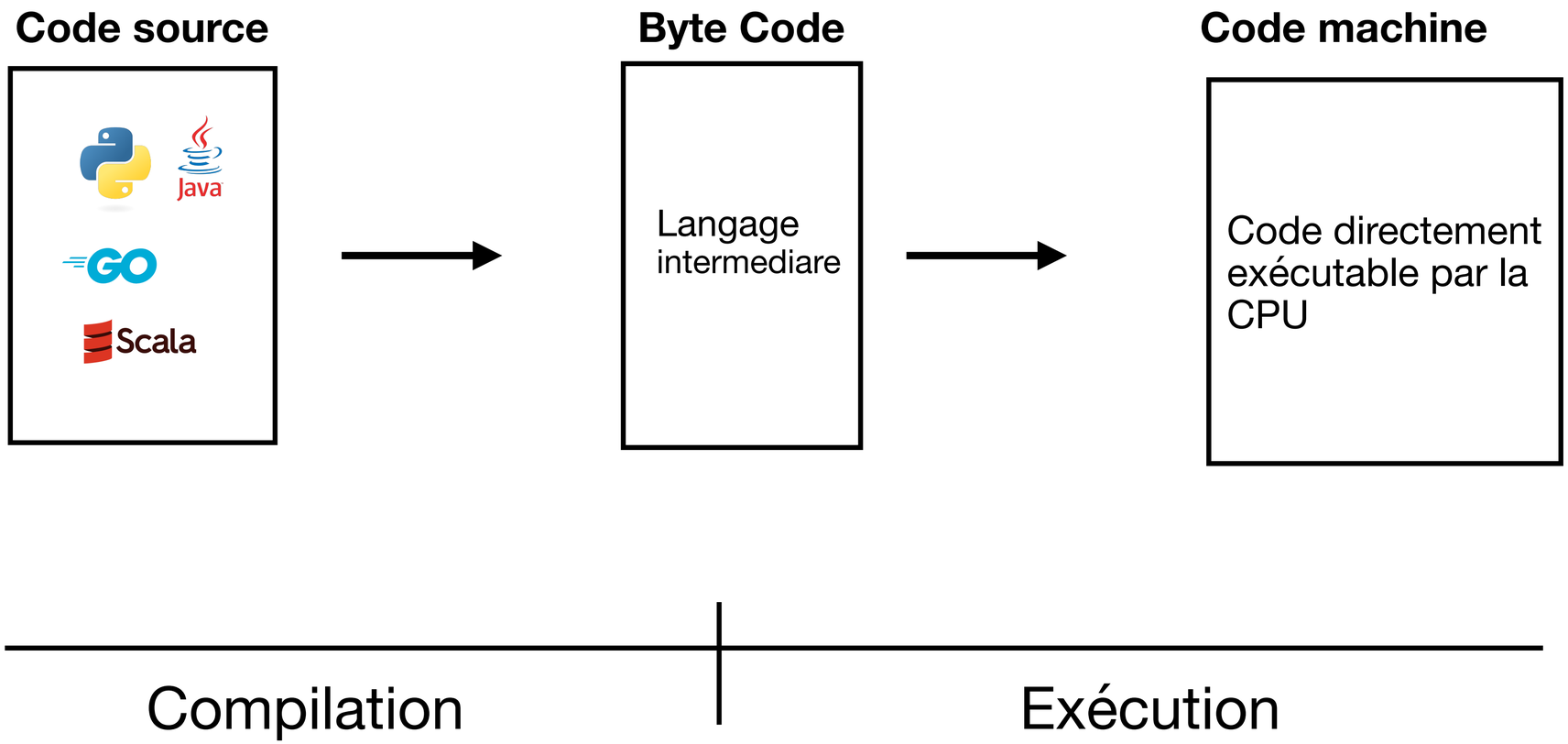
Programmation distribuée

High scalability

Lionel SOUOP

Du code à l’exécution: Le parcours d’un programme

Vue d’ensemble



Pour les langages comme le C, C++, la compilation génère directement le langage machine

Du code à l’exécution: Le parcours d’un programme

Le Code

Le code est:

- Une séquence d’opérations qui implémente un algorithme
- Spécifique à un langage de programmation

```python
from PIL import Image, ImageDraw
import random
import json

def read_box_description(path):  1 usage
    box_desc = dict()
    with open(path, 'r') as file:
        line = file.readline()
        while line:
            elements = line.split(";")
            tmp_box_desc = {
                "type": elements[0],
                "libelle": elements[1],
                "long": elements[2],
                "prof": elements[3],
                "haut": elements[4]
            }
            box_desc[elements[1]] = tmp_box_desc
            print(line.strip())
            line = file.readline()
    return box_desc

def read_shelf_description(path):  1 usage
    with open(path, 'r') as file:
        data = json.load(file)
    return data
```

Du code à l’exécution: Le parcours d’un programme

La compilation

- C & C++ —> Source Code (.c/.cpp) → Preprocessing → Compilation → Assembly → Linking → Executable (.exe/.out)

- Java —> Source Code (.java) → Compilation → Bytecode (.class) → JVM Execution

- Python —> Source Code (.py) → Interpreter Execution → Machine Code

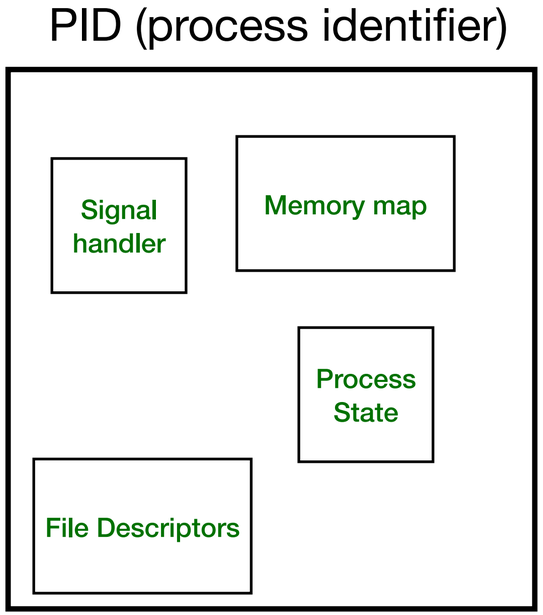
Du code à l’exécution: Le parcours d’un programme

L’exécution

- Chargement de l’exécutable en mémoire. Le système d’exploitation localise et charge l’exécutable dans la RAM
- Création du processus: Un process est crée pour la gestion de l’exécution
- Signal handler - gestion de signal
- File descriptor - int identifiant un fichier ouvert ou une ressource - default= stdin(0), stdout(1), stderr(2)

Un processus est une instance d’une application

En cours d’exécution.



Structure d’un processus

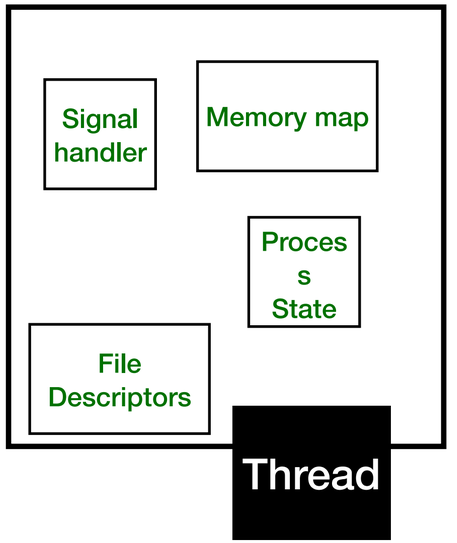
Du code à l’exécution: Le parcours d’un programme

L’exécution

- Allocation de la mémoire: un processus se voit attribuer un espace virtuel de mémoire réparti comme suit:
  - **Code Segment** → stoque le code binaire
  - **Data Segment** → Variables Globales et statiques.
  - **Heap** → Pour l’allocation dynamique de la mémoire
  - **Stack** → appel des fonctions et variables locales

- Création de thread(s)

Un thread est une unité élémentaire de calcul créée
Dans un processus



Structure d’un processus

Du code à l’exécution: Le parcours d’un programme

L’exécution

- Exécution:
    - La CPU exécute les instructions du processus
    - Si le processus interagit avec hardware, il utilise les System Calls disponible dans l’OS
- Fin de l’exécution et libération des ressources

NB: Le Garbage collector libère régulièrement la mémoire

Du code à l’exécution: Le parcours d’un programme

L’exemple d’un convertisseur de minuscule à majuscule de mots

Écrire un programme python qui lit un fichier de plusieurs lignes. Chaque ligne contient une série de mots séparés par une virgule. Le programme transforme chaque mots en majuscule et écrit le résultat dans un autre fichier

Du code à l’exécution: Le parcours d’un programme

Résumé

- Une application classique, après compilation s’exécute en mémoire dans un processus et dans un thread
- Les instructions sont exécutées en séquences et dans l’ordre d’écriture du programme
- Aucun parallélisme par défaut n’est utilisé dans le processus

Programmation distribuée

Définition et principe

Dans un programme classique, l'exécution se fait dans un processus, et ce processus démarre avec un thread qui est crée.

Questions:

- Est-il possible de créer plusieurs threads dans un processus ?
- Est-il possible de créer plusieurs processus ?
- Est-il possible d'exécuter un code en utilisant plusieurs processus ou plusieurs threads ?
- Quels sont les avantages et les inconvénients de l'utilisation d'un tel parallélisme ?

Ces questions définissent le principe de la programmation distribuée

Programmation distribuée

Définition et principe

La programmation distribuée est un paradigme de programmation, qui permet de répartir l'exécution d'un programme sur plusieurs processus ou threads and le but de:

- Améliorer les performances
- Améliorer la scalabilité

Programmation distribuée

Processus: Création

```python
import multiprocessing

if __name__ == "__main__":
    # Array to store de PIDs
    processes = []

    # Create and launch the processes
    number_of_processes = 4
    for i in range(number_of_processes):
        p = multiprocessing.Process()
        processes.append(p)
        p.start()

```

Création de processus

Les processus crées n’ont pas de logique à appliquer

Programmation distribuée

Processus: Création

```python
import multiprocessing

def logic(pid, name):  1 usage
    print(f"I am the process with pid: {pid} and name {name}")

if __name__ == "__main__":
    # Array to store de PIDs
    processes = []

    # Create and launch the processes
    number_of_processes = 4
    for i in range(number_of_processes):
        p = multiprocessing.Process(target=logic, args=(i, i))
        processes.append(p)
        p.start()

```

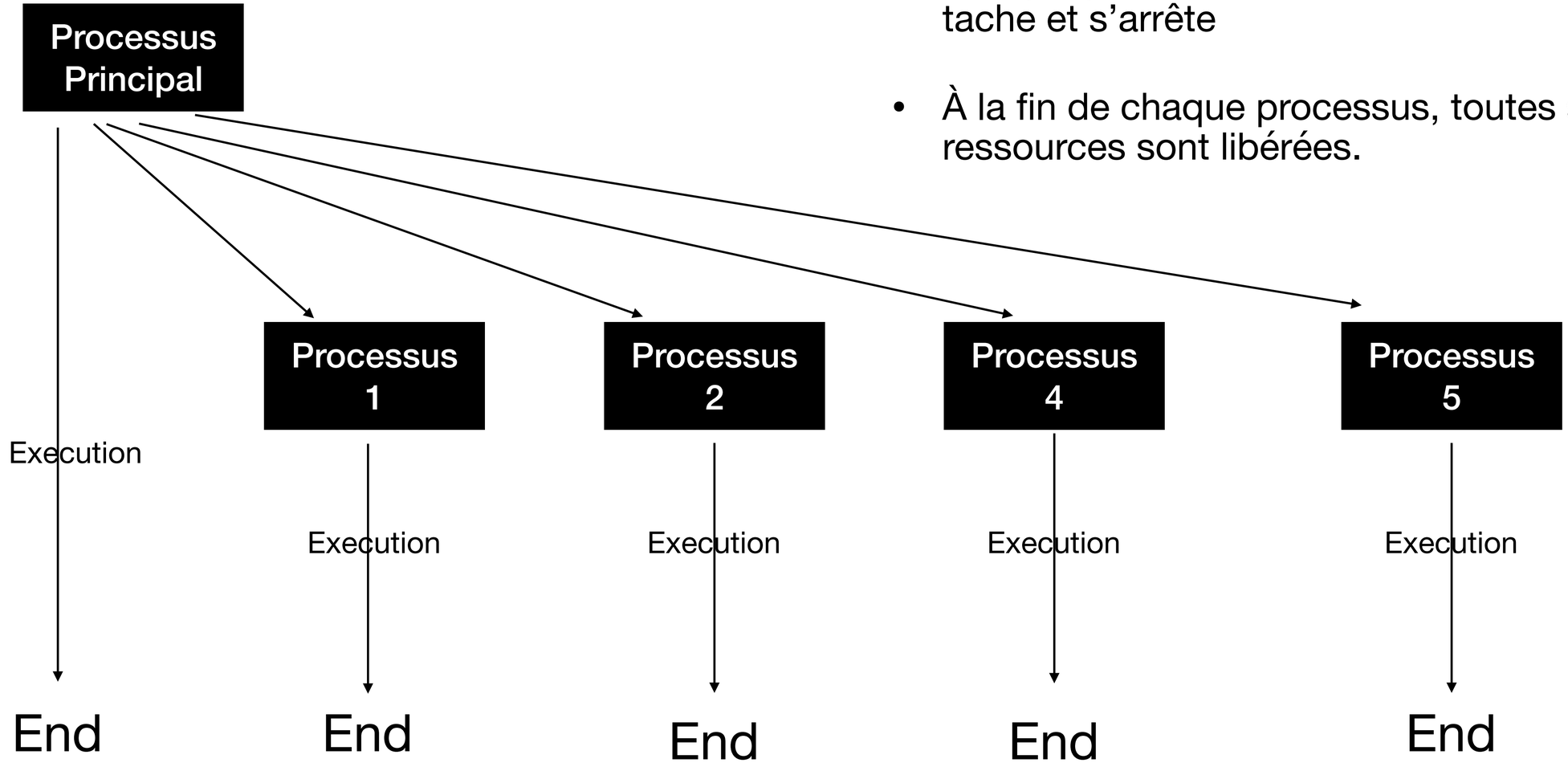
Création de processus

Chaque processus a une logic à exécuter

Programmation distribuée

Processus: Fin d’un processus

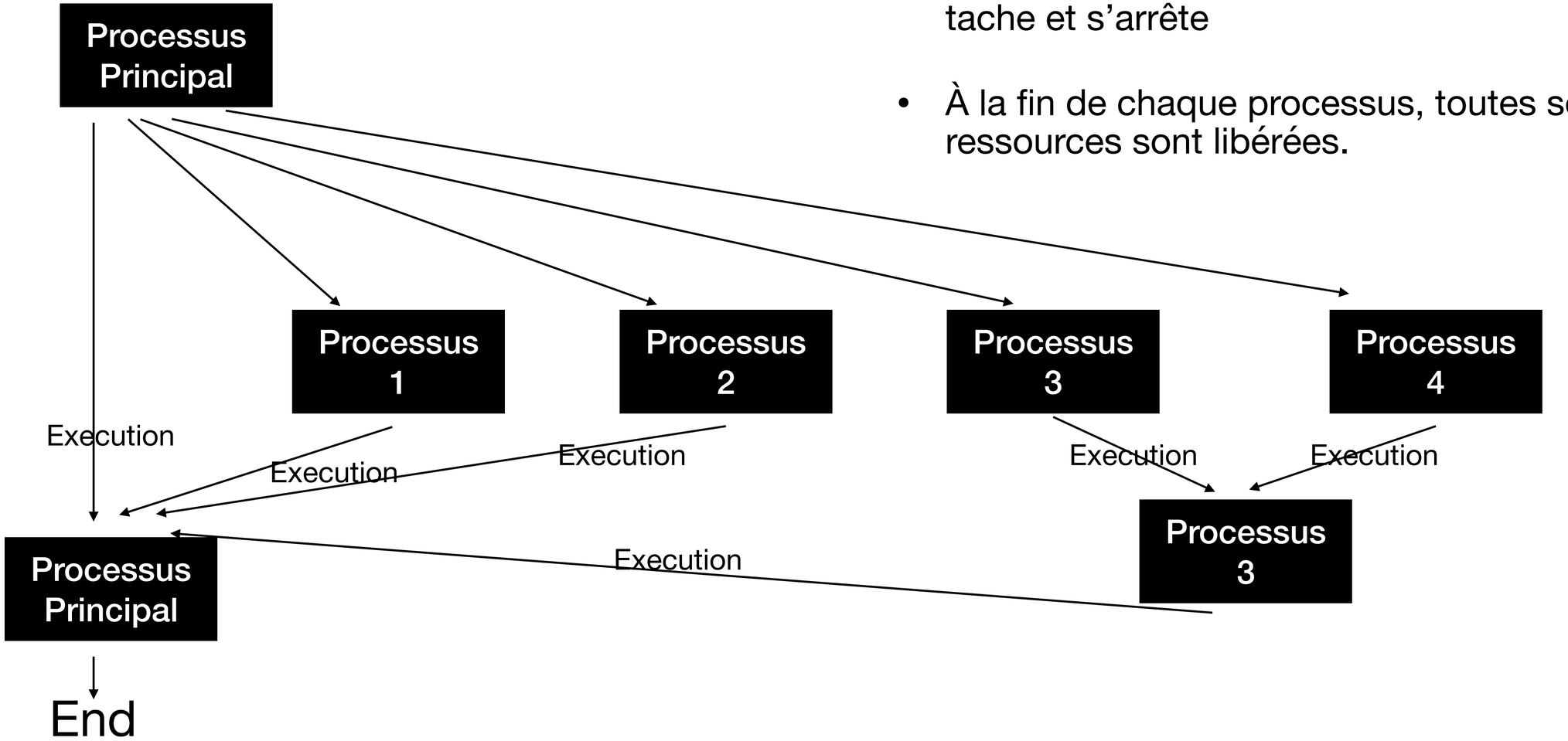
- Chaque processus crée est indépendant
- Chaque processus se lance, exécute sa tache et s’arrête
- À la fin de chaque processus, toutes ses ressources sont libérées.



Programmation distribuée

Processus: Fin d’un processus

- Chaque processus crée est indépendant
- Chaque processus se lance, exécute sa tache et s’arrête
- À la fin de chaque processus, toutes ses ressources sont libérées.



Programmation distribuée

Le processus - fin d’un processus

Un plus:

- Orphan processes - Le processus enfant continue de fonctionner alors que le principal est terminé
- Using daemon=True (Child Dies with Main) - les processus enfant se terminent avec le principal.
- Tuer tous les processus enfant à la fin du principal avec terminate(), atexit() ou avec un signal SIGTERM/SIGINT (kill)

Programmation distribuée

Processus: Synchronisation de processus

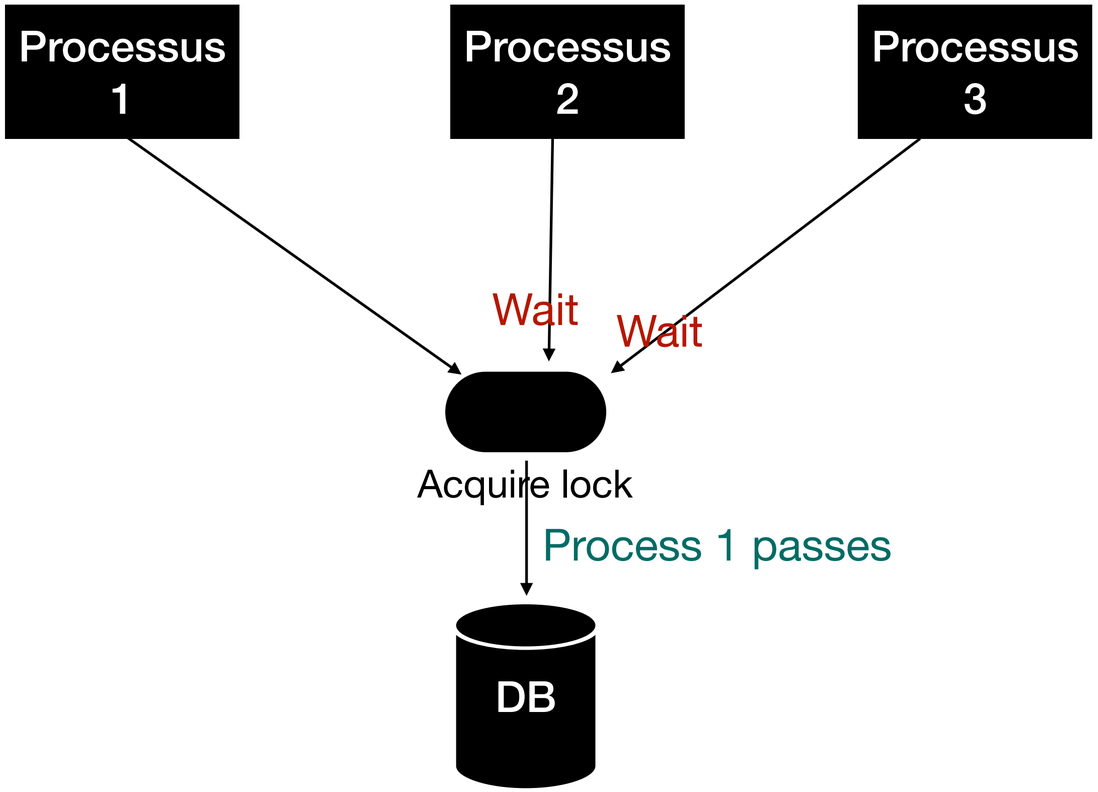
La synchronisation des processus est un mécanisme de contrôle de l’exécution de plusieurs processus:

- Afin d’éviter des conflits d’accès aux resources partagées:
  - **La race condition** -> Résultat imprévisible dû à l'ordre d'exécution des processus.
  - **L’incohérence des données** -> Deux processus écrivent sur un fichier en même temps.
- Pour les besoins de l’algorithme: une action qui doit s’exécuter après une autre.

Attention: Deux processus attendent indéfiniment une ressource que l'autre détient. C’est ce qu’on appelle un **deadlock**

Programmation distribuée

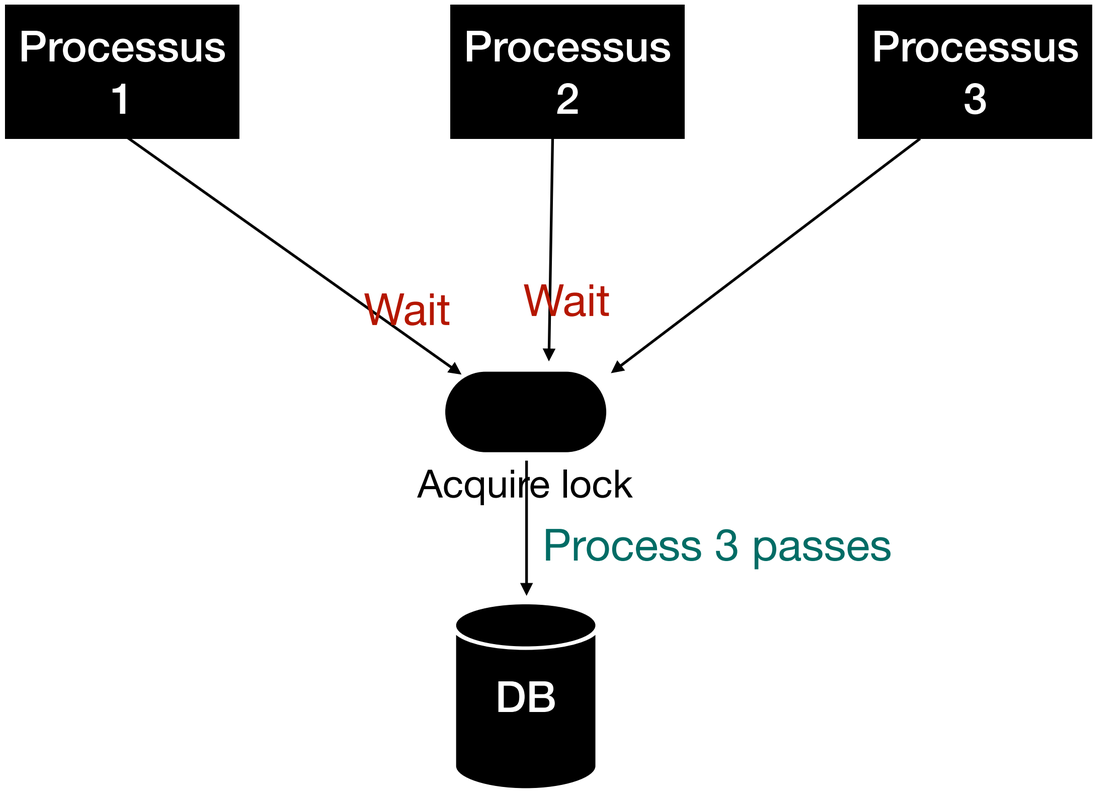
Processus: Synchronisation de processus avec un verrous (lock)



Le premier processus à actionner le verrous (lock) est celui qui peut modifier la ressource. Les autres attendent

Programmation distribuée

Processus: Synchronisation de processus avec un verrous (lock)



Programmation distribuée

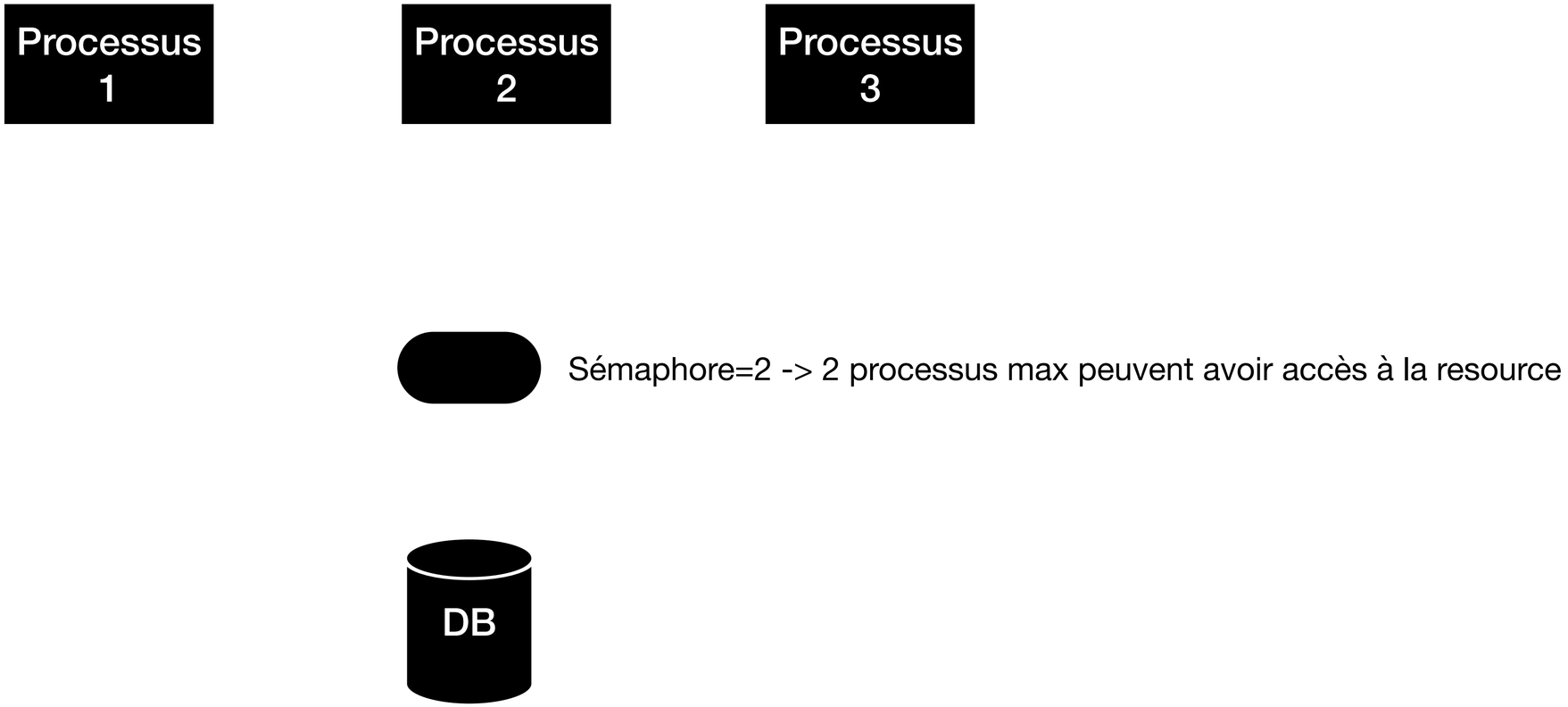
Processus: Synchronisation de processus avec un verrous (lock)

```python
import multiprocessing
import time

def worker(lock, compteur):  1 usage
    with lock:
        # Seul un processus à la fois peut modifier le compteur
        compteur.value += 1
        print(f"Process {multiprocessing.current_process().name} - Compteur : {compteur.value}")
        # Simule un traitement
        time.sleep(0.5)

if __name__ == "__main__":
    # Création d'un verrou
    lock = multiprocessing.Lock()
    # Variable partagée
    compteur = multiprocessing.Value("i", 0)

    processes = [multiprocessing.Process(target=worker, args=(lock, compteur)) for _ in range(5)]

    for p in processes:
        p.start()
    for p in processes:
        p.join()

    print("Final Counter:", compteur.value)
```

Programmation distribuée

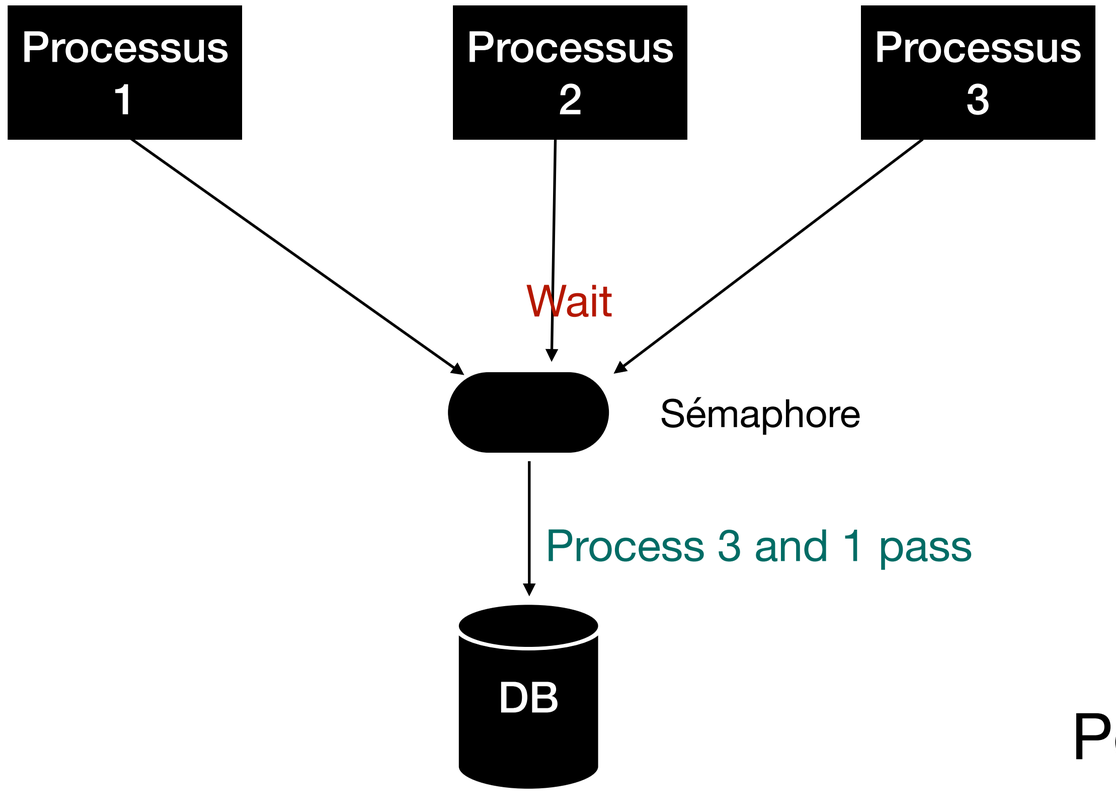
Processus: Synchronisation de processus avec un sémaphore

Un sémaphore limite le nombre de processus accédant à une ressource.



Programmation distribuée

Processus: Synchronisation de processus

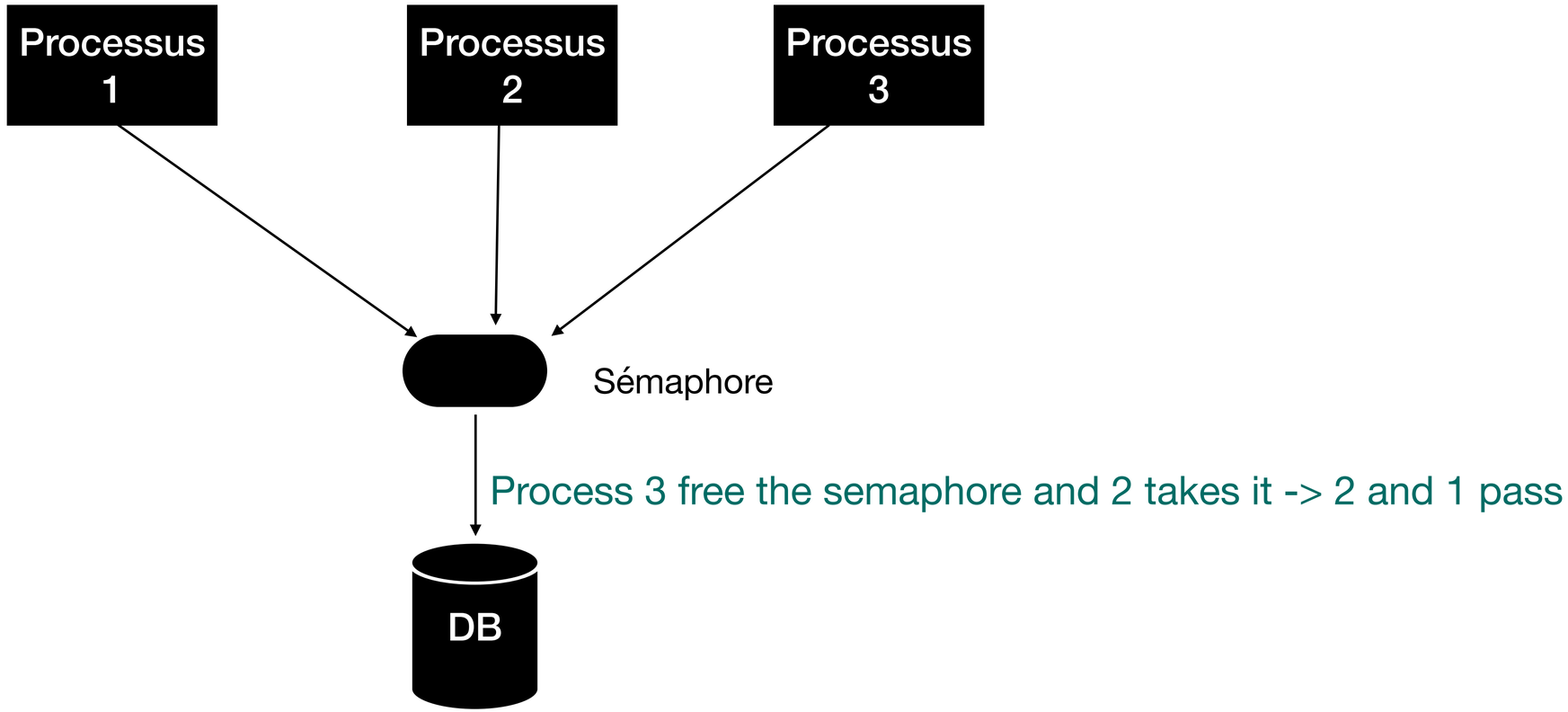


Peut permettre de protéger

une DB de la surcharge

Programmation distribuée

Processus: Synchronisation de processus avec sémaphore



Programmation distribuée

Processus: Synchronisation de processus avec sémaphore

```
 import multiprocessing
import time


def worker(semaphore, num):
    with semaphore:
        print(f"Process {num} accède à la ressource")
        time.sleep(1)  # Simule une utilisation de ressource
        print(f"Process {num} libère la ressource")


if __name__ == "__main__":
    semaphore = multiprocessing.Semaphore(2)  # Max 2 processus à la fois
    processes = [multiprocessing.Process(target=worker, args=(semaphore, i)) for i in range(5)]
    for p in processes:
        p.start()
    for p in processes:
        p.join()
```

Programmation distribuée

Processus: Synchronisation de processus

- Les évènements - Un événement (Event) est un signal permettant aux processus d'attendre ou de continuer. Un évènement bloque un processus jusqu’à ce qu’un autre vienne le débloquer.

- Communication via une Queue - Une file (Queue) permet aux processus d'envoyer et de recevoir des messages de manière synchronisée.

Programmation distribuée

Processus: Synchronisation de processus via Queue

```python
import multiprocessing

def sender(queue):  1 usage
    queue.put("Hello from process!")

def receiver(queue):  1 usage
    message = queue.get()  # Attend un message
    print(f"Reçu : {message}")

if __name__ == "__main__":
    queue = multiprocessing.Queue()
    p1 = multiprocessing.Process(target=sender, args=(queue,))
    p2 = multiprocessing.Process(target=receiver, args=(queue,))

    p1.start()
    p2.start()

    p1.join()
    p2.join()
```

Programmation distribuée

Les threads

- Tous les principes qui s’appliquent au processus, s’appliquent aussi aux threads.
- Threads dans les TP

Programmation distribuée

Le Avantages

- Exécution des applications plus rapide
- Scalabilité - des processus peuvent être exécutés sur des machines distantes

Programmation distribuée

Inconvénients

- Overhead pour la création des threads et processus
- Plus grande utilisation des resources de ou des machines
- Programmation plus complexe car il faut synchroniser les processus ou les threads, il faut partager parfois des informations entre processus et threads, ce qui peut être assez complexe pour un code à grande échelle(Thread safety)

Programmation distribuée

Threads vs processus

🔷 Threads

✔ Avantages :

- Rapide à créer et consomme moins de mémoire.
- Communication facile entre threads (mémoire partagée).
- Idéal pour les tâches I/O (réseau, fichiers, interfaces graphiques).

❌ Inconvénients :

- Risque de race conditions (modification simultanée d'une variable).
- Moins stable : un bug dans un thread peut planter tout le processus.
- Pas de vrai parallélisme en Python (à cause du GIL).
-