
    id_fils.join()
    print("La somme totale du tableau (par mp.process) : ", somme_totale.value)
    print("Comparer avec La somme calculée par Python  : ", sum(tableau))
"""
Je suis le fils num  0 et je fais la somme du tableau  [7, 8, 9, 10, 11, 12, 13, 14]
Je suis le fils num  1 et je fais la somme du tableau  [0, 1, 2, 3, 4, 5, 6]
La somme totale du tableau (par mp.process) :  105
Comparer avec La somme calculée par Python  :  105
"""
```

Comparaison avec la version séq.

- **Où est le gain** (vs. une version séquentiel) ?
- Comparaison du temps moyen en fonction du nombre de processus

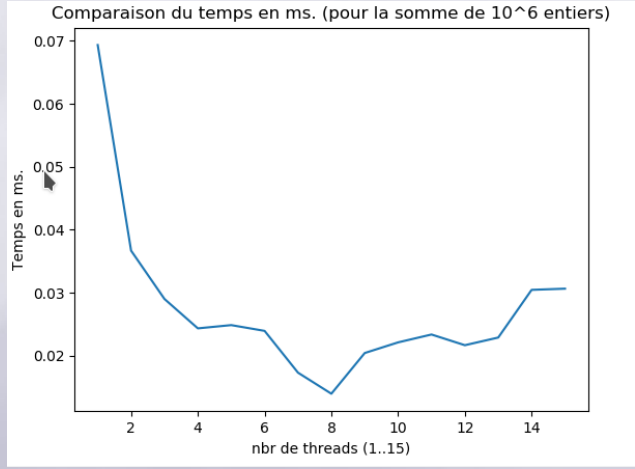


FIGURE 6 – moyennes des temps sur 50 itérations sur un Intel-I7

☞ Le gain est assez important pour ce calcul O(N).

➛ Le gain sera moins fort pour des complexités supérieures.

Comparaison avec la version séq. (suite)

Quelques remarques sur l'exemple *somme* :

- On peut généraliser en créant plusieurs processus avec un tableau de taille très grande
- Chaque *processus* calcule sa "part" dans la variable `somme_locale` puis verse ce résultat dans la variable globale `somme_totale`.
- La fonction *main* attend la fin de chaque fils avec **wait/join**.

  ☞ Sans *wait()/join()*, **main** se termine avant que les threads aient fait leur travail.

  ➛ La Somme totale sera alors erronée.
- Il y a un risque (faible mais non nul) que les processus se marchent sur les pieds lors la manipulation de `somme_totale`.

  ➛ Différentes solutions : *exclusion mutuelle*, Pipe, ...

Somme avec array et Pipe

On revisite l'exemple somme (vars une version plus efficace) :

- Le main crée deux fils
- Utilisation de "array" (ne pas confondre avec "mp.Array")
- Résultats transmis via un "mp.Pipe"

- Le code de la fonction qui somme une tranche du tableau :

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

# La fonction des 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 ", tableau[debut: fin_exclue] )
    S_local=0
    for i in range(debut, fin_exclue) :
        S_local += tableau[i]

    # send() n'est pas bloquant
    pour_fils_to_send.send(S_local)       # «— Chaque fils transmet son résultat
```

Somme avec array et Pipe (suite)

- Le **main**

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

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

    pour_pere_to_receive, pour_fils_to_send=mp.Pipe()

    id_fils1 = mp.Process(target=somme,args=(1, tableau, 0, taille // 2, pour_fils_to_send,))
    id_fils2 = mp.Process(target=somme,args=(2, tableau, taille // 2, taille, pour_fils_to_send,))
    id_fils1.start()
    id_fils2.start()

    moitie1=pour_pere_to_receive.recv() # Blocks until there is something to receive.
    moitie2=pour_pere_to_receive.recv()

    # On laisse "join()" mais inutile dans ce cas car "recv()" est bloquant et les fils terminent par send()
    id_fils1.join(); id_fils2.join()

    print("La somme totale du tableau oobtenue : ", moitie1+moitie2)
    print(f"On vérifie que la somme par Python : {sum(tableau)}")
```

Somme avec array et Pipe (suite)

- Trace : test avec un petit tableau (array).

TRACE :
array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

Je suis le fils num 1 et je fais la somme du tableau array('i', [0, 1, 2, 3, 4])
le fils num 1, envoie par send 10

Je suis le fils num 2 et je fais la somme du tableau array('i', [5, 6, 7, 8, 9])
le fils num 2, envoie par send 35

La somme totale du tableau oobtenue : 45
On vérifie que la somme par Python : 45

Somme avec array et Pipe (suite)

- Trace : test avec un tableau de 10 puis 10^6 éléments.

  ➛ On a supprimé les print() des tableaux.

```
"""
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 2, envoie par send 374999750000
le fils num 1, envoie par send 124999750000
La somme totale du tableau oobtenue :  499999500000
On vérifie que la somme par Python : 499999500000
"""
```

Comparaison avec séquentielle

Petite comparaison avec la version séquentielle :

- Version séquentielle : on appelle simplement 2 fois la fonction somme.
  - ➛ Code à réaliser en exercice (avec le nbr. processus en paramètre).
- La trace (pour 2 processus) : on a un gain intéressant

```
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 2, envoie par send 374999750000
le fils num 1, envoie par send 124999750000
La somme totale du tableau obtenue :  499999500000  en 323747 msec
On vérifie que la somme par Python : 499999500000
-----------------------------------------------
Version séquentielle :
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 581986 msec.
```

Calcul de la somme avec Queue

On reprends l'exemple du calcul de la somme d'un grand tableau (ici *array*) utilisant une Queue.

- Le tableau est partagé via un array.
- La seule différence par rapport à la version avec Pipe :

  get / **put** au lieu de **send** / **recv**.

```
import array, os, time
import multiprocessing as mp

# La fonction des fils
def somme(num_process, table, debut, fin_exclue, fil_des_resultats) :
    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]

    fil_des_resultats.put(S_local) # Non bloquant
    print(f"le fils num {num_process}, envoie par put {S_local}")
```

Calcul de la somme avec Queue (suite)

La partie main :

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

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

    fil_des_resultats=mp.Queue()

    deb=time.time()
    id_fils1 = mp.Process(target=somme,args=(1, tableau, 0, taille // 2,fil_des_resultats))
    id_fils2 = mp.Process(target=somme,args=(2, tableau, taille // 2, taille,fil_des_resultats))

    id_fils1.start(); id_fils2.start()

    moitie1=fil_des_resultats.get() # Block jsq à ce que qq chose soit dispo
    moitie2=fil_des_resultats.get()
    # On laisser "join" mais inutile dans ce cas car "recv()" est bloquant et les fils terminent
    # avec send (non bloquants)
    id_fils1.join(); id_fils2.join()
    fin=time.time()
    print("La somme totale du tableau obtenue : ", moitie1+moitie2, " en ", int((fin-deb)*1000000), "msec.")
    print(f"On vérifie que la somme par Python : {sum(tableau)}")
```

Calcul de la somme avec Queue (suite)

Suite "main" : la partie séquentielle :

```
#-----------------------------------------------------------------------
print('-'*50)
print("Version séquentielle : ")
deb=time.time()
somme(1, tableau, 0, taille // 2, fil_des_resultats)
somme(2, tableau, taille // 2, taille,fil_des_resultats)
moitie1=fil_des_resultats.get() # Block jsq à ce que qq chose soit dispo
moitie2=fil_des_resultats.get()
fin=time.time()
print("La somme totale du tableau en version séquentielle : ", moitie1+moitie2, " en ",  int((fin-deb)*1000000),
"msec.")

"""
TRACE :
...
La somme totale du tableau obtenue :  499999500000  en 286344 msec.
On vérifie que la somme par Python : 499999500000
--------------------------------------------------
Version séquentielle :
...
La somme totale du tableau en version séquentielle :  499999500000  en 490479 msec.
"""
```

Résumé : mécanisme d'échange

Pour communiquer des résultats des processus :

1. par value :

```
import multiprocessing as mp

# Déclaration et initialisation
somme_totale = mp.Value('i'', 0)

# Utilisation
Val.value += S_local
```

2. par Array (ne pas confondre avec 'array')

```
import multiprocessing as mp

# Déclaration et initialisation
tableau_partage = mp.Array('i', 2)  # tableau de 2 entiers

# Utilisation
tableau_partage[i] = ...
```

Résumé : mécanisme d'échange (suite)

3. par Pipe :

```
import multiprocessing as mp

# Déclaration et initialisation
pour_pere_to_receive, pour_fils_to_send=mp.Pipe()

# Utilisation
pour_fils_to_send.send(S_local)  # send() n'est pas bloquant
moitie1=pour_pere_to_receive.recv()
```

4. par Queue :

```
import multiprocessing as mp

# Déclaration et initialisation
fil_des_resultats=mp.Queue()
# Utilisation
fil_des_resultats.put(S_local)  # send() n?est pas bloquant
moitie1=fil_des_resultats.get()
```

Résumé : mécanisme d'échange (suite)

- Nous avons utilisé le type **array** à la place de **list**.

```
import array
# Déclaration et initialisation
tableau = array.array('i',[i for i in range(taille)])
# Utilisation : comme une liste
tableau = valeur
```

- A la place de **'i'** (integer), on peut utiliser (voir https://docs.python.org/fr/3/library/array.html)

Code d'indication du type	Type C	Type Python	Taille minimum en octets	Notes
'b'	signed char	*int*	1	
'B'	unsigned char	*int*	1	
'u'	wchar_t	Caractère Unicode	2	(1)
'h'	signed short	*int*	2	
'H'	unsigned short	*int*	2	
'i'	signed int	*int*	2	
'I'	unsigned int	*int*	2	
'l'	signed long	*int*	4	
'L'	unsigned long	*int*	4	
'q'	signed long long	*int*	8	
'Q'	unsigned long long	*int*	8	
'f'	*float*	*float*	4	
'd'	double	*float*	8	

Résumé : mécanisme d'échange (suite)

Pour 'value', les types utilisables sont :

ctypes type	C type	Python type
c_bool	_Bool	bool (1)
c_char	char	1-character bytes object
c_wchar	wchar_t	1-character string
c_byte	char	int
c_ubyte	unsigned char	int
c_short	short	int
c_ushort	unsigned short	int
c_int	int	int
c_uint	unsigned int	int
c_long	long	int
c_ulong	unsigned long	int
c_longlong	__int64 or long long	int
c_ulonglong	unsigned __int64 or unsigned long long	int
c_size_t	size_t	int
c_ssize_t	ssize_t or Py_ssize_t	int
c_float	float	float
c_double	double	float
c_longdouble	long double	float
c_char_p	char * (NUL terminated)	bytes object or None
c_wchar_p	wchar_t * (NUL terminated)	string or None
c_void_p	void *	int or None

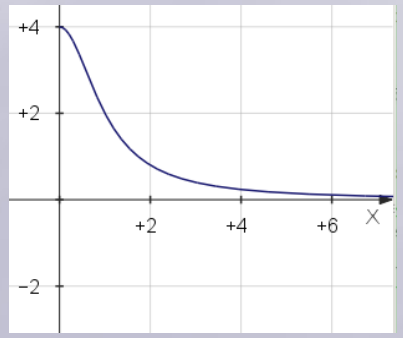
Ex2 : Calcul la valeur de π

- Un autre ex. de tâches décomposables : **Méthode arc-tangente** :

- On peut calculer une valeur approchée de PI par la méthode suivante :

$$\pi \approx \int_0^1 \frac{4}{1+x^2} dx$$ ou la version discrète $$\pi \approx 1/n \sum_1^n \frac{4}{1+x_i^2}$$

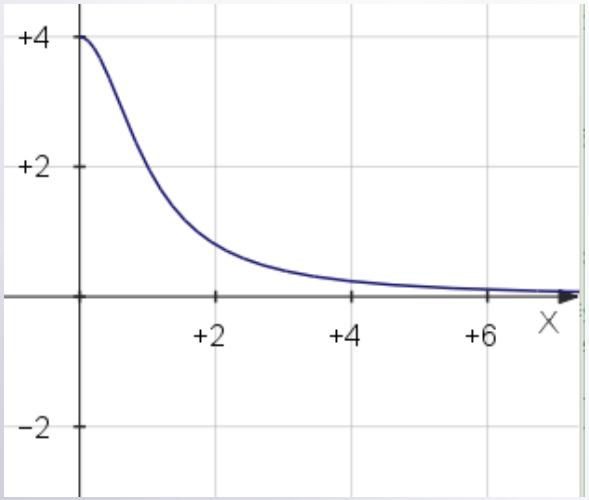
où l'intervalle $[0, 1]$ est divisé en n partitions (bâton) égales.



Ex2 : Calcul la valeur de π (suite)

N.B. : pour que la somme des bâtons soit plus proche de l'aire sous la courbe, considérons le milieu des bâtons :

$$\sum_{1}^{n} \frac{4}{1+x_i^2} \approx \sum_{i=1}^{n} \frac{4}{1+(\frac{i-0.5}{n})^2}$$
$$= \sum_{i=0}^{n-1} \frac{4}{1+(\frac{i+0.5}{n})^2}$$



➔ $(\frac{i-0.5}{n})$ ramène i dans $[0,1]$ (i.e. x_i)

☞ Cette méthode de discrétisation est proche des méthode des "Trapèzes" et de "Simpson".

Ex2 : Calcul la valeur de π (suite)

- La version séquentielle :

```
# Calcul de PI par Arctg
import multiprocessing as mp
import random, time

from math import *

def arc_tangente(n):
    pi = 0
    for i in range(n):
        pi += 4/(1+ ((i+0.5)/n)**2)
    return (1/n)*pi

if __name__ == "__main__" :
    nb_total_iteration = 1000000    # Nombre d'essai pour l'estimation

    start_time = time.time()

    result = arc_tangente(nb_total_iteration)
    print("Valeur estimée Pi par la méthode Tangente : ", result)
    print("Temps d'execution  séquentielle : ", time.time() – start_time)
```

- Pour le temps d'exécution, voir la version parallèle (avec 1 processus)

Ex2 : Calcul la valeur de π (suite)

La version parallèle :

- On décide de "faire faire" ce travail par k processus
- Chacun fera "sa part" (dans une variable locale);

  le père additionnera ces sommes partielles dans une variable visible par tous pour obtenir l'aire sous la courbe.
- La tâche (la fonction) de chaque processus est (ici) identique.

Pas de variable "globale" au sens habituel; chacun est chez soi.
Comment les processus transmettent leurs résultats ?

➛ Pourquoi "return" ne fonctionne pas ?

☞ "return" s'addresse ici au système d'exp.

➛ Utiliser un moyen de communication (IPC+synchro)

Ex2 : Calcul la valeur de π (suite)

Le code d'un processus :

```
def calculer_PI_arc_tangente(my_num, mon_nb_iter, integrale):
    pi = 0.0
    for i in range(0,mon_nb_iter): # Noter le pas de l'itération (entrelacement)
        pi += 4/(1+ ((i+0.5)/mon_nb_iter)**2)

     # Je verse mon résultat (Pi) dans la variable partagée
    integrale.value += (1/mon_nb_iter)*pi
```

Ex2 : Calcul la valeur de π (suite)

Le code du "chef" :

```
import multiprocessing as mp
import random, time, os

if __name__ == "__main__" :
    nb_processus=psutil.cpu_count() #8 sur un I7
    # Nombre d'essai pour l'estimation
    nb_total_iteration = 1000000
    integrale = mp.Value('f', 0.0)

    start_time = time.time()
    tab_pid=[0 for i in range(nb_processus)]
    for i in range(nb_processus) :
        tab_pid[i]=mp.Process(target=calculer_une_part_de_PI_arc_tangente, \
            args=(i+1, nb_total_iteration // nb_processus, integrale,))
        tab_pid[i].start()

    for i in range(nb_processus) :
        tab_pid[i].join()

    # On divise par nb_processus car chaque processus calcule PI
    print(f"Valeur estimée Pi par la méthode Arc-Tangente avec {nb_processus} processus: ", integrale.value/
   nb_processus)
    print("Temps d'execution : ", time.time() - start_time)
```

Ex2 : Calcul la valeur de π (suite)

- Trace avec différents nombres de processus

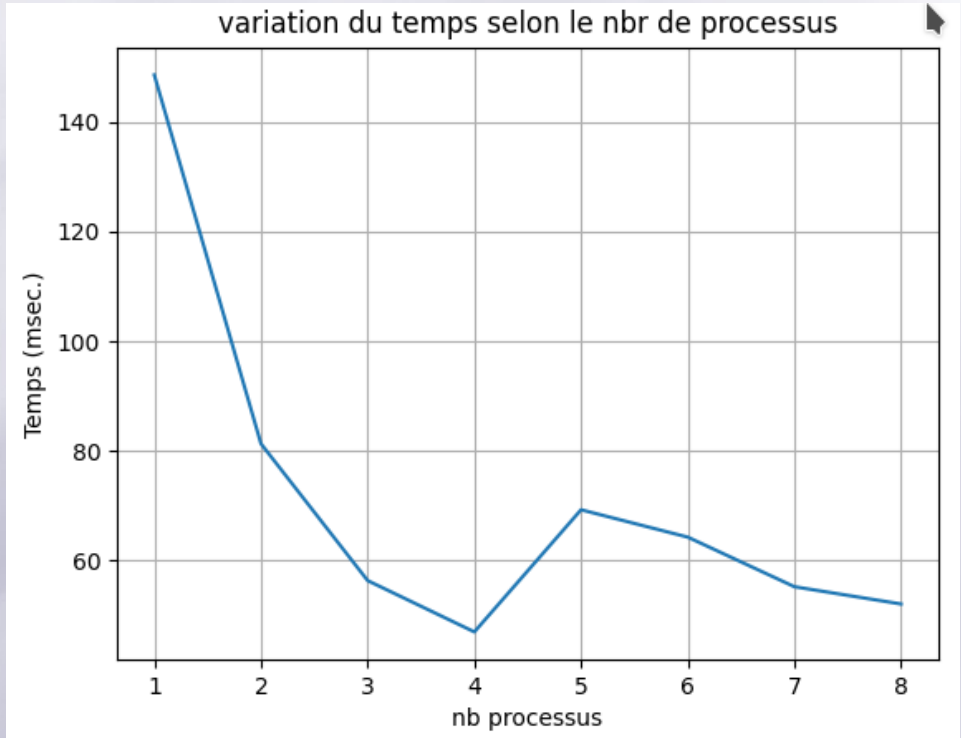
```
"""
Valeur estimée Pi par la méthode Arc-Tangente :  3.141648769378662
...
Temps d'execution :  0.1508796215057373
Temps d'execution :  0.0832059383392334
Temps d'execution :  0.05703926086425781
Temps d'execution :  0.04500699043273926
Temps d'execution :  0.06762290000915527
Temps d'execution :  0.061187744140625
Temps d'execution :  0.057219743728637695
Temps d'execution :  0.05214047431945801

"""
```

☞ Sur un Intel I7, les calculs sont ici (env.) > 3 **fois + rapides** qu'avec un seul processus.

Ex2 : Calcul la valeur de π (suite)

Comparaison des temps (1 à 8 processus)



Remarques

- Connaître le nombre de processus disponibles dans votre PC :

```
>>> import psutil
>>> try:
...     print(psutil.cpu_count())
... except :
...     pass
...
8
>>>
```

Remarques (suite)

☞ Toujours faire attention aux **variables partagées** modifiées par les processus (voir + loin).

- En cas de ressource unique + accès concurrent (en W/R) : PROTÉGEZ les accès à la variable partagée !

  ➛ Quel est le problème ?

  ➛ Un exemple de mauvais goût (mais efficace) : un slot du W.C. ouvert à tout vent !

Différentes mesures du temps

Voir aussi la documentation : https://docs.python.org/3/library/time.html

- **time.time()** : en secondes

```
>>> import time
>>> time.time()       #return seconds from epoch
1261367718.971009
```

- **time.time_ns()** : en nano secondes

```
>>> import time
>>> time.time_ns()
1530228533161016309

>>> time.time_ns() / (10 ** 9) # convertion vers   seconds
1530228544.0792289
```

Queue et Pipe en multiprocessing

- Deux moyen de communication.
- Un exemple : le père crée 2 fils et communique avec eux via un Pipe et une Queue.

```
import multiprocessing as mp

def fils1(file): file.put('hello')

def fils2(pour_fils_to_send): pour_fils_to_send.send('ECL')

if __name__ == '__main__':
    file = mp.Queue()
    pour_pere_to_receive, pour_fils_to_send=mp.Pipe()

    p1 = mp.Process(target=fils1, args=(file,));  p1.start()
    p2 = mp.Process(target=fils2, args=(pour_fils_to_send,));  p2.start()

    print("On a recu du fils 1 via la Queue " , file.get())

    str=pour_pere_to_receive.recv()
    print("On a recu du fils 2 via le Tubee " , str)
    p1.join() ;  p2.join()
```

Queue et Pipe en multiprocessing (suite)

- Trace

```
"""
TRACE
On a recu du fils 1 via la Queue  hello
On a recu du fils 2 via le Tubee  ECL
"""
```

☞ **Différence entre Pipe et Queue :**

- **Pipe()** a seulement deux extrémités (deux processus),
- **Queue()** peut avoir de multiples extrémités
  - ➛ On peut faire de multiple get() et put()

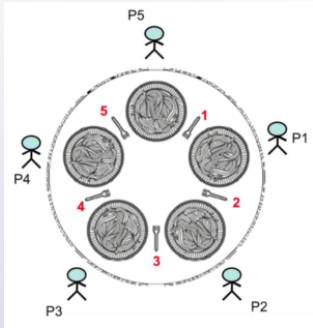
  cf. multiples producteurs et consommateurs

Accès concurrent à une ressource

(I) Un exemple classique : les **5-Φ**

Ils pensent / mangent / pensent / mangent / ...

- Pour manger, un philosophe a besoin de 2 fourchettes placées sur les deux côtés de son assiette.
  - ➛ P. ex., ϕ_2 aura besoin de f_2 et de f_3 pour manger.



- Une mauvaise gestion des fourchettes conduira à affamer un ou plusieurs philosophes,

(II) Places de parking

(III) Mme. PiPi !

etc ...

Accès concurrent à une ressource (suite)

Pour illustrer le propos : **les variables qui se marchent sur les pieds**:

Ex. : 2 processus incrémentent concurremment une variable globale.

- Qu'a-t-on à la fin dans cette variable ?
- Bien remarquer les valeurs affichées par les processus.

- Le code de l'incrémentation (simple) :

```
mport multiprocessing as mp

# Incrémentation avec protection de la variable partagée
def incrementer_la_variable_partagee(nb_iterations):
    """ Chacun incrémente la variable partagée """

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

Accès concurrent à une ressource (suite)

- Le main :

```
#---------- PARTIE principale (le point d'entrée de cet exemple) -------
if __name__ == '__main__' :
    nb_iterations = 5000

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


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

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

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

Accès concurrent à une ressource (suite)

- Quelques traces (plusieurs exécutions) :

```
"""
QQ traces
la valeur de variable_partagee AVANT les incrémentations :  0
la valeur de variable_partagee APRES les incrémentations 6120 (attendu 10000):

la valeur de variable_partagee AVANT les incrémentations :  0
la valeur de variable_partagee APRES les incrémentations 6496 (attendu 10000):

la valeur de variable_partagee AVANT les incrémentations :  0
la valeur de variable_partagee APRES les incrémentations 6819 (attendu 10000):

"""
```

➛ La somme n'est (presque) jamais bonne !

- Cause : voir plus loin l'anatomie d'une instruction d'incrémentation.

Explications

Pourquoi les incrémentations n'ont pas toutes lieu ?

il faut observer les endroits (instructions machines) où cette incrémentation peut être interrompue.

Exemple : soit une simple fonction d'incrémentation d'une variable x et la traduction (en byte-code ou pseudo-assembleur) de $x = x + 1$.

```
def incremeter(x) :
    x = x + 1
```

Puis

```
import dis      # pour voir les détails

dis.dis(incremeter)
```

Explications (suite)

On obtient (pour *x=x+1*):

```
0     LOAD_FAST        0 (x)      # x est à un décalage de 0 p/r au début de la zone des variables
3     LOAD_CONST       1 (1)       # charger la constate 1
6     BINARY_ADD                  # additionner
7     STORE_FAST       0 (x)      # stocker le résultat dans x


# Epilogue : préparer en renvoyer None (Une fonction Python renvoie tjs qq chose !)
10    LOAD_CONST           0 (None)
13    RETURN_VALUE
```

Le processus qui exécute ces instructions peut être interrompu à la fin de chacune.

➛ Anatomie d'une incrémentation...

Si on ne veut pas être interrompu pendant cette incrémentation, on devrait considérer l'incrémentation ($x = x + 1$) comme une action critique (sensible).

- On appellera cela une **section critique**.

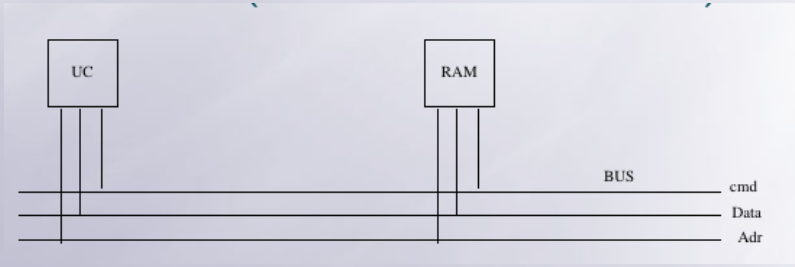
Anatomie d'une instruction

- Granulité d'activité d'un système (via "processus" / "tache", ...)

☞ Un exemple intuitif : qu'est-ce qui se passe lors de l'exécution de $N \leftarrow N + 1$ sur une machine basique (à *Accumulateur*) ?

- L'instruction est décomposée en (psoeudo-) Assembleur :

Load N, R1
Add R1, #1
Store R1, N



Anatomie d'une instruction (suite)

Exemple : cadencement de *'Load N'* :

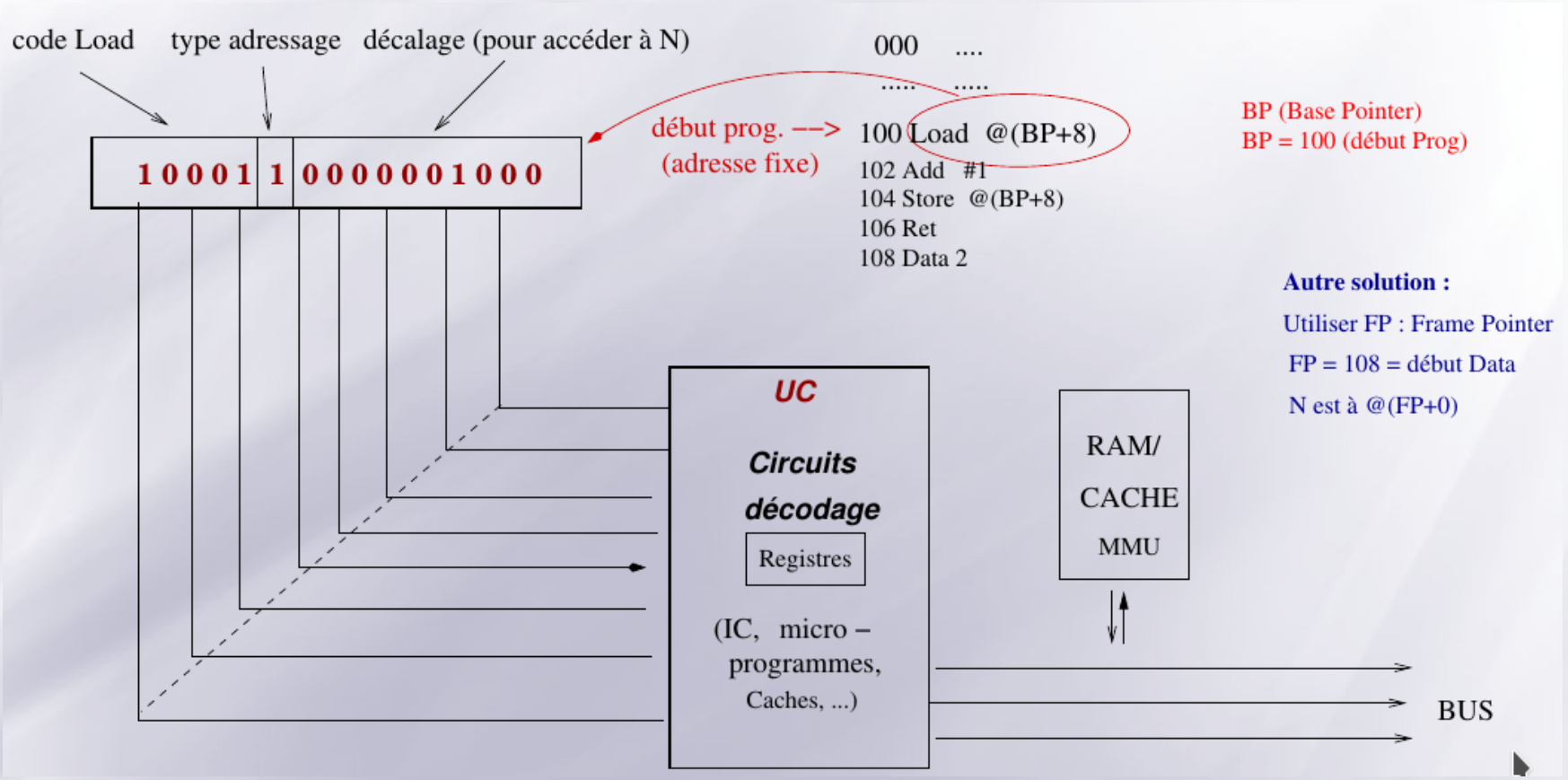
- L'UC demande le Bus, dépose 1 cmd "Read RAM", Dépose @N, ...
- MMU : reçoit demande *Read*, récupère @N, dépose N sur le *Bus Data*
- L'UC récupère (après k cycles) ...

- Ces mécanismes sont à la disposition du SE (ce n'est pas lui qui s'en charge).
- Comment organiser les tâches plus compliquées ?
  - Ne pas faire attendre l'UC (sauf dans les mono tâches).
  - **Signal** et interruption (*Ready*, *Done*, ..) vs. **cycle** et top d'Horloge.
- Notions : Interruptions, Réquisition de Ressources (BUS), Priorité, ...

Comprendre : le décodage d'une instruction (*Load*) chargée dans le *registre d'instruction* de l'UC d'une machine à accumulateur (Z80 !)

Anatomie d'une instruction (suite)

- Détails (on utilise l'accumulateur à la place du registre) :



Solutions

Comment faire ? :

on protège la 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

# Incrémentation avec protection de la variable partagée
def incrementer_la_variable_partagee(nb_iterations):
    """ Chacun incrémente dans la section protégée """

    for i in range(nb_iterations):
        verrou.acquire()              # ← entrée dans la section critique (SC)
        variable_partagee.value += 1
        verrou.release()              # ← sortie de la section critique (SC)
```

Le reste du code :

Solutions (suite)

```
#---------- PARTIE principale (le point d'entrée de cet exemple -------
# On recommence avec la version protégée par un verrou
if __name__ == '__main__' :
    nb_iterations = 5000

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

     verrou=mp.Lock()

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

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

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

☞ Remarquer l'intérêt de "main" :

"verrou"" et "variable_partagee" sont visibles ailleurs !

Solutions (suite)

- Trace : on teste plusieurs 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):
...
```

➛ Le code et *Lock* (verrou équivalent à un sémaphore binaire)

➛ Possible : RLock() (un Lock Ré-entrant = Récursif)

Solutions (suite)

With-Statement-Context Manager : il est possible de simplifier la fonction ci-dessus

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

Noter *With-Statement-Context Manager* :

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

L'intérêt : écrire moins de choses, ne pas risquer d'oublier *release()*

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 (le sémaphore est appelé *privé*).
    - déclaré par **S=multiprocessing.Semaphore(0)**
    - la tâche qui exécute *S.release()* n'est en général pas 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)

La fonction d'incrémentation ne change pas.

```
import multiprocessing as mp

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


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


def count2_SC_sem(nb_iterations):
    """ Chacun écrit à son rythme (Section Critique protégée avec un verrou)"""

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

Solution avec un sémaphore (suite)

```
if __name__ == "__main__" :
    # if __name__ == '__main__' :
    nb_iterations = 5000

    # La variable partagée : placée hors cette fonction (sinon, la passr en param)
    # 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=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 incrémentations %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 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 et son équivalent (qui suit)

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

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

Verrou implicite

Verrou implicite pour les variables partagées

A la création, les variables partagées peuvent demander à être protégées.

- ➛ Il s'agit d'un paramètre de la déclaration de *mp.Value(type, *args, lock=True)*
- ➛ Le *Lock* est donc fourni par défaut

- Exemple : la déclaration du verrou disparaît :

```
import multiprocessing as mp


# Incrémentation avec protection de la variable partagée
def incrementer_la_variable_partagee(nb_iterations):
    """ Chacun incrémente dans la section protégée """

    for i in range(nb_iterations):
        with variable_partagee.get_lock(): # ← IMPORTANT
            variable_partagee.value += 1
```

☞ Notez bien l'utilisation du lock lors de l'incrémentation.

Verrou implicite (suite)

```
#---------- PARTIE principale (le point d'entrée de cet exemple -------
if __name__ == '__main__' :
    nb_iterations = 5000 # chaque processus incrémenter 5000 fois

    # La variable partagée
     variable_partagee = mp.Value('i',lock=True)   # ce sera un entier avec verrou "foruni"

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

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

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

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

Incrémentation : solution avec Array

- Parfois il est possible d'éviter une SC.
- On s'arrange pour avoir chacun "sa case" !
- On utilise le type **Array** (notez le 'A' majuscule).
  - ➛ On s'arrange pour qu'à aucun moment la *race-condition* survienne.
  - ➛ Le verrou disparaît (plus besoin dans ce cas précis).

```
import multiprocessing as mp

# Ici, chaque process incrémente 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
```

Incrémentation : solution avec Array (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 redéfinira  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 incrémentations : ", tableau_partage[:])

    # On crée 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 incrémentations : %d (doit etre %d)"\
          %(sum(tableau_partage),nb_iterations*2) )
```

Incrémentation : solution avec Array (suite)

- La trace :

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

☞ Remarque sur 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()
```

TabMat

TabMat (suite)

12 Passage à "multiprocessing"

13 Solutions à la synchronisation
- Protection de la section critique
- Solution avec un sémaphore
- Verrou implicite
- Incrémentation : solution alternative