DIU EIL BLOC3

Architectures matérielles et robotique, systèmes et réseaux

Processus légers (Threads)

Vania Marangozova-Martin

Maître de Conférences, UGA

Vania.Marangozova-Martin@imag.fr

Pourquoi allons nous parler de processus légers (threads)?

- **Utilisation massive**
  de la programmation par threads

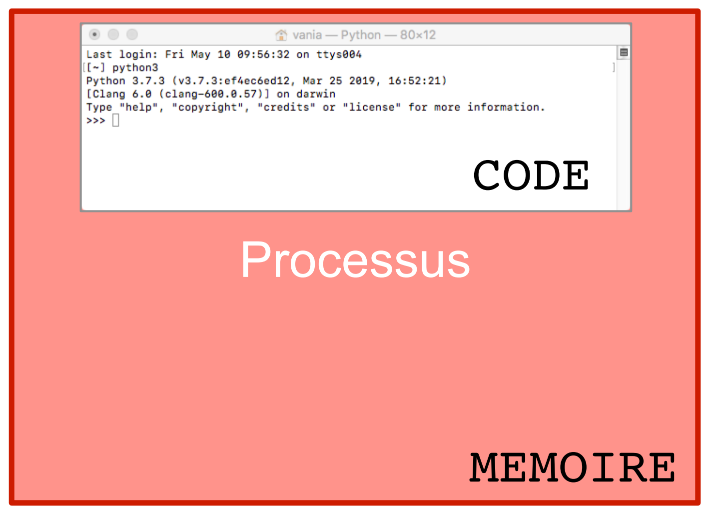
- **Optimisation**
  de l'utilisation des ressources matérielles

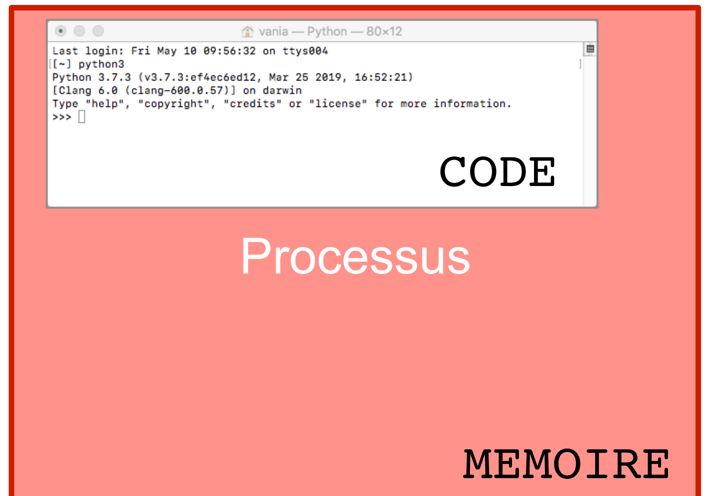
- **Mise en évidence des problèmes de concurrence**
  et illustration de possibles solutions

- **La programmation multi-threadée est difficile**

Les processus = processus **lourds**

- **La gestion des processus est coûteuse en ressources**
  - Chaque processus a besoin d'espace mémoire dédié
  - Dans beaucoup de cas la mémoire est une ressource insuffisante
- **La gestion des processus est coûteuse en temps**
  - Les opérations de création, destruction, etc. de processus prennent du temps
- **Exemple : deux lancements du même programme**





Qu'est-ce un processus **lourd**?

UTILISATEUR

- **Un processus représente l'exécution d'un programme**

SYSTEME

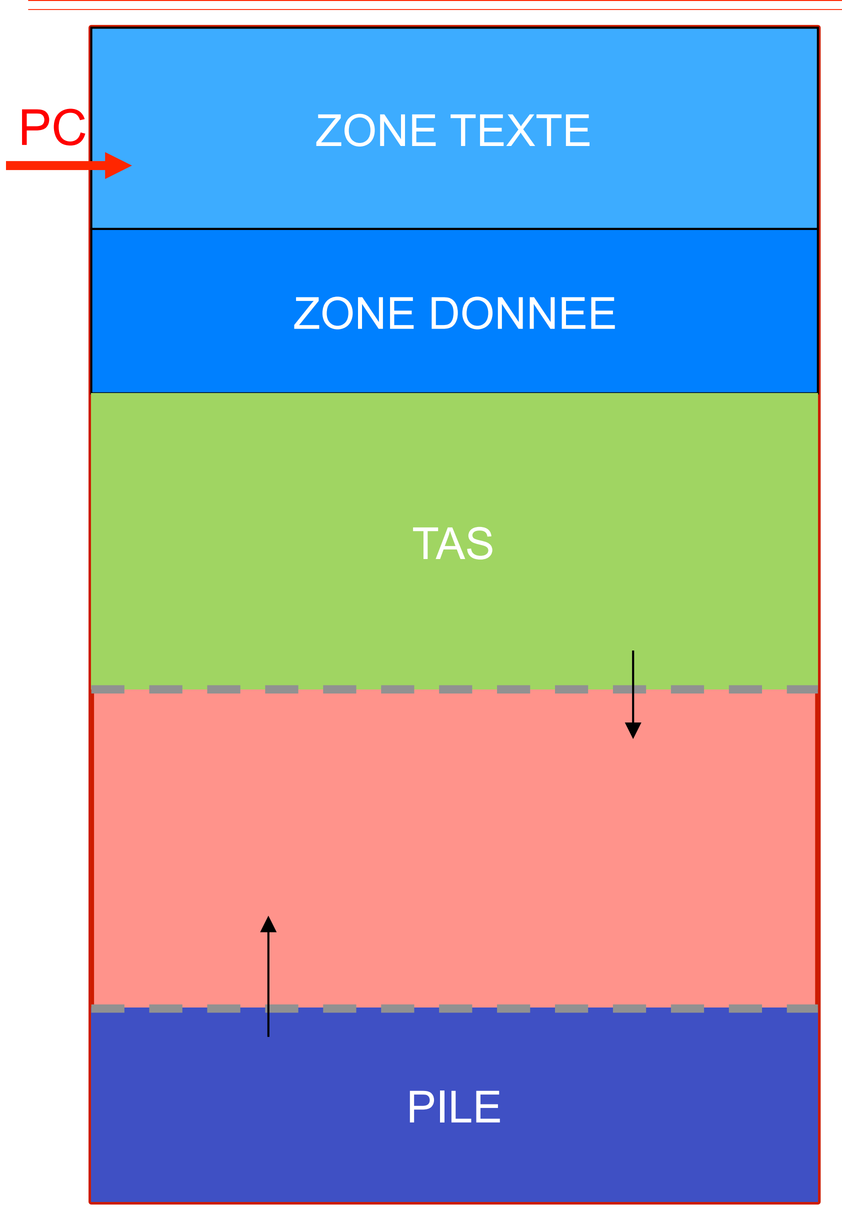
- **Un processus est la somme de**
  - **Un espace mémoire dédié**
    = le processus est chargé en mémoire
  - **Une structure de gestion maintenue par le système**
    où il y a **toutes** les informations le concernant

noyau

mon processus

Mémoire vive

Qu'est ce qu'il y a dans la mémoire d'un processus lourd?



Le code =
les instructions exécuter

Les données statiques
(connues en avance)

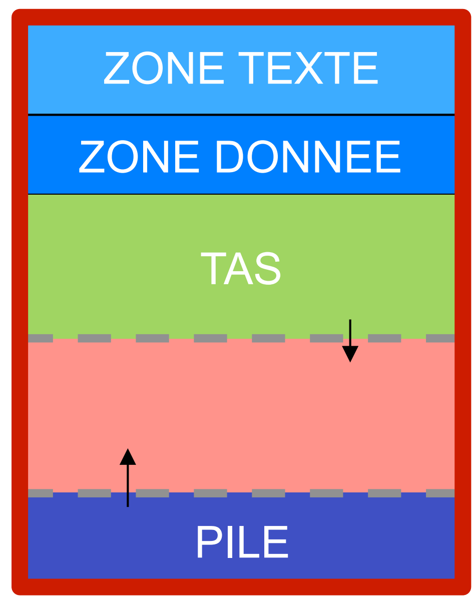
Les données dynamiques
(apparaissent en cours d'exécution)

PC = Program Counter Register
pointe vers la prochaineinstruction

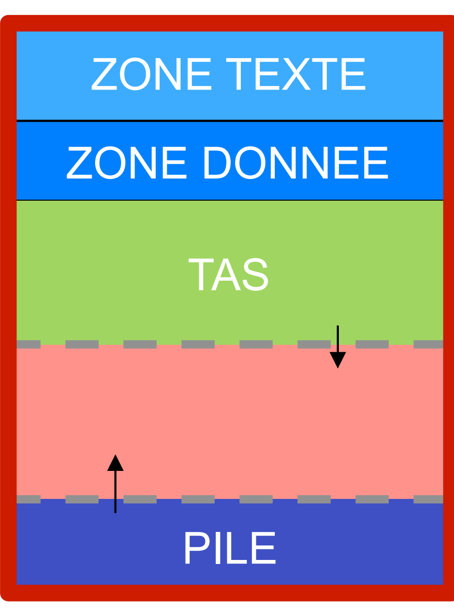
L'historique d'exécution
(dans quelle fonction je suis,
d'où je viens)

Avec les processus **lourds**

- **Il n'est pas possible de factoriser le code**
  - Dans beaucoup d'applications, nous avons besoin de multiples exécutions, en parallèle, du même code (Exemple: serveur web)
- **Il n'est pas possible de travailler sur des données partagées**
  - Exemple : faire la somme des éléments d'une liste en parallèle, en utilisant plusieurs processus



Les processus sont **isolés**



Avec les **threads**

- **Il est possible de factoriser le code**
  - les threads partagent la même copie de code
- **Il est possible de travailler sur des données partagées**
- **Les threads ne sont pas isolés**

PC

ZONE TEXTE
ZONE DONNEE
TAS
PILE

ZONE TEXTE
ZONE DONNEE
TAS

ZONE TEXTE
ZONE DONNEE
TAS
PILE
PILE
PILE

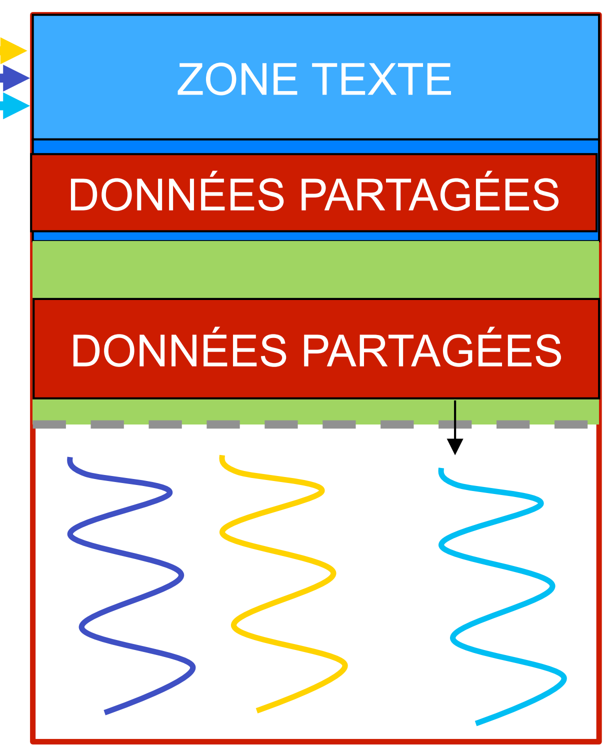
ZONE TEXTE
ZONE DONNEE
TAS

Mono-thread

Multi-thread

Les threads

- **S'exécutent en parallèle**
  - ils ont leurs exécutions propres
    i.e. n'exécutent pas la même instruction en même temps
  - l'ordre d'exécution dépend de l'**ordonnancement** des threads
- **Ont accès à des données partagées**



données non modifiables

données modifiables

Threads en Python

- **Quand on lance un programme Python, il y a un premier thread qui est lancé**
  - habituellement appelé le thread **main**
    - Dans l'exemple,
      le thread main exécute le code
      de la fonction main()

```
#!/usr/bin/env python3
import threading

class myThread (threading.Thread):

    def __init__(self):
        threading.Thread.__init__(self)

    def run(self):
        print("Starting " + self.name)
        print("Exiting " + self.name)


def main():
    print("Starting the main thread!!!")

    # Create a new thread
    thread = myThread()

    # start the thread
    thread.start()

    print("Exiting the main thread!!!")

main()
```

démarrage du thread main

Threads en Python (cont.)

- **Pour programmer des threads**
  - Package en python3 : `import threading`
  - Définition d'une classe par type de threads
    - La classe hérite de `threading.Thread`
    - Une classe = un comportement donné
    - Constructeur `__init__`: initialisation des attributs des threads i.e. des variables manipulées par ce type de threads.
      - Chaque thread a ses propres valeurs
    - Fonction **run** : ce qu'un thread va exécuter une fois lancé (comportement du thread)
    - Fonction **start** : lancement du thread

démarrage d'un thread par le main

démarrage du thread main

```python
#!/usr/bin/env python3
import threading

class myThread (threading.Thread):

    def __init__(self):
        threading.Thread.__init__(self)

    def run(self):
        print("Starting " + self.name)
        print("Exiting " + self.name)


def main():
    print("Starting the main thread!!!")

    # Create a new thread
    thread = myThread()

    # start the thread
    thread.start()

    print("Exiting the main thread!!!")

main()
```

Exécution de l'exemple

```python
#!/usr/bin/env python3
import threading

class myThread (threading.Thread):

    def __init__(self):
        threading.Thread.__init__(self)

    def run(self):
        print("Starting " + self.name)
        print("Exiting " + self.name)


def main():
    print("Starting the main thread!!!")

    # Create a new thread
    thread = myThread()

    # start the thread
    thread.start()

    print("Exiting the main thread!!!")

main()
```

```
[~/Enseignement/DIU_EIL/3_threads/exos] ./0_threads.p
Starting the main thread!!!
Starting Thread-1
Exiting Thread-1
Exiting the main thread!!!
[~/Enseignement/DIU_EIL/3_threads/exos] ./0_threads.p
Starting the main thread!!!
Starting Thread-1
Exiting the main thread!!!
Exiting Thread-1
```

Exemple 2 : un peu plus de threads

```python
#!/usr/bin/env python3
import threading

class myThread (threading.Thread):

    def __init__(self, phrase):
        threading.Thread.__init__(self)
        self.phrase = phrase

    def run(self):
        print("Starting " + self.name)
        for i in range(0,3):
            print(i, self.phrase)
        print("Exiting " + self.name)

def main():
    # Create new threads
    thread1 = myThread("Hello")
    thread2 = myThread("Salut")
    thread3 = myThread("Ciao")
    thread4 = myThread("Marhaba")


    # Start new Threads
    thread1.start()
    thread2.start()
    thread3.start()
    thread4.start()

    print("Exiting the Program!!!")

main()
```

```python
#!/usr/bin/env python3
import threading

class myThread (threading.Thread):

    def __init__(self, phrase):
        threading.Thread.__init__(self)
        self.phrase = phrase

    def run(self):
        print("Starting " + self.name)
        for i in range(0,3):
            print(i, self.phrase)
        print("Exiting " + self.name)

# Create new threads
thread1 = myThread("Hello")
thread2 = myThread("Salut")
thread3 = myThread("Ciao")
thread4 = myThread("Marhaba")


# Start new Threads
thread1.start()
thread2.start()
thread3.start()
thread4.start()

print("Exiting the Program!!!")
```

Exécution et non déterminisme

```
[~/Enseignement/DIU_EIL/3_threads/exos] ./1_th
Starting the main thread!!!
Starting Thread-1
0 Hello
1 Hello
2 Hello
Exiting Thread-1
Starting Thread-2
0 Salut
1 Salut
2 Salut
Exiting Thread-2
Starting Thread-3
0 Ciao
Starting Thread-4
1 Ciao
Exiting the main thread!!!
0 Marhaba
2 Ciao
Exiting Thread-3
1 Marhaba
2 Marhaba
Exiting Thread-4
[~/Enseignement/DIU_EIL/3_threads/exos]
```

2. bash

```
[~/Enseignement/DIU_EIL/3_threads/exos] ./1_threa
Starting the main thread!!!
Starting Thread-1
0 Hello
1 Hello
Starting Thread-2
0 Salut
2 Hello
Starting Thread-3
Exiting Thread-1
1 Salut
0 Ciao
Starting Thread-4
2 Salut
Exiting the main thread!!!
Exiting Thread-2
1 Ciao
0 Marhaba
2 Ciao
1 Marhaba
Exiting Thread-3
2 Marhaba
Exiting Thread-4
[~/Enseignement/DIU_EIL/3_threads/exos]
```

Exécution et non déterminisme

```
[~/Enseignement/DIU_EIL/3_threads/exos] ./1_th
Starting the main thread!!!
Starting Thread-1
0 Hello
1 Hello
2 Hello
Exiting Thread-1
Starting Thread-2
0 Salut
1 Salut
2 Salut
Exiting Thr
Starting Thread-3
0 Ciao
Starting Thread-4
1 Ciao
Exiting the main thread!!!
0 Marhaba
2 Ciao
Exiting Thread-3
1 Marhaba
2 Marhaba
Exiting Thread-4
[~/Enseignement/DIU_EIL/3_threads/exos]
```

2. bash

```
[~/Enseignement/DIU_EIL/3_threads/exos] ./1_threa
Starting the main thread!!!
Starting Thread-1
0 Hello
1 Hello
Starting Thread-2
0 Salut
2 Hello
Starting Thread-3
Starting Thread-4
2 Salut
Exiting the main thread!!!
Exiting Thread-2
1 Ciao
0 Marhaba
2 Ciao
1 Marhaba
Exiting Thread-3
2 Marhaba
Exiting Thread-4
[~/Enseignement/DIU_EIL/3_threads/exos]
```

Quand c'est "petit",
on pourrait avoir l'impression qu'ils commencent et finissent dans l'ordre...

Passer de 3 à 10 itérations

- **Ordre de lancement**
  - main, 1, 2, 4, 3
- **Ordre de terminaison**
  - main, 2, 3, 4, 1

```
[~/Enseignement/DIU_EIL/3_threads/exos] ./1_threads.py
Starting the main thread!!!
Starting Thread-1
0 Hello
Starting Thread-2
1 Hello
0 Salut
Starting Thread-4
Exiting the main thread!!!
1 Salut
0 Marhaba
Starting Thread-3
2 Salut
1 Marhaba
0 Ciao
3 Salut
2 Marhaba
1 Ciao
4 Salut
3 Marhaba
2 Ciao
5 Salut
4 Marhaba
3 Ciao
6 Salut
5 Marhaba
4 Ciao
7 Salut
6 Marhaba
5 Ciao
8 Salut
7 Marhaba
6 Ciao
9 Salut
8 Marhaba
7 Ciao
Exiting Thread-2
9 Marhaba
8 Ciao
2 Hello
9 Ciao
3 Hello
Exiting Thread-3
4 Hello
5 Hello
Exiting Thread-4
6 Hello
7 Hello
8 Hello
9 Hello
Exiting Thread-1
```

Introduire un peu d'ordre...

- **Un thread peut en attendre un autre à l'aide de la fonction `join()`**
  - 1,2,3,4, main

```
8 Salut
9 Ciao
9 Salut
9 Marhaba
Exiting Thread-2
Exiting Thread-3
Exiting Thread-4
Exiting the main thread!!
```

```
Exiting Thread-1
8 Salut
8 Ciao
5 Marhaba
9 Salut
9 Ciao
Exiting Thread-2
6 Marhaba
Exiting Thread-3
7 Marhaba
8 Marhaba
9 Marhaba
Exiting Thread-4
Exiting the main thread!!!
```

```python
#!/usr/bin/env python3
import threading

class myThread (threading.Thread):

    def __init__(self, phrase, toWait):
        threading.Thread.__init__(self)
        self.phrase = phrase
        self.toWait = toWait

    def run(self):
        print("Starting " + self.name)
        for i in range(0,10):
            print(i, self.phrase)
        if self.toWait!=None:
            self.toWait.join()
        print("Exiting " + self.name)

def main():
    print("Starting the main thread!!!")
    # Start new Threads
    thread1.start()
    thread2.start()
    thread3.start()
    thread4.start()

    thread4.join()
    print("Exiting the main thread!!!")

# Create new threads
thread1 = myThread("Hello",None)
thread2 = myThread("Salut", thread1)
thread3 = myThread("Ciao", thread2)
thread4 = myThread("Marhaba", thread3)

main()
```

Concurrence et données partagées : le problème

- **Les threads s'exécutent en parallèle (en même temps*)**
- **Les threads partagent des données**
- **Les données (variables) peuvent donc être lues et modifiées**
- **Ces deux modes d'accès (LECTURE/READ et ÉCRITURE/WRITE) sont conflictuels**
  - Peuvent créer des problèmes de cohérence/integrité de données
  - Exemple connu : lire et écrire un fichier en même temps

*** de point de vue utilisateur.**
Pour savoir comment *exactement*, il faudrait rentrer dans bcp de détails/couches dont on ne s'occupe pas ici.

L'exemple "évident" avec les fichiers

```python
#!/usr/bin/env python3
import threading

# WRITER THREAD
---------------------------------------------
class WriterThread (threading.Thread):

    def __init__(self, nb, phrase):
        threading.Thread.__init__(self)
        self.nb = nb
        self.phrase = phrase

    def run(self):
        file = open("test.txt","w")
        file.write('%3d %15s\n' % (self.nb,
self.phrase))

# READER THREAD
---------------------------------------------
class ReaderThread (threading.Thread):

    def __init__(self, nb):
        threading.Thread.__init__(self)
        self.nb = nb

    def run(self):
        file = open("test.txt","r")
        print("Reader thread...")
        print(file.readline())
```

```python
# main function
---------------------------------------------
def main():
    # Create new threads
    thread1 = WriterThread(1, "Bonjour DIU!" )
    thread2 = WriterThread(2, "Il fait beau" )

    thread3 = ReaderThread(3)

    threads = []
    threads.append(thread1)
    threads.append(thread2)
    threads.append(thread3)

    # Start new Threads
    for t in threads:
        t.start()

    # Wait for the end of the threads
    for t in threads:
        t.join()

    print("Exiting the main thread")


# MAIN -----------------------------------
print("Starting the main thread!!!")
main()
```

CC BY NC SA

L'exécution

- **En fonction de l'ordonnancement, un des deux `WriterThread` passe en 1er**
- **Celui qui passe en 2ème, écrase ce que le 1er a écrit**
  - Parce qu'on n'a pas fait attention que le programme est multi-threadé
  - Faudrait mettre `file = open("test.txt","a")`
- **Dans tous les cas, ne sait pas ce que va lire le `ReaderThread` !**

```
[~/Enseignement/DIU_EIL/3_threads/exos] ./pb_file_threads.py
Starting the main thread!!!
Reader thread...
  1     Bonjour DIU!

Exiting the main thread
[~/Enseignement/DIU_EIL/3_threads/exos] ./pb_file_threads.py
Starting the main thread!!!
Reader thread...
  2     Il fait beau

Exiting the main thread
```

Il y a plus gênant encore!

- **Considérons un programme où nous manipulons une liste de réels (`float`)**
  - 100 éléments, initialisés à 0.0
- **Nous allons créer deux types de threads**
  - **PlusThread**
    - boucle pour rajouter 1.0 à chaque élément de la liste
    - la boucle est exécutée 1000 fois (donc au total +1000)
  - **MinusThread**
    - boucle pour enlever1.0 à chaque élément de la liste
    - la boucle est exécutée 1000 fois (donc au total -1000)
- **Il va y avoir le même nombre de PlusThread et de MinusThread**
  - 500 de chaque type
- **Valeurs attendues à la fin?** `[0.0, 0.0, … 0.0]`

Le programme Python

```python
#!/usr/bin/env python3
import threading

ITERATIONS = 1000

# PlusThread ------------------------------------
class PlusThread (threading.Thread):
    def __init__(self, pair):
        threading.Thread.__init__(self)
        self.pair = pair

    def run(self):
        global lx
        for i in range(0,ITERATIONS):
            for j in range(0,list_size):
                lx[j] += 1.0


# MinusThread -----------------------------------
class MinusThread (threading.Thread):
    def __init__(self, pair):
        threading.Thread.__init__(self)
        self.pair = pair

    def run(self):
        global lx
        for i in range(0,ITERATIONS):
            for j in range(0,list_size):
                lx[j] -= 1.0
```

```python
def main():
    global lx
    NB_THREADS = 1000

    # Create new threads
    threads = []
    for t in range(0,NB_THREADS//2):
        thread = PlusThread(t)
        threads.append(thread)

    for t in range(NB_THREADS//2,NB_THREADS):
        thread = MinusThread(t)
        threads.append(thread)

    # Start new threads
    for t in threads:
        t.start()

    # Wait for threads
    for t in threads:
        t.join()

    print("Exiting the main thread with lx = ", lx)

# MAIN ------------------------------------------
print("Starting the main thread!!!")

#shared list
list_size=100
lx=[]
for i in range(0,list_size):
    lx.append(0.0)
```

L'exécution

```
[~/Enseignement/DIU_EIL/3_threads/exos] ./pb_threads.py
Starting the main thread!!!
Exiting the main thread with lx = [41.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 863.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -69.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 47.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0]
```

```
[~/Enseignement/DIU_EIL/3_threads/exos] ./pb_threads.py
Starting the main thread!!!
Exiting the main thread with lx =  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 42.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -932.0, 0.0, 0.0, 0.0, -80
7.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 962.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0
.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -48.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

```
[~/Enseignement/DIU_EIL/3_threads/exos] ./pb_threads.py
Starting the main thread!!!
Exiting the main thread with lx =
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0
.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -27.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -805.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0]
```

Le problème de **non atomicité** des opérations

- **Le développeur a écrit une instruction Python**

```
lx[j] += 1.0
```

Le problème de **non atomicité** des opérations

- **Le développeur a écrit une instruction Python**

```
lx[j] += 1.0
```

- **En réalité, il y en a plusieurs**
  - Rappelons nous que le CPU exécute des instructions de bas niveau
  - **Les instructions de haut niveau donnent lieu à plusieurs instructions de bas niveau**

```
1 lire la valeur de lx[j] en mémoire
2 calculer +1.0
3 écrire la nouvelle valeur en mémoire
```

Le problème de **non atomicité** des opérations

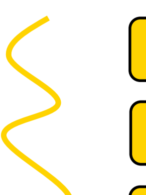
- **Le développeur a écrit une instruction Python**

```
lx[j] += 1.0
```

- **En réalité, il y en a plusieurs**
  - Rappelons nous que le CPU exécute des instructions de bas niveau
  - **Les instructions de haut niveau donnent lieu à plusieurs instructions de bas niveau**

```
1 lire la valeur de lx[j] en mémoire
2 calculer +1.0
3 écrire la nouvelle valeur en mémoire
```

- **Exécution avec deux threads, un jaune et un bleu**



1	lx[j] est 4
1	lx[j] est 4
2	4 + 1 = 5
2	4 + 1 = 5
3	lx[j] devient 5
3	lx[j] devient 5

Le problème de **non atomicité** des opérations

```
1  lire la valeur de lx[j] en mémoire
2  calculer +1.0
3  écrire la nouvelle valeur en mémoire
```

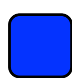
- **Exécution avec 3 threads : jaune, bleu et vert**

Étape	Thread	Opération
1	jaune	lx[j] est 4
1	vert	lx[j] est 4
2	jaune	4 + 1 = 5
3	jaune	lx[j] devient 5
1	bleu	lx[j] est 5
2	bleu	5 + 1 = 6
2	vert	4 + 1 = 5
3	bleu	lx[j] devient 6
3	vert	lx[j] devient 5

Le problème de **non atomicité** des opérations

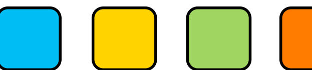
```
1 lire lx[j]
2 calculer +1.0
3 écrire lx[j]
```

PlusThread



```
1 lire lx[j]
2 calculer -1.0
3 écrire lx[j]
```

MinusThread



1	lx[j] est 1
2	1 + 1 = 2
1	lx[j] est 1
3	lx[j] devient 2
1	lx[j] est 2
2	2 - 1 = 1
3	lx[j] devient 1
1	lx[j] est 1
2	1 - 1 = 0
2	1 - 1 = 0
3	lx[j] devient 0
1	lx[j] est 0
3	lx[j] devient 0
2	0 - 1 = -1
3	lx[j] devient -1

Pour garantir la cohérence de la donnée partagée

- **Besoin d'atomicité**
  - Exécuter de manière insecable ("d'un coup")

```
1 lire lx[j]
2 calculer +1.0
3 écrire lx[j]
```

PlusThread

```
1 lire lx[j]
2 calculer -1.0
3 écrire lx[j]
```

MinusThread

- **Besoin d'isolation**
  - Le fait que PlusThread s'exécute ne doit pas perturber l'exécution de MinusThread*
  - Nous n'avons pas d'isolation à ce niveau pour les threads, il faut l'introduire autrement

- **Vers la notion de SECTION CRITIQUE**

Pour résoudre le problème : la synchronisation

- **Pour imposer un ordre d'exécution entre les threads, il est nécessaire de les synchroniser i.e. les faire s'attendre à certains endroits sensibles du programme**

- **Il existe plusieurs mécanismes de synchronisation**
  - verrous
  - sémaphores
  - conditions
  - moniteurs
  - ...

- **Regardons le plus simple : le verrou**

Le verrou (LOCK)

- **Une structure qui peut avoir deux états**
  - vérrouillé
  - déverrouillé
- **Les opérations**
  - `verrouiller` (= prendre le verrou)
  - `déverrouiller` (= relâcher le verrou)
- **Règles d'utilisation**
  - On ne déverrouille que si on a verrouillé au préalable
  - Si on verrouille, au bout d'un moment on déverrouille
- **Fonctionnement**
  - `verrouiller` : Si le verrou est libre, on le prend. Sinon, on attend.
  - `déverrouiller` : Si qqn attend, il ressaye automatiquement de `verrouiller`

Pour illustrer

c'est libre → c'est libre → verrouiller

WC WC

Pour illustrer

ce n'est pas libre

ce n'est pas libre

ce n'est pas libre

Pour illustrer

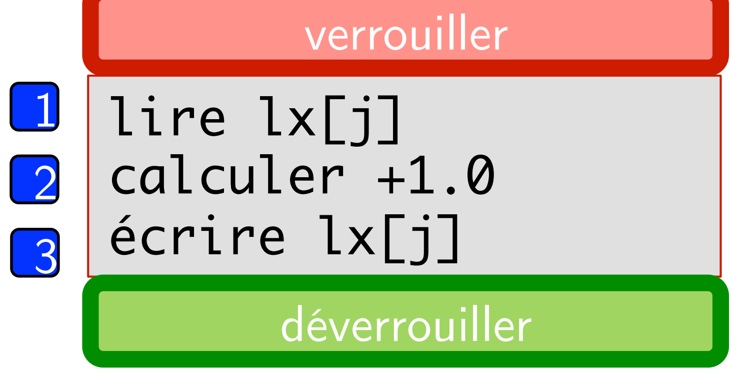
ce n'est pas libre

déverrouiller

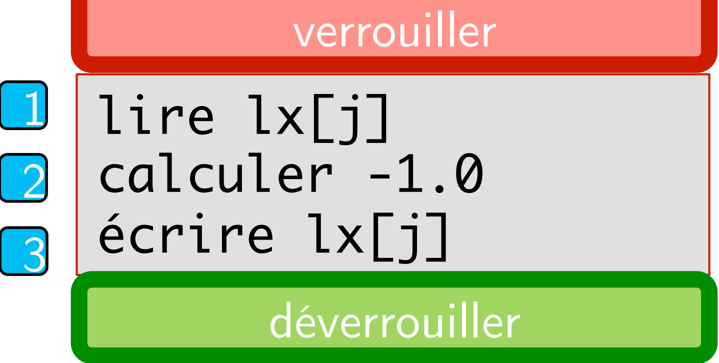
W C

verrouiller

Pour l'exemple précédent



PlusThread



MinusThread

En Python

```python
#!/usr/bin/env python3
import threading

ITERATIONS = 1000

# PlusThread ------------------------------------
class PlusThread (threading.Thread):
    def __init__(self, pair):
        threading.Thread.__init__(self)
        self.pair = pair

    def run(self):
        global lx
        global verrou
        for i in range(0,ITERATIONS):
            for j in range(0,list_size):
                verrou.acquire()
                lx[j] += 1.0
                verrou.release()

# MinusThread -----------------------------------
class MinusThread (threading.Thread):
    def __init__(self, pair):
        threading.Thread.__init__(self)
        self.pair = pair

    def run(self):
        global lx
        global verrou
        for i in range(0,ITERATIONS):
            for j in range(0,list_size):
                verrou.acquire()
                lx[j] -= 1.0
                verrou.release()
```

```python
# main function ----------------------------------
def main():
    global lx
    NB_THREADS = 1000

    # Create new threads
    threads = []
    for t in range(0,NB_THREADS//2):
        thread = PlusThread(t)
        threads.append(thread)

    for t in range(NB_THREADS//2,NB_THREADS):
        thread = MinusThread(t)
        threads.append(thread)

    # Start new threads
    for t in threads:
        t.start()

    # Wait for threads
    for t in threads:
        t.join()

    print("Exiting the main thread with lx = \n", lx)

# MAIN ----------------------------------------
print("Starting the main thread!!!")

#shared list
list_size=100
lx=[]
for i in range(0,list_size):
    lx.append(0.0)

#shared lock
verrou = threading.Lock()
```

Exécution

```
[~/Enseignement/DIU_EIL/3_threads/exos] ./lock_threads.py
Starting the main thread!!!
Exiting the main thread with lx =
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0
.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

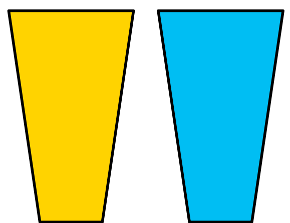
- **Un (tout petit) détour sur les performances**

```
[~/Enseignement/DIU_EIL/3_threads/exos] time ./lock_threads.py
Starting the main thread!!!
Exiting the main thread with lx =
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

real    0m0.560s
user    0m0.465s
sys     0m0.062s
```

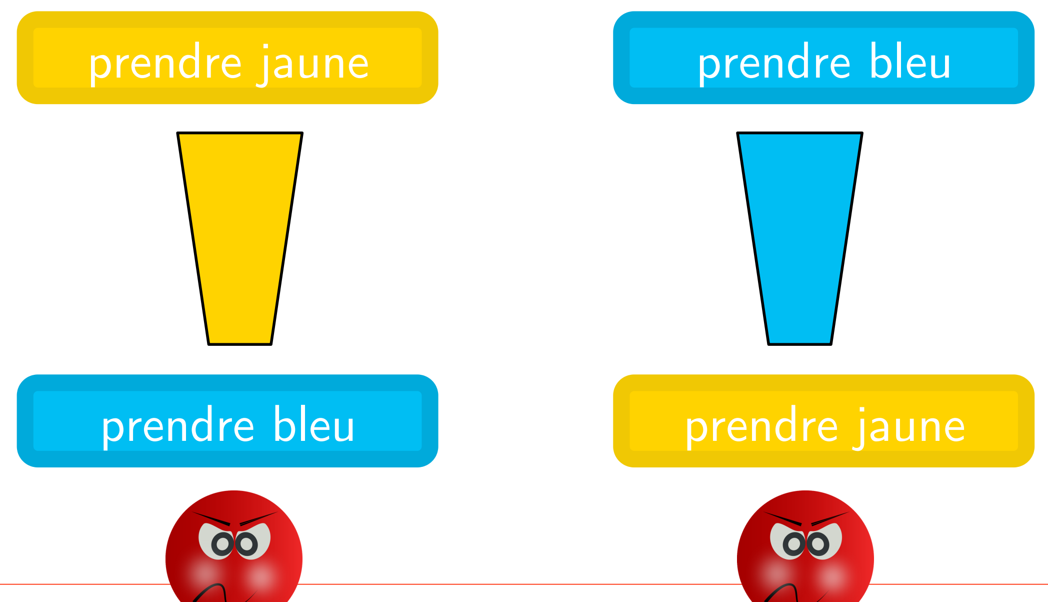
```
[~/Enseignement/DIU_EIL/3_threads/exos] time ./pb_threads.py
Starting the main thread!!!
Exiting the main thread with lx =
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

real    0m0.168s
user    0m0.149s
sys     0m0.012s
```

On n'a pas fini avec les problèmes...

- **Que se passe-t-il dans la situation suivante?**
- **Situation**
  - Deux pots de peinture : verte et jaune
  - Les mêmes deux pots pour tous les threads
  - Les threads veulent faire du vert. Pour cela Ils doivent avoir les deux pots
  - Un pot ne doit être manipulé par plus d'un thread à la fois



- **Exécution**



INTERBLOCAGE

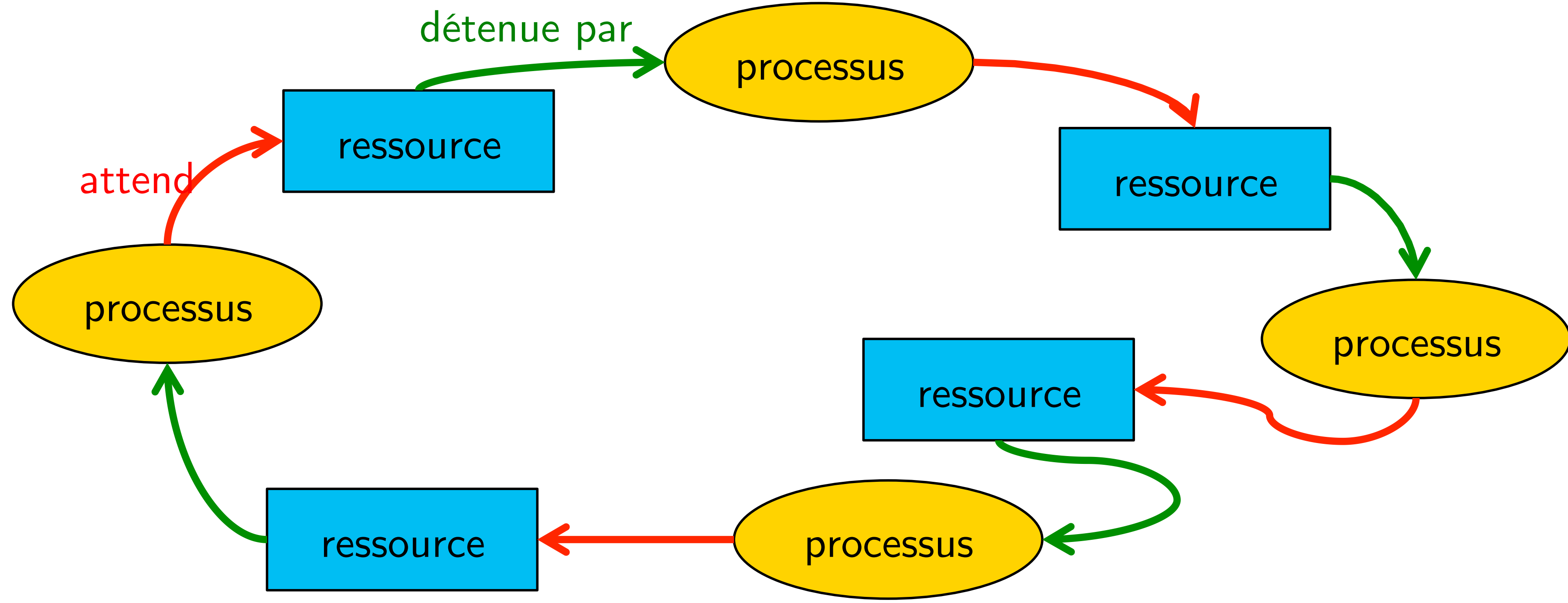
Exécution

```
class JauneDabordThread (threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)

    def run(self):
        global jaune, bleu
        print("JauneDabord : je suis parti!")
        jaune.acquire()
        print("JauneDabord : JAUNE OK!")
        time.sleep(5)
        bleu.acquire()
        print("JauneDabord : BLEU OK!")
        print("JauneDabord : je fabrique du vert!!!")
        time.sleep()
        bleu.release()
        jaune.release()

class BleuDabordThread (threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)

    def run(self):
        global jaune, bleu
        print("BleuDabord : je suis parti!")
        bleu.acquire()
        print("BleuDabord : BLEU OK!")
        time.sleep(5)
        jaune.acquire()
        print("BleuDabord : JAUNE OK!")
        print("BleuDabord : je fabrique du vert!!!")
        time.sleep()
        bleu.release()
        jaune.release()
```

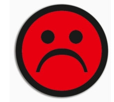
```
[~/Enseignement/DIU_EIL/3_threads/exos] ./deadlock_threads.py
Starting the main thread!!!
JauneDabord : je suis parti!
JauneDabord : JAUNE OK!
BleuDabord : je suis parti!
BleuDabord : BLEU OK!
```

L'interblocage

- **De manière générale, un interblocage peut survenir ssi**
  1. Des processus ou threads s'exécutent en parallèle
  2. obtiennent des ressources partagées
  3. ne relâchent pas les ressources tant qu'ils n'ont pas terminé leur travail
  4. le graphe des attentes forme un cycle



En résumé

- **Les threads sont partout**
  - ils facilitent l'exploitation du parallélisme matériel
  - ils permettent d'effectuer des traitements en parallèles à moindre coût en termes de ressources
  - ils permettent des interactions entre activités parallèles
- **mais**
  - le développement est plus difficile 
  - de nouveaux problèmes (et même bugs) à résoudre
  - ce n'est pas forcément plus performant
    - "en mettant deux threads, cela s'exécutera deux fois plus vite"? non.
      - Coût de synchronisation et de création de threads
      - En général, une partie des traitements d'un programme est séquentielle et ne peut être parallélisée