La programmation concurrente avec Python

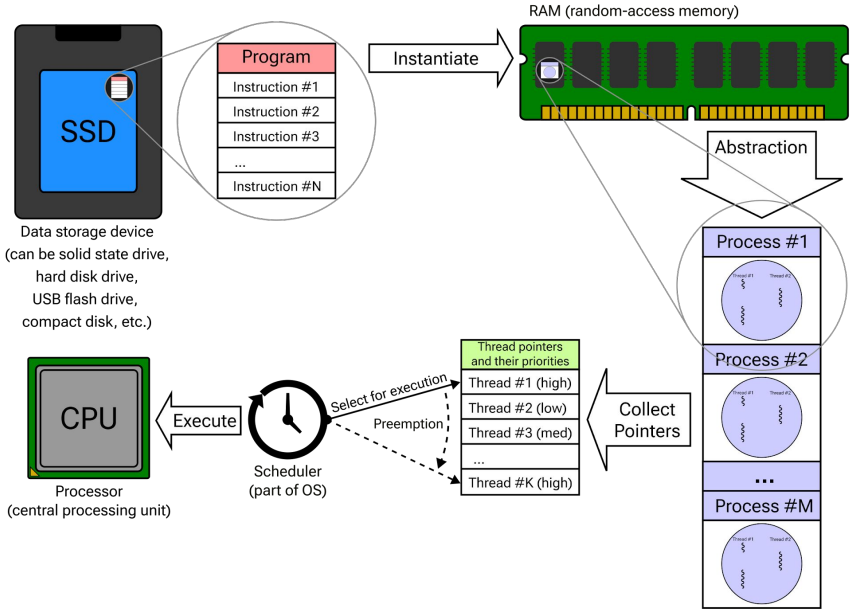
Concepts de Base

Définition : La programmation concurrente permet l'exécution de plusieurs tâches en parallèle pour améliorer les performances et la réactivité des applications.

Processus : Une instance d'un programme en cours d'exécution, avec son propre espace mémoire.

Threads : La plus petite unité d'exécution dans un processus, partageant le même espace mémoire.

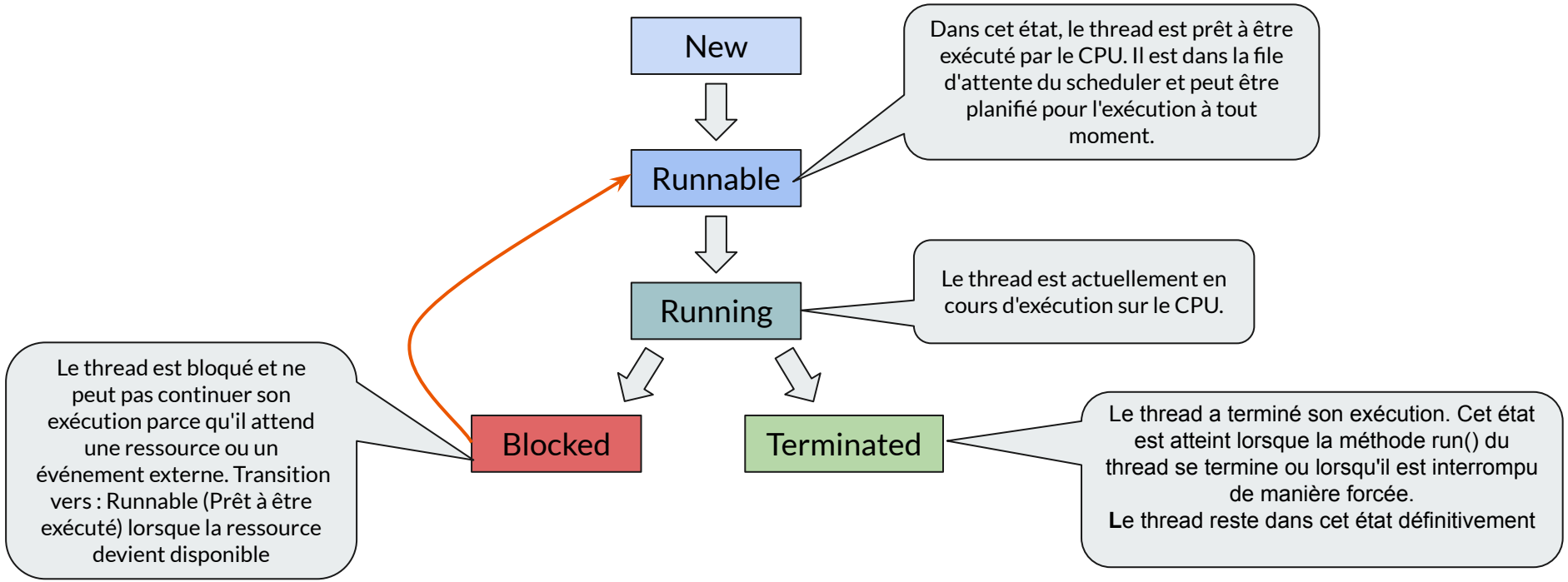
Processus et threads



Multithreading

```python
import threading  # Importation du module threading pour la gestion des threads


1 usage
def print_numbers():
    # Définition de la fonction print_numbers qui imprimera les nombres de 1 à 5
    for i in range(1, 6):
        print(i)  # Imprime le nombre courant dans la boucle

# Création d'un thread pour exécuter la fonction print_numbers
thread = threading.Thread(target=print_numbers)
thread.start()  # Démarre l'exécution du thread, appelant ainsi print_numbers
thread.join()  # Attend que le thread se termine avant de continuer l'exécution du programme principal
```

Cycle d'un thread



Le verrou et partage de ressources communes

Qu'est-ce qu'un verrou ?

Un verrou est un outil de synchronisation qui permet de s'assurer qu'une seule unité d'exécution (un thread) peut accéder à une ressource partagée à la fois. Cela empêche les problèmes de concurrence tels que les conditions de course, où plusieurs threads tentent de modifier les mêmes données simultanément, ce qui peut entraîner des résultats imprévisibles ou des corruptions de données.

Fonctionnement d'un verrou

1. **Acquisition d'un verrou** :
   - Lorsqu'un thread souhaite accéder à une ressource partagée, il doit d'abord acquérir le verrou associé.
   - Si le verrou est disponible, le thread l'acquiert et peut accéder à la ressource.
   - Si le verrou est déjà détenu par un autre thread, le thread est mis en attente jusqu'à ce que le verrou soit libéré.
2. **Libération d'un verrou** :
   - Une fois que le thread a terminé son travail avec la ressource partagée, il libère le verrou.
   - Cela permet à d'autres threads qui sont en attente d'acquérir le verrou et d'accéder à la ressource.

GIL (Global Interpreter Lock)

“Le système de gestion de la mémoire de Python suit l'utilisation des objets en comptabilisant le nombre de références à chaque objet. Lorsque le nombre de références d'un objet tombe à zéro, l'objet est destiné à être supprimé. Comme Python a été créé à une époque où les systèmes multiprocesseurs étaient rares et où les processeurs multicœurs n'existaient pas, ce mécanisme de comptage des références n'est pas sûr pour les threads. Au lieu de cela, Python assure la sécurité des threads en n'autorisant qu'un seul thread à accéder à un objet à la fois. C'est la raison d'être de GIL.”

Python enfin prêt à faire sauter le verrou GIL, **Serdar Yegulalp, Le Monde Informatique, 14 août 2023**

Solutions de contournement à GIL

- Utiliser le module multiprocessing au lieu de threading. Chaque processus a son propre espace de mémoire et son propre GIL, permettant une exécution parallèle réelle.
- Écrire des extensions en C/C++ qui libèrent le GIL lors de l'exécution de tâches CPU-bound.
- Utiliser des implémentations de Python sans GIL, comme Jython ou IronPython, bien que celles-ci puissent ne pas être compatibles avec toutes les bibliothèques Python.

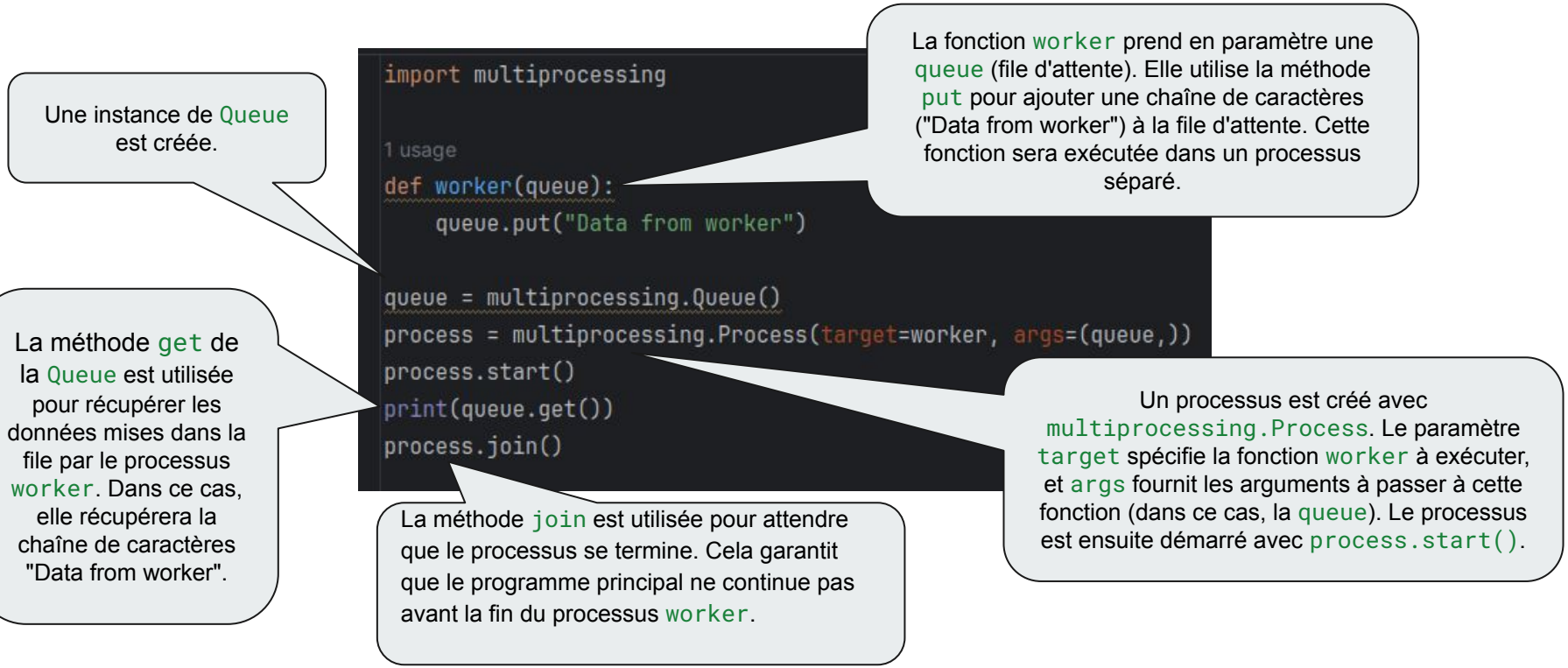
Exemple d'utilisation de multiprocessing

```python
import multiprocessing

2 usages
def cpu_bound_task():
    count = 0
    for i in range(10**6):
        count += i
    print("Task completed")

# Création de deux processus
process1 = multiprocessing.Process(target=cpu_bound_task)
process2 = multiprocessing.Process(target=cpu_bound_task)

# Démarrage des processus
process1.start()
process2.start()

# Attendre que les processus se terminent
process1.join()
process2.join()
```

Communication inter-processus et synchronisation

- Le module `multiprocessing` en Python permet de créer des processus parallèles, semblables aux threads, mais avec des capacités d'isolation mémoire, ce qui les rend plus robustes pour les tâches gourmandes en CPU.
- Mais, chaque processus a sa propre mémoire, comment les faire communiquer entre-eux ?
- Le module `multiprocessing` fournit plusieurs moyens pour faciliter cette communication, notamment les `Queues` (files d'attente) et les verrous.

Multiprocessing et queue



Multiprocessing et verrou

L'accès simultané aux ressources partagées peut causer des incohérences, l'utilisation des verrous permet d'y synchroniser l'accès.

Crée un objet lock. Un lock est un mécanisme de synchronisation utilisé pour s'assurer que seulement un thread à la fois peut exécuter un morceau de code critique, évitant ainsi les problèmes de concurrence.

```
import threading
lock = threading.Lock()

1 usage
def thread_safe_print():
    with lock:
        print("This is thread-safe")

thread = threading.Thread(target=thread_safe_print)
thread.start()
thread.join()
```

Utilise un contexte de gestion (`with` statement) pour acquérir le lock avant d'exécuter le bloc de code à l'intérieur. Lorsque le bloc de code est terminé, le lock est automatiquement libéré.

Crée un thread en spécifiant que la fonction `thread_safe_print` doit être exécutée par ce thread.

join() bloque le thread principal jusqu'à ce que le thread spécifié (`thread`) ait terminé son exécution.

Introduction aux Futures

Définition : Une future est un objet qui encapsule une opération qui sera exécutée dans le futur. Le résultat de cette opération n'est pas immédiatement disponible, mais l'objet future permet de récupérer ce résultat une fois l'opération terminée.

Les futures sont particulièrement utiles dans la programmation concurrente, car elles permettent de gérer des opérations asynchrones et de vérifier leur état ou de récupérer leurs résultats ultérieurement.

Vérification de l'état, quelques méthodes

Méthode	Description	Exemple
done()	Renvoie `True` si la future est terminée, indépendamment de son succès ou échec.	future.done()
running()	Renvoie `True` si la future est actuellement en cours d'exécution.	future.running()
cancelled()	Renvoie `True` si la future a été annulée avant d'avoir commencé.	future.cancelled()
result(timeout=None)	Bloque jusqu'à ce que la future soit terminée, puis renvoie le résultat de la tâche. Lève une exception si la tâche a levé une exception. Le paramètre `timeout` (facultatif) spécifie le nombre de secondes à attendre avant de lever une exception `TimeoutError`.	future.result()
exception(timeout=None)	Bloque jusqu'à ce que la future soit terminée, puis renvoie l'exception levée par la tâche, ou `None` si la tâche a réussi. Le paramètre `timeout` (facultatif) spécifie le nombre de secondes à attendre avant de lever une exception `TimeoutError`.	future.exception()
add_done_callback(fn)	Ajoute une fonction de rappel qui sera appelée lorsque la future sera terminée.	future.add_done_callback(callback_function)

Asynchrone

Définition : La programmation asynchrone permet d'exécuter des tâches de manière non bloquante en utilisant des boucles d'événements. Cela signifie que vous pouvez effectuer d'autres opérations pendant que vous attendez que certaines tâches se terminent (par exemple, des opérations d'I/O).

Bibliothèque asyncio : les coroutines

```python
import asyncio

2 usages
async def say_hello():
    print("Hello")
    await asyncio.sleep(1)
    print("World")

1 usage
async def main():
    task1 = asyncio.create_task(say_hello())
    task2 = asyncio.create_task(say_hello())
    await task1
    await task2

asyncio.run(main())
```

Lorsque vous exécutez ce code, voici ce qui se passe :

- main crée deux tâches (task1 et task2) pour exécuter say_hello.
- Les deux tâches commencent à s'exécuter simultanément.
- Chaque tâche affiche "Hello", puis attend 1 seconde de manière non bloquante.
- Après 1 seconde, chaque tâche affiche "World".
- main attend que les deux tâches soient terminées avant de se terminer elle-même.