

Threading vs Multiprocessing

Quelques définitions avant de commencer

Quand on parle de **threading** et de **multiprocessing**, on manipule des concepts qui se ressemblent en surface mais qui fonctionnent très différemment. Voici ce qu'il faut avoir en tête :

Un process, c'est un programme une fois lancé : il est chargé en mémoire, il a ses propres ressources, et surtout son propre espace mémoire. Deux instances du même programme = deux process distincts, qui ne partagent rien.

Un thread, c'est l'unité d'exécution à l'intérieur d'un process. Un process peut avoir plusieurs threads qui tournent en même temps, et ils partagent tous le même espace mémoire du process parent.

Le multithreading, c'est quand un process crée plusieurs threads pour accomplir des tâches en parallèle... ou presque. On dit "en parallèle" mais en Python, les threads ne s'exécutent pas vraiment en même temps à cause du GIL (on y revient juste après). Ils s'alternent très rapidement, ce qui donne l'impression de simultanéité.

Le multiprocessing, c'est le vrai parallélisme : plusieurs process indépendants tournent chacun sur un cœur CPU distinct, avec leur propre mémoire et leur propre interpréteur Python. Là, les choses se passent vraiment en même temps.

Le GIL, le nœud du problème

Le **GIL** (Global Interpreter Lock) est un verrou interne à CPython (l'interpréteur Python standard) qui empêche plusieurs threads d'exécuter du bytecode Python en même temps. Concrètement : même si vous avez 8 cœurs et 8 threads, un seul thread peut "parler" à l'interpréteur à la fois.

Pourquoi ce verrou existe ? En interne, CPython utilise un **compteur de références** pour gérer la mémoire. Chaque objet Python a un compteur qui dit combien de variables pointent vers lui. Quand le compteur tombe à 0, l'objet est libéré de la mémoire. Le problème : si deux threads modifient ce compteur en même temps, sans protection, deux choses peuvent arriver :

- Le compteur descend trop vite → l'objet est libéré alors qu'un thread l'utilise encore → **crash**.
- Le compteur ne descend jamais à 0 → l'objet n'est jamais libéré → **fuite mémoire**.

La solution la plus simple : un **verrou unique global** qui garantit qu'un seul thread touche aux objets Python à la fois. C'est plus simple et plus rapide que de mettre un verrou sur chaque objet individuellement (ce qui provoquerait aussi des deadlocks). C'est un compromis : on sacrifie le vrai parallélisme des threads pour garantir la stabilité de l'interpréteur.

C'est ce mécanisme qui explique pourquoi le multithreading en Python ne donne pas toujours les gains de performance qu'on attend. Mais il reste très utile dans certains cas, on va voir lesquels.

Ce qui change avec Python 3.13 et 3.14

Le GIL est en train de devenir optionnel. Depuis Python 3.13, il existe un build alternatif de l'interpréteur appelé **free-threaded build**, distribué sous le nom **python3.13t**. Dans ce mode, le GIL est désactivé :

les threads peuvent tourner vraiment en parallèle sur plusieurs cœurs, y compris pour des tâches CPU-bound.

Python 3.14 va plus loin : ce mode n'est plus considéré comme expérimental, il est officiellement supporté. La pénalité de performance sur le code classique (single-thread) est passée de ~40% en 3.13 à seulement **5-10%** en 3.14.

Ce que ça change pour vous en pratique :

- Le free-threaded build n'est **pas par défaut**, vous installez `python3.13t` ou `python3.14t` séparément
- Les bibliothèques tierces avec des extensions C (numpy, pandas...) peuvent **réactiver le GIL** si elles n'ont pas encore été adaptées
- Vous pouvez forcer le retour au GIL avec la variable d'environnement `PYTHON_GIL=1` ou l'option `-X gil`

Pour l'instant, dans la grande majorité des projets, les règles qu'on va voir dans ce cours restent valables. Mais gardez en tête que cette frontière entre threading et multiprocessing est amenée à évoluer dans les prochaines versions de Python.

IO-bound vs CPU-bound

Pour comprendre quand utiliser l'un ou l'autre, il faut distinguer deux types de tâches :

- **IO-bound** : la tâche passe son temps à attendre. Attendre une réponse réseau, lire un fichier, interroger une base de données. Le CPU est là, il attend, il ne fait rien.

Exemple concret : vous écrivez un script qui envoie 100 requêtes à une API météo pour récupérer les données de 100 villes. Chaque requête part sur le réseau et votre programme attend la réponse. Le CPU ne calcule rien pendant ce temps, il attend juste que les octets arrivent. C'est une tâche IO-bound.

- **CPU-bound** : la tâche occupe le CPU à 100%. Calculs intensifs, traitement d'images, chiffrement, simulations. Le CPU est le goulot d'étranglement.

Exemple concret : vous écrivez un script qui redimensionne et compresse 10 000 images. Pour chaque image, le CPU doit lire chaque pixel, faire des calculs de transformation, réécrire le résultat. Pas d'attente réseau, pas d'attente disque, le CPU tourne à plein régime en permanence. C'est une tâche CPU-bound.

Cette distinction est clé pour choisir entre threading et multiprocessing.

Le code de base

On va tester six approches différentes avec deux fonctions : une IO-bound (le CPU dort) et une CPU-bound (le CPU compte jusqu'à 200 millions).

```
import time, os
from threading import Thread, current_thread
from multiprocessing import Process, current_process
```

```

COUNT = 200000000
SLEEP = 10

def io_bound(sec):

    pid = os.getpid()
    threadName = current_thread().name
    processName = current_process().name

    print(f"{pid} * {processName} * {threadName} \
        ---> Start sleeping...")
    time.sleep(sec)
    print(f"{pid} * {processName} * {threadName} \
        ---> Finished sleeping...")

def cpu_bound(n):

    pid = os.getpid()
    threadName = current_thread().name
    processName = current_process().name

    print(f"{pid} * {processName} * {threadName} \
        ---> Start counting...")

    while n>0:
        n -= 1

    print(f"{pid} * {processName} * {threadName} \
        ---> Finished counting...")

if __name__=="__main__":
    start = time.time()

    # YOUR CODE SNIPPET HERE

    end = time.time()
    print('Time taken in seconds -', end - start)

```

`if __name__ == "__main__":` — cette ligne est indispensable avec `multiprocessing`. Quand Python crée un nouveau process, il réimporte le fichier principal. Sans cette protection, le code de lancement serait ré-exécuté dans chaque process fils, créant une boucle infinie de processus. Avec le `threading`, ce n'est pas obligatoire, mais c'est une bonne habitude à prendre systématiquement.

`os.getpid()` retourne l'identifiant du process en cours. `current_thread().name` et `current_process().name` vous donnent le nom du thread et du process actifs. Ces informations vont nous permettre de voir exactement qui fait quoi.

Pour chaque partie ci-dessous, remplacez `# YOUR CODE SNIPPET HERE` par le snippet correspondant.

Partie 1 : IO-bound en séquentiel

```
# Code snippet pour la Partie 1
io_bound(SLEEP)
io_bound(SLEEP)
```

Le MainProcess appelle `io_bound()` deux fois, l'une après l'autre. Chaque appel attend 10 secondes. Total : **20 secondes**. Pas de surprise, on n'a fait aucun effort pour paralléliser.

Partie 2 : IO-bound avec threading

```
# Code snippet pour la Partie 2
t1 = Thread(target = io_bound, args =(SLEEP, ))
t2 = Thread(target = io_bound, args =(SLEEP, ))
t1.start()
t2.start()
t1.join()
t2.join()
```

Cette fois, on crée deux threads. Les deux démarrent presque en même temps, et pendant que Thread-1 dort, Thread-2 dort aussi, le CPU n'a rien à faire de toute façon. Résultat : **~10 secondes** au lieu de 20.

`Thread(target=io_bound, args=(SLEEP,))` : `target` est la fonction à exécuter dans le thread, `args` est un tuple contenant les arguments à lui passer. Le tuple doit avoir une virgule finale quand il n'y a qu'un seul élément : `(SLEEP,)` et non `(SLEEP)` (sans virgule, Python interprète les parenthèses comme un simple groupement, pas un tuple).

`t.start()` : lance réellement l'exécution du thread. Avant ce point, le thread est créé mais pas démarré.

`t.join()` sert à dire au programme principal d'attendre que le thread ait fini avant de continuer. Sans ça, le programme principal pourrait se terminer avant les threads, ce qui peut provoquer des comportements imprévisibles.

`Thread` et `Process` ont exactement la même interface : `target`, `args`, `start()`, `join()`. Ce que vous apprenez pour l'un s'applique à l'autre.

Le threading est parfait pour les tâches IO-bound : pendant le temps d'attente d'un thread, Python peut passer la main à un autre.

Partie 3 : CPU-bound en séquentiel

```
# Code snippet pour la Partie 3
cpu_bound(COUNT)
cpu_bound(COUNT)
```

Même principe qu'en partie 1, mais avec `cpu_bound()`. Le CPU compte de 200 millions à 0 deux fois de suite. Le temps total dépend de la puissance de votre machine : comptez entre **8 et 25 secondes** selon le processeur.

Partie 4 : CPU-bound avec threading, attention, piège !

```
# Code snippet pour la Partie 4
t1 = Thread(target = cpu_bound, args =(COUNT, ))
t2 = Thread(target = cpu_bound, args =(COUNT, ))
t1.start()
t2.start()
t1.join()
t2.join()
```

On applique la même recette qu'en partie 2. On a deux threads, donc on pourrait s'attendre à diviser le temps par deux. Mais non : le résultat est **à peu près identique au séquentiel**, voire légèrement plus lent.

C'est le GIL en action. Thread-1 démarre et verrouille le GIL. Thread-2 attend. Thread-1 libère le verrou régulièrement pour que Python puisse "switcher", Thread-2 prend la main, et ainsi de suite. Les deux threads ne tournent jamais vraiment en même temps — ils se relaient sur un seul cœur.

Sur les anciennes versions de Python, ce ping-pong entre threads générait un overhead important et le résultat pouvait être **plus lent** qu'en séquentiel. Depuis Python 3.12+, le mécanisme de switching du GIL a été optimisé, donc l'overhead est réduit. Mais le constat reste le même : aucun gain de performance.

Conclusion : **le threading n'apporte rien pour les tâches CPU-bound. Au mieux le temps est identique, au pire il est dégradé.**

Partie 5 : CPU-bound avec multiprocessing

```
# Code snippet pour la Partie 5
p1 = Process(target = cpu_bound, args =(COUNT, ))
p2 = Process(target = cpu_bound, args =(COUNT, ))
p1.start()
p2.start()
p1.join()
p2.join()
```

Même interface que les threads, mais on crée des `Process` au lieu de `Thread`. Chaque process est totalement indépendant : il a son propre GIL, son propre interpréteur, sa propre mémoire. Ils tournent en parallèle sur deux cœurs distincts.

Résultat : **le temps d'un seul appel** (environ la moitié du séquentiel), parce que les deux tournent vraiment en même temps.

Si vous ouvrez le gestionnaire de tâches pendant l'exécution, vous verrez 3 instances de Python : le process principal, Process-1 et Process-2.

Notez que l'ordre des `print` dans la console peut varier : les process sont indépendants et ne se synchronisent pas sur l'affichage.

Partie 6 : IO-bound avec multiprocessing

```
# Code snippet pour la Partie 6
p1 = Process(target = io_bound, args =(SLEEP, ))
p2 = Process(target = io_bound, args =(SLEEP, ))
p1.start()
p2.start()
p1.join()
p2.join()
```

Ça fonctionne aussi bien que le threading pour les tâches IO-bound (~10 secondes). Mais ce n'est pas le bon outil ici : créer un process est beaucoup plus coûteux que créer un thread (plus de mémoire, plus de temps de démarrage). Pour une tâche qui ne fait qu'attendre, c'est du gaspillage.

Résumé : quoi utiliser et quand

Situation	Outil recommandé
Tâches IO-bound (réseau, fichiers, BDD)	Threading
Tâches CPU-bound (calculs, traitement de données)	Multiprocessing
IO-bound avec beaucoup de connexions simultanées	asyncio (on verra ça après)

La règle à retenir : **threading pour les temps d'attente, multiprocessing pour la puissance de calcul.**

Cas d'usage concrets

Threading :

- Faire plusieurs requêtes HTTP en même temps
- Lire/écrire des fichiers en parallèle
- Maintenir une interface graphique réactive pendant un traitement en arrière-plan
- Gérer plusieurs connexions réseau simultanées

Multiprocessing :

- Traitement d'images ou de vidéos
- Calculs scientifiques ou simulations
- Analyse de gros volumes de données
- Tout ce qui ferait monter un cœur CPU à 100%