Remarque préliminaire: ce document a été réalisé avec l'IA Gemini.

En Python, la façon standard et recommandée de créer de nouveaux processus qui s'exécutent en parallèle est d'utiliser la classe Process du module `multiprocessing`.

Bien que `multiprocessing` puisse utiliser `fork()` en coulisse sur les systèmes Unix, il offre une abstraction de plus haut niveau qui fonctionne de manière transparente sur différents systèmes d'exploitation (Windows, macOS, Linux, etc.) sans que vous ayez à gérer les subtilités du fork().

Comment ça marche
Au lieu de dupliquer l'intégralité du processus parent comme le fait `fork()`, vous définissez la fonction que le nouveau processus doit exécuter.

Voici comment la classe Process est utilisée :

Définir la cible : Vous créez une fonction (la tâche) que le nouveau processus doit exécuter.

Créer l'objet Process : Vous instanciez la classe Process, en lui passant la fonction cible.

Démarrer : Vous appelez la méthode `.start()` pour lancer l'exécution du nouveau processus.

```
from multiprocessing import Process
import os

def ma_tache():
    # Ce code est exécuté par le processus enfant
    print(f"PID Enfant: {os.getpid()}")
    print(f"PID Parent de l'enfant: {os.getppid()}")


print(f"PID Principal (Parent): {os.getpid()}")

# 1. Création de l'objet Process
p = Process(target=ma_tache)

# 2. Démarrage du processus (équivalent conceptuel de fork/exec)
p.start()

# 3. Attente de la fin du processus enfant
p.join()

print("Fin du programme principal.")
```

Pourquoi pas simplement os.fork() ?
La fonction os.fork() existe bel et bien en Python (elle est disponible sur les systèmes POSIX comme Linux et macOS), mais son utilisation est souvent découragée au profit de multiprocessing pour plusieurs raisons :

Portabilité : os.fork() ne fonctionne pas sur Windows. multiprocessing utilise des méthodes alternatives (comme l'exécution d'un nouveau processus via spawn) pour assurer la portabilité.

Sécurité des ressources : fork() pose des problèmes complexes lorsque le processus parent a des threads actifs ou des verrous ouverts, ce qui peut conduire à des impasses ou des corruptions de données. Le module multiprocessing gère mieux ces problèmes.

En résumé, si vous souhaitez créer un nouveau processus en Python pour profiter du parallélisme, utilisez from multiprocessing import Process.

Un autre exemple

```
from multiprocessing import Process
from os import getppid

# function exécutée par le processus enfant
def recupere_pid():
    # on récupère le pid du processus parent
    pid = getppid()
    # on affiche
    print(pid)



# On configure le processus enfant
enfant = Process(target=recupere_pid)
# On démarre le processus enfant
enfant.start()
print(enfant.is_alive())
# On attend que le processus enfant se termine
enfant.join()
print(enfant.is_alive())
```

L'objectif est de :

- Démarrer un processus enfant (enfant) depuis le processus principal du script.
- Dans le processus enfant, récupérer et afficher l'ID du processus parent (le processus principal).
- Afficher l'état de vie du processus enfant avant et après son exécution.

Explication étape par étape:

- from multiprocessing import Process: Importe la classe Process pour la création de processus.
- from os import getppid: Importe la fonction getppid(), qui, lorsqu'elle est appelée, renvoie l'ID du processus parent (PPID) de l'appelant.
- def recupere_pid():: Définit la fonction qui sera exécutée par le processus enfant.
- pid = getppid(): Appelée par le processus enfant, cette ligne stocke le PID du processus qui l'a démarré (le processus principal).
- print(pid): Affiche cet ID de processus parent.

Le reste du code est exécuté par le processus principal du script.

- enfant = Process(target=recupere_pid): Crée un nouvel objet Process nommé enfant. Il est configuré pour exécuter la fonction recupere_pid lorsqu'il sera démarré.
- enfant.start(): Démarre le processus enfant. Le système d'exploitation crée un nouveau processus, et celui-ci commence à exécuter la fonction recupere_pid, ce qui conduit à l'affichage du PID du processus principal.
- print(enfant.is_alive()): Affiche l'état du processus enfant immédiatement après son démarrage. À ce stade, le processus enfant est en cours d'exécution.

  Résultat attendu : True
- enfant.join(): Bloque l'exécution du processus principal et attend que le processus enfant se termine (c'est-à-dire qu'il finisse d'exécuter recupere_pid).

- print(enfant.is_alive()): Affiche l'état du processus enfant après qu'il a terminé son exécution et que la méthode join() est revenue.

  Résultat attendu : False

```
from multiprocessing import Process
from multiprocessing import parent_process

# fonction exécutée par le processus enfant
def task():
    # récupérer le processus parent
    parent = parent_process()
    # récupérer le pid du processus parent
    pid = parent.pid
    # report the pid
   print(pid)


# configuration du processus enfant
child = Process(target=task)
#démarrer le processus enfant
child.start()

# attendre la fin d'exécution du processus enfant
child.join()
# récupérer le pid du processus enfant
pid1 = child.pid
# afficher le pid
print(pid1)
```