# Introduction à la programmation concurrente

## Programmation Concurrente en Python
## Quelques notions sur les activités parallèles

Alexander Saidi
ECL - LIRIS - CPE

Nov 2020

# 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)

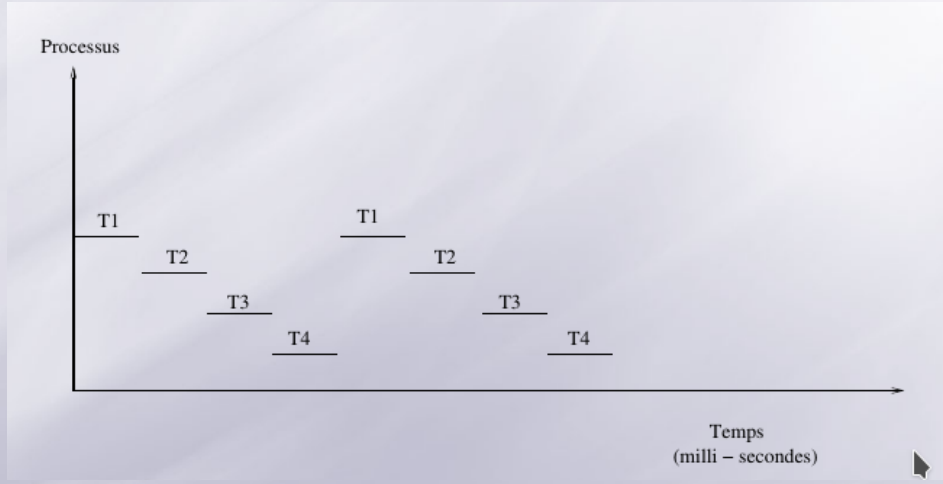
- Liens avec l'embarqué, temps réel, ...
- Formalisation :

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

# Introduction (suite)

## C'était comment (Monoprocesseur / Mono corps) ?

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)

# Quelques exemples introductifs

- Somme des des éléments d'un tableau
- Calcule de la valeur approchée de $\pi$
- Schéma client-serveur simplifié (opérations arith)
- Exemple avec pipe() : "ps -ef | wc -l"

# Un exemple : somme PIle d'un tableau

## Somme des éléments d'un tableau

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*
- Quel est le problème ?
  - ➛ transmission d'un résultats ? "return" ?

# Un exemple : 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 !)

```
# importations déjà faites

if __name__ == "__main__" :
    taille = 11
    tableau = [1 for i in range(taille)]

    somme_totale = mp.Value('i', 0)
    id_fils = os.fork()
    if not id_fils : # Je suis le fils
        somme(1, somme_totale, tableau[:taille // 2])
    else : # Le père fais l'autre moitié
        somme(0, somme_totale, tableau[taille // 2:])
        os.wait()
        print("La somme totale du tableau est ", somme_totale.value)
"""
TRACE :
Je suis le fils num  0 et je fais la somme du tableau  [1, 1, 1, 1, 1, 1]
Je suis le fils num  1 et je fais la somme du tableau  [1, 1, 1, 1, 1]
La somme totale du tableau est  11
"""
```

# Comparaison avec la version séquentielle

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

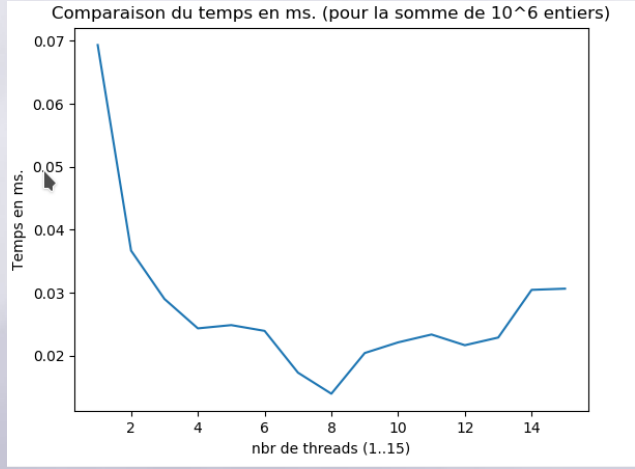



FIGURE 1 – moyennes des temps sur 50 itérations sur un Intel-I7

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

➛ Le gain sera moins fort pour des complexités supérieures.

# Comparaison avec la version séquentielle (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_globale`.
- La fonction *main* attend la fin de chaque fils avec **wait**.

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

  ➛ La Somme totale sera alors erronée.

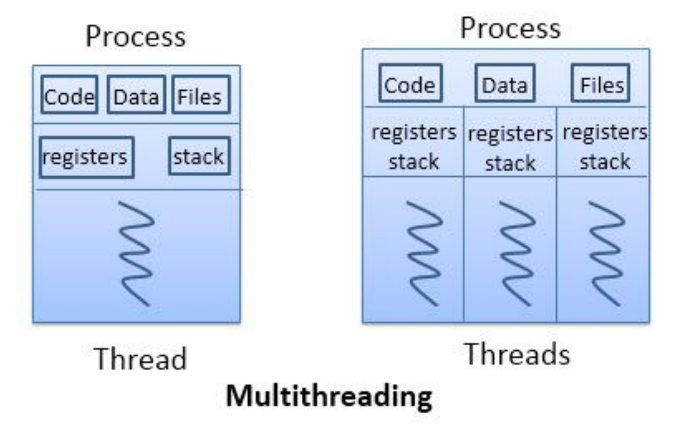
# Comparaison avec la version séquentielle (suite)

- Il y a un risque (faible mais non nul) que les processus se marchent sur les pieds lors la manipulation de `somme_globale`.
  - ➛ Différentes solutions : *exclusion mutuelle*, Pipe, ...

# GIL

A propos des "threads" sous Python

- Cas particulier de Python (GIL : Global Interpreter Lock)
- Threads sont utiles dans "quelques" de cas !




- Les threads sont plus réactifs
- Le partage des ressources est plus simple

# Somme avec array et Pipe

On revisite l'exemple somme :

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

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

# La fonction des fils
def somme(num_process, table, debut, fin_inclue, entree_pere_Write, entree_fils_Read) :
    print("Je suis le fils num ", num_process, "et je fais la somme du tableau ", tableau[debut: fin_inclue] )
    S_local=0
    for i in range(debut, fin_inclue) :
        S_local += tableau[i]

    entree_pere_Write.send(S_local)        # «— Chaque fils transmet son résultat
```

# Somme avec array et Pipe (suite)

- Le **main**

```
if __name__ == "__main__" :
    taille = 1000
    tableau = [i for i in range(taille)]

    entree_pere_Write, entree_fils_Read=mp.Pipe()   # «– W/R des pipes

    id_fils1 = os.fork()
    if not id_fils1 : # Je suis le fils1
        somme(1, tableau, 0, taille // 2, entree_pere_Write, entree_fils_Read)
        os._exit(0)
    else : # Le père
        id_fils2 = os.fork()
        if not id_fils2 : # Je suis le fils2
            somme(2, tableau, taille // 2, taille,entree_pere_Write, entree_fils_Read) # taille//2 car on commence à 0
            os._exit(0)

    # Le fils revient pas ici
    moitie1=entree_fils_Read.recv()     # «– Le père reçoit les résultats
    moitie2=entree_fils_Read.recv()
    #os.wait()   <<–– NO MORE NEEDED ! why ?
    print("La somme totale du tableau est ", moitie1+moitie2)
    print(f"LE pere vérifie que la somme doit être {sum(tableau)}")   # «— Vérification
```

# Somme avec array et Pipe (suite)

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

```
"""
TRACE :
Je suis le fils num  1 et je fais la somme du tableau  [0, 1, 2, 3, 4]
Je suis le fils num  2 et je fais la somme du tableau  [5, 6, 7, 8, 9]
le fils num 1, envoie par send 10
le fils num 2, envoie par send 35
La somme totale du tableau est  45
LE pere vérifie que la somme doit être 45

Je suis le fils num  1 et je fais la somme du tableau .... bcp d'éléments...
le fils num 1, envoie par send 124750
le fils num 2, envoie par send 374750
La somme totale du tableau est  499500
LE pere vérifie que la somme doit être 499500
"""
```

# Exemple 2 : valeur de $\pi$

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

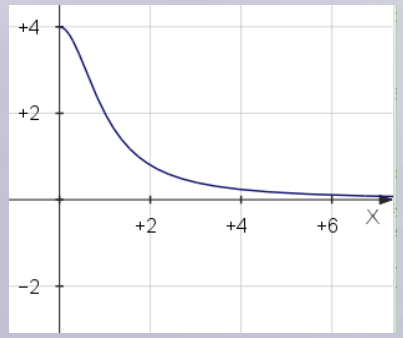
- On peut calculer une valeur approchée de PI par la méthode suivante :

$$\pi \approx \int_0^1 \frac{4}{1+x^2} dx$$

ou la version discrète

$$\pi \approx 1/n \sum_1^n \frac{4}{1+x_i^2}$$

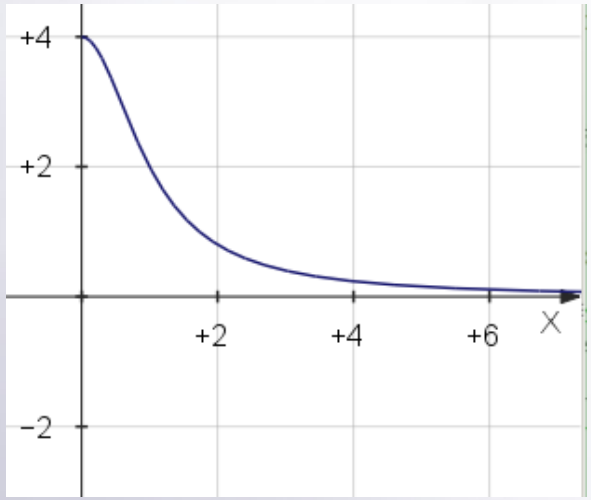
où l'intervalle $[0, 1]$ est divisé en $n$ partitions (bâton) égales.



# Exemple 2 : valeur de $\pi$ (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. $x_i$)

# Exemple 2 : valeur de $\pi$ (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)
```

# Exemple 2 : valeur de $\pi$ (suite)

**Démarche** pur 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)

# Exemple 2 : valeur de $\pi$ (suite)

## Principe :

*Fonction d'un processus (1 travailleur) :*
*_____'*
*La tache de chaque Thread No i=1..k :*
*somme_locale = 0.0*
*verser dans la somme_locale l'aire des battons propres à ce processus*
*Avant de finir, verser la somme_locale dans la Somme_globale*

*Le travail du chef (main) :*
*_____*
*Définir le nombre N de bâtons : soit 10^6*
*Définir le nombre k de processus : soit 5*
*Somme globale = 0.0 (le résultat des calculs)*

*Créer les k processus et assigner à chacun un No (1..k) et sa "part"*
*Les lancer (conséquence de la création) = faire appeler la fonction ci–dessus*
*Attendre qu'ils finissent tous*
*Récupérer et afficher la Somme_globale*

# Exemple 2 : valeur de $\pi$ (suite)

- N.B. : à propos du découpage en intervalles (ici entrelacé)
  - Pour que la collaboration ait un sens :
  - ➛ Au lieu de demander à chacun de calculer $\pi$ (toute l'aire)
  - ➛ On lui demande de calculer $\frac{\pi}{k}$

**Le code d'un processus :**

```
def calculer_une_part_de_PI_arc_tangente(my_num, nb_iter, nb_processus, integrale):
    ma_part_de_pi = 0.0
    for i in range(0,nb_iter,nb_processus):
        ma_part_de_pi += 4/(1+ ((i+0.5)/nb_iter)**2)

     # Je verse ma part dans la variable partagée
    integrale.value += (1/nb_iter)*ma_part_de_pi
```

# Exemple 2 : valeur de $\pi$ (suite)

## Le code du "chef" :

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

if __name__ == "__main__" :
    nb_processus=8              # Nombre travailleurs
    nb_total_iteration = 1000000  # Nombre total d'essai pour l'estimation
    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]=os.fork()
        if  tab_pid[i] == 0 :
            calculer_une_part_de_PI_arc_tangente(i+1, nb_total_iteration // nb_processus,nb_processus,integrale)
            os._exit(0)
        else : pass # le père


    for i in range(nb_processus) : _, _ = os.waitpid(tab_pid[i], 0) # OU os.wait()

    print("Valeur estimée Pi par la méthode Tangente : ", integrale.value)
    print("Temps d'execution : ", time.time() – start_time)
```

# Exemple 2 : valeur de $\pi$ (suite)

- Trace avec différents nombres de processus

```
"""
Valeur estimée Pi par la méthode Tangente :  3.141648769378662
...
1 processus : Temps d'execution :  0.7277581691741943
2 processus : Temps d'execution :  0.19691705703735352
4 processus : Temps d'execution :  0.09310531616210938
8 processus : Temps d'execution :  0.03893399238586426
"""
```

☞ Sur un Intel I7, les calculs sont ici (env.) $> 10$ **fois + rapides** qu'avec un seul processus.

# Exemple 2 : valeur de $\pi$ (suite)

- Question du nombre de processus disponibles dans votre PC :

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

# Exemple 2 : valeur de $\pi$ (suite)

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

- ➛ Ressource unique + accès concurrent (en W/R) : PROTÉGEZ !
- ➛ Parallèle 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
```

# Echange avec pipe

L'exemple suivant montre l'utilisation de fork(), pipe() anonymes, ....

- Le travail du fils :
  - Il attend un message du père contenant une opération (p. ex. 2+3);
  - Il réalise l'opération et transmet le résultat au père via un pipe.

```
import time,os,random
def fils_calculette(rpipe_commande, wpipe_reponse):
    print('Bonjour du Fils', os.getpid())

    while True:
        cmd = os.read(rpipe_commande, 32)
        print("Le fils a recu ", cmd)
        res=eval(cmd)
        print("Dans fils, le résultat =", res)
        os.write(wpipe_reponse, str(res).encode())
        print("Le fils a envoyé", res)
        time.sleep(1)

    os._exit(0)
```

# Echange avec pipe (suite)

- Le père :
  - Prépare une opération arithmétique (p. ex. 2+3); la transmet au fils
  - Récupère le résultat sur un *pipe*.

```
def parent():
    rpipe_reponse, wpipe_reponse = os.pipe()
    rpipe_commande, wpipe_commande = os.pipe()

    pid = os.fork()

    if pid == 0:
        fils_calculette(rpipe_commande, wpipe_reponse)
        assert False, 'fork du fils n a pas marché !'    # Si échec, on affiche un message

    else :
        # On ferme les "portes" non utilisées
        os.close(wpipe_reponse)
        os.close(rpipe_commande)
```

../..

# Echange avec pipe (suite)

```
# ../..


        while True :
            # Le pere envoie au fils un calcul aléatoire à faire et récupère le résultat
            opd1 = random.randint(1,10)
            opd2 = random.randint(1,10)
            operateur=random.choice(['+', '-', '*', '/'])
            str_commande = str(opd1) + operateur + str(opd2)

            os.write(wpipe_commande, str_commande.encode())
            print("Le père va demander à faire : ", str_commande)
            res = os.read(rpipe_reponse, 32)
            print("Le Pere a recu ", res)
            print('-' * 60)
            time.sleep(1)

if __name__ == "__main__" :
    parent()
```

# Echange avec pipe (suite)

- Trace :

```
Le père va demander à faire :  5/9
Bonjour du Fils 12851
Le fils a recu  5/9
Dans fils, le résultat = 0.5555555555555556
Le fils a envoyé 0.5555555555555556
Le Pere a recu  0.5555555555555556
--------------------------------------------------------------
Le père va demander à faire :  5/2
Le fils a recu  5/2
Dans fils, le résultat = 2.5
Le fils a envoyé 2.5
Le Pere a recu  2.5
--------------------------------------------------------------
Le père va demander à faire :  8*6
Le fils a recu  8*6
Dans fils, le résultat = 48
Le fils a envoyé 48
Le Pere a recu  48
--------------------------------------------------------------
...
```

# Exemple Pipe nommé + execlp

- On reprend le même exercice
- L'exemple suivant illustre l'utilisation de *execlp()* avec *fork()*
- On n'utilise plus un pipe anonyme (*execlp()* nous en empêche)
  - ➛ Qui fait *execlp()* quitte le code !
  - ➛ On utilisera un pipe nommé.
  - ➛ Autres solutions compliquées possibles
- Comme pour l'exemple précédent :
  - Un processus envoie des calculs sur un pipe (nommé)
  - La calculatrice effectue l'opération et renvoie le résultat.
- Bien noter les ouvertures des pipe (par open())

☞ Syntaxe un peu "pénible" ! On fera mieux après !

# Exemple Pipe nommé + execlp (suite)

- Le code du demandeur des opérations :

```
import os, time, sys,random

fifoname1 = '/tmp/pipefifo1'
fifoname2 = '/tmp/pipefifo2'

def Chef(  ):
    pipe_operation = os.open(fifoname1, os.O_WRONLY)   # ouvrir le fifo comme un fd
    pipe_res = open(fifoname2, 'r')                    # ouvrir fifo comme un objet stdio

    for i in range(10):
        # Le pere envoie au fils un calcul aléatoire à faire et récupère le résultat
        opd1 = random.randint(1,10)
        opd2 = random.randint(1,10)
        operateur=random.choice(['+', '-', '*', '/'])
        str_commande = str(opd1) + operateur + str(opd2)+"\n"     # Il faut RC

        print("Le chef envoie ", str_commande)
        os.write(pipe_operation, str_commande.encode())

        # On lit le résultat
        line = pipe_res.readline()[:-1]          # bloquant jsq'à l'arrivée de data
        print("Le chef a lu ", line)
        time.sleep(1)
```

# Exemple Pipe nommé + execlp (suite)

```
if __name__ == '__main__':
    if not os.path.exists(fifoname1):
        os.mkfifo(fifoname1)                    # creation
    if not os.path.exists(fifoname2):
        os.mkfifo(fifoname2)

    if os.fork() ==0 :
        os.execlp("/usr/bin/python3.8", "python", 'pipe-nomme-fils.py',fifoname1,fifoname2)

    Chef( )
```

# Exemple Pipe nommé + execlp (suite)

- Le code de l'opérateur : **pipe-nomme-fils.py** activé par *execlp()* :

```
import os, time, sys, random

def calculette(  ):
    pipe_operation = open(fifoname1, 'r')              # <<-- ouvrir fifo comme un objet stdio
    pipe_resultat = os.open(fifoname2, os.O_WRONLY)    # <<-- ouvrir le fifo comme un  fd

    for i in range(10):
        line = pipe_operation.readline(  )[:-1]        # bloquant jsq'à l'arrivée de data
        print('calculette %d a recu "%s"' % (os.getpid(), line))
        res=eval(line)
        print("res = ", res)
        os.write(pipe_resultat, (str(res)+"\n").encode())
        print("La calculette a envoyé ", str(res).encode())
        time.sleep(1)

if __name__ == '__main__':
    fifoname1 = sys.argv[1]
    fifoname2 = sys.argv[2]
    if not os.path.exists(fifoname1):
        os.mkfifo(fifoname1)                  # creation
    if not os.path.exists(fifoname2):
        os.mkfifo(fifoname2)
    calculette()
```

# Exemple Pipe nommé + execlp (suite)

- Trace

```
Le chef envoie  3*5

calculette 24735 a recu "3*5"
res =  15
La calculette a envoyé  b'15'
Le chef a lu  15
Le chef envoie  2*1

calculette 24735 a recu "2*1"
res =  2
La calculette a envoyé  b'2'
Le chef a lu  2
Le chef envoie  7-8

calculette 24735 a recu "7-8"
res =  -1
La calculette a envoyé  b'-1'
Le chef a lu  -1
Le chef envoie  7-7
....
```

# Exemple pipe avec Linux

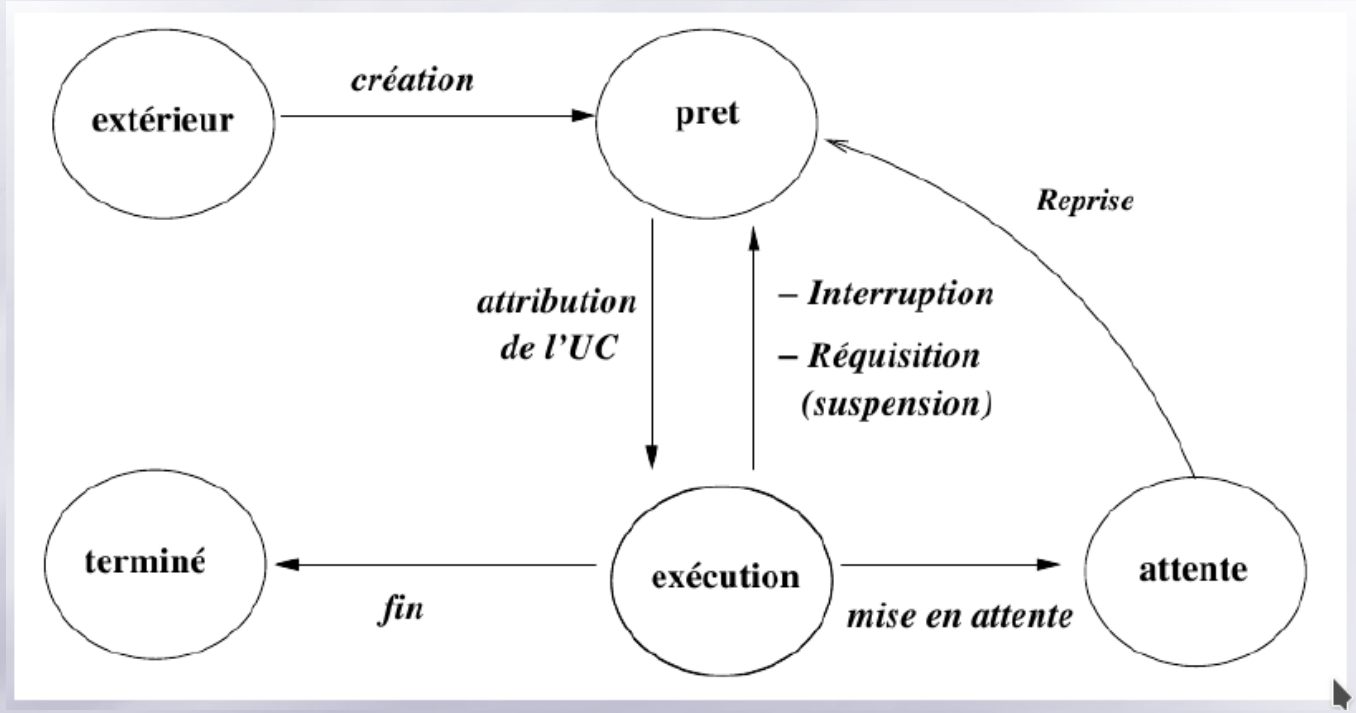
- L'exemple suivant réalise la commande Linux "ps -ef | wc -l"

☞ A exécuter dans un terminal.

```
import os
if __name__ == "__main__" :
    STDIN, STDOUT = 0,1
    pipein, pipeout = os.pipe()

    if os.fork(): # Le père
        # parent
        os.close(pipeout)
        os.dup2(pipein, STDIN)
        os.execlp('wc', '–l')

    else: # Le fils
        os.close(pipein)
        os.dup2(pipeout, STDOUT)

        # Ex : pour éxécuter "gcc –c fic.c", écrire : os.execvp('gcc', 'gcc –c fic.c'.split())

        os.execvp('ps', ['ps'])
# TRACE :     371    4690   49750
```

# Etats d'un processus

- Simplifié



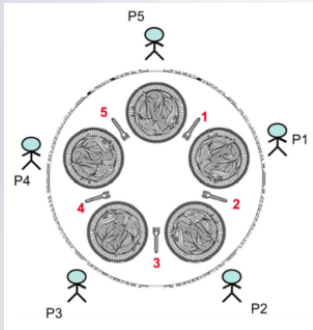
# Accès concurrent à une ressource

## (I) Déménager sans synchro : pas possible !


## (II) Un exemple classique : les **5-**$\Phi$

pensent / mangent / pensent / mangent / ...

- Pour manger, un philosophe a besoin de 2 fourchettes placées sur les deux côtés de son assiette.
  - ➛ P. ex., $\phi_2$ aura besoin de $f_2$ et de $f_3$ pour manger.
- Une mauvaise gestion des fourchettes conduira à affamer un ou plusieurs philosophes,




## (III) Places de parking, Mme. PiPi !

# 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) :

```
import multiprocessing as mp
import os

# Incrémentation sans protéger la variable partagée (non protégée)
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)
```

# Accès concurrent à une ressource (suite)

- Le main :

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

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

    # On crée 2 process
    id1=os.fork()
    if id1 == 0 : # le fils No 1
        count1_on_se_marche_sur_les_pieds(nb_iterations)
    else : # Le père
        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 incrémentations %d (attendu %d) "% (variable_partagee.value,
    nb_iterations*2))
```

# Accès concurrent à une ressource (suite)

- Une trace (plusieurs exécutions) :

```
"""
QQ traces

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

- 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)
```

On obtient :

# Explications (suite)

```
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 nous ne voulons 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**.

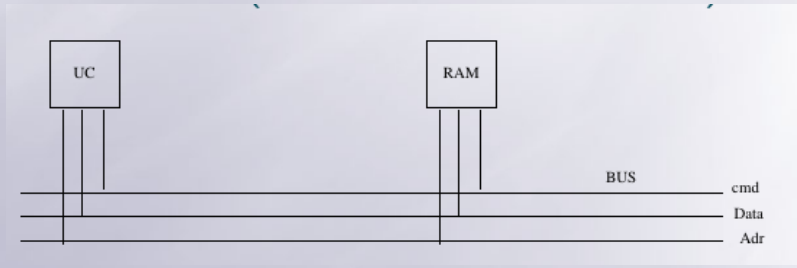
# Anatomie d'une instruction

- 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)

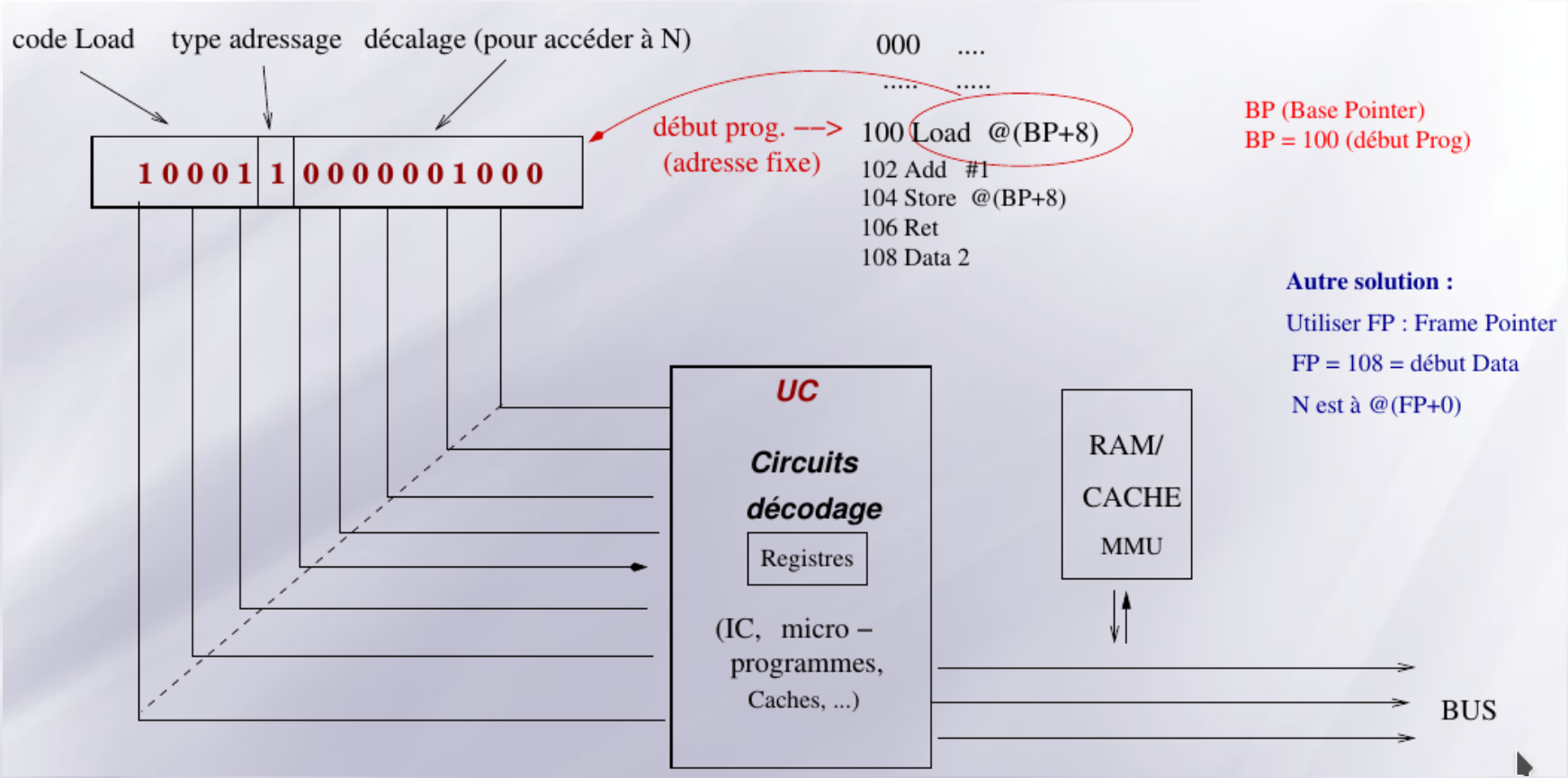
**Exemple : cadencement de *'Load N'*** :

- 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)



# Passage à "multiprocessing"

- Au fait : à travers les exemples suivants ....

# 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 count2_on_protege_la_section_critique(nb_iterations):
    """ Chacun incrémente dans la section protégée """

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

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=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 incrémentations %d (attendu %d): " %
(variable_partagee.value,nb_iterations*2))
```

- Trace : on teste plusieurs fois !

# Solutions (suite)

```
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)

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

```
def count2_on_protege_la_section_critique(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()
```

# Solutions (suite)

Noter *With-Statement-Context Manager* :

gestionnaire des expressions **with xxx**)

```
def count2_on_protege_la_section_critique(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()*

→ danger de **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(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)

☞ On en profite pour passer à multiprocessing !

```
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 (non protégée)"""
    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.Smephore
      variable_partagee.value += 1
```

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

# Solution alternative

- 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).
  - ➛ Le verrou disparaît (plus besoin dans ce cas précis).
  - ➛ A aucun moment la *race-condition* survient.

```
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
```

# 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 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) )
```

# Solution alternative (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()
```