Cours 4 : Programmation Asynchrone et Parallèle en Python

Introduction

La programmation parallèle et asynchrone permet d'exécuter plusieurs tâches simultanément pour améliorer les performances des programmes, surtout lors de traitements lourds, d'attentes d'I/O ou d'exécution de plusieurs tâches en parallèle. Nous allons voir deux approches principales : la **programmation parallèle** avec `multiprocessing` et la **programmation asynchrone** avec `asyncio`.

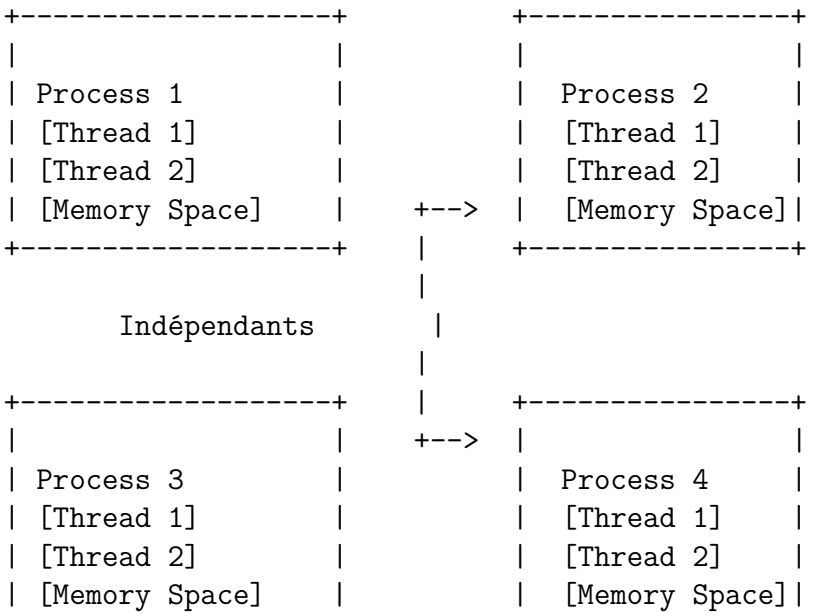
---

1. Processus et Threads : Concepts de Base

a. Processus Un processus est une instance d'un programme en cours d'exécution. Chaque processus est indépendant et possède son propre espace mémoire. Les processus ne partagent pas de mémoire, ce qui rend leur gestion plus sûre mais plus coûteuse en termes de ressources.

b. Threads Les threads sont des unités d'exécution plus légères que les processus. Plusieurs threads peuvent s'exécuter dans un même processus et partagent le même espace mémoire. Cependant, en Python, à cause du **GIL** (Global Interpreter Lock), les threads ne peuvent pas s'exécuter en parallèle pour les tâches CPU-bound (tâches demandant beaucoup de calculs).

Schéma : Différence entre Processus et Threads



```
+------------------+          +----------------+
```

- Chaque **processus** a son propre espace mémoire (totalement isolé).
- Les **threads** partagent la mémoire au sein du même processus.

Gestion de la concurrence : Dans les programmes multi-threads, si plusieurs threads accèdent à des ressources partagées (variables, fichiers, etc.), il faut gérer leur accès avec des **verrous** (`Locks`) pour éviter les conflits.

---

2. Programmation Parallèle avec `multiprocessing`

a. Module `multiprocessing` Le module `multiprocessing` permet de créer des **processus distincts** qui s'exécutent en parallèle, chacun ayant son propre espace mémoire. Cela permet de contourner les limitations du GIL (qui concerne uniquement les threads Python).

Exemple de création de processus avec `multiprocessing` :

```python
from multiprocessing import Process

def worker():
    print("Tâche du processus")

if __name__ == '__main__':
    p = Process(target=worker)
    p.start()  # Démarre le processus
    p.join()   # Attend la fin du processus
```

b. Multiprocessing avec `Pool` `multiprocessing.Pool` permet d'exécuter plusieurs tâches en parallèle sans avoir à gérer individuellement chaque processus. Cela est utile pour exécuter une même fonction sur plusieurs données en parallèle.

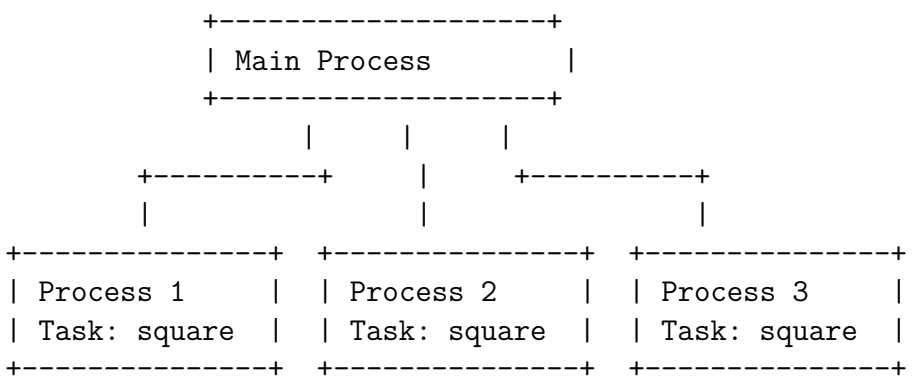
Exemple avec `Pool` :

```python
from multiprocessing import Pool

def carre(x):
    return x ** 2

if __name__ == '__main__':
    with Pool(4) as p:  # Crée un pool de 4 processus
        print(p.map(carre, [1, 2, 3, 4, 5]))
```

Schéma : Multiprocessing avec `Pool`



Dans cet exemple, un pool de processus est créé pour exécuter plusieurs tâches en parallèle (calcul de carrés dans cet exemple).

Gestion de la concurrence avec `multiprocessing` : Pour coordonner les processus et partager des données, Python offre des mécanismes comme les **queues** et **pipes**, ou des **verrous** pour gérer l'accès à la mémoire partagée.

---

3. Programmation Asynchrone avec `asyncio`

a. Concepts de Base La programmation asynchrone permet d'exécuter des tâches sans bloquer l'exécution du programme. Le module `asyncio` en Python gère cela via une **boucle d'événements** qui exécute des **coroutines** (fonctions spéciales définies avec `async def`).

Exemple basique d'une fonction asynchrone :

```
import asyncio

async def hello():
    print("Hello")
    await asyncio.sleep(1)
    print("World")

asyncio.run(hello())
```

b. Exécution Concurrente avec `asyncio.gather()` Avec `asyncio.gather()`, plusieurs coroutines peuvent s'exécuter de manière **concurrente**. Cela permet d'attendre plusieurs tâches en parallèle sans bloquer l'exécution globale du programme.

Exemple avec `asyncio.gather` :

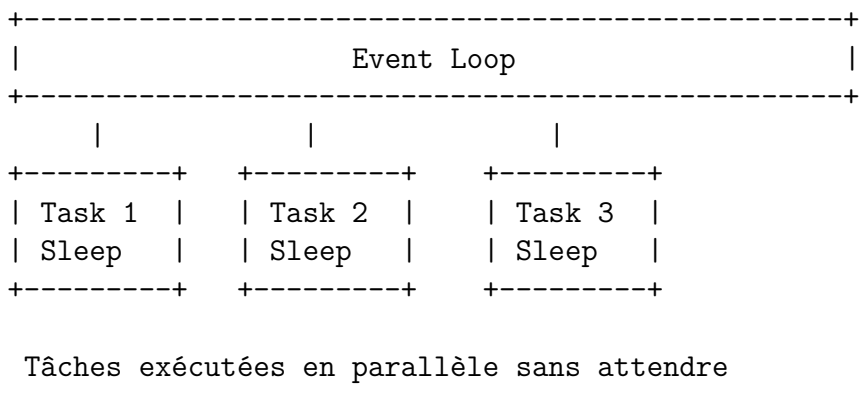
```python
import asyncio

async def task(id, seconds):
    print(f"Tâche {id} démarrée")
    await asyncio.sleep(seconds)
    print(f"Tâche {id} terminée après {seconds} secondes")

async def main():
    await asyncio.gather(task(1, 2), task(2, 3), task(3, 1))

asyncio.run(main())
```

Schéma : Exécution Concurrente avec `asyncio`



Dans cet exemple, trois tâches sont lancées de manière asynchrone. Aucune tâche ne bloque l'exécution des autres, et la boucle d'événements les gère.

Gestion de la concurrence avec `asyncio` : Avec `asyncio`, il n'y a pas de **GIL** à gérer, car tout se passe dans une seule boucle d'événements. Cependant, vous devez être prudent lors de l'accès aux ressources partagées entre coroutines.

---

4. Gérer la Concurrence en Python

a. Threads Utilisez des **threads** lorsque vous devez attendre des réponses d'I/O (entrées/sorties), comme des fichiers ou des connexions réseau. Cela permet de continuer d'autres opérations pendant l'attente.

b. Processus Utilisez **multiprocessing** pour les tâches **CPU-bound**, qui demandent beaucoup de calculs et de puissance de traitement. Le multiprocessing permet de tirer parti de plusieurs cœurs du processeur, chaque processus étant indépendant.

c. Asyncio Utilisez `asyncio` pour les opérations **I/O-bound** où il y a des attentes (comme pour les requêtes réseau), mais peu de calcul intensif.