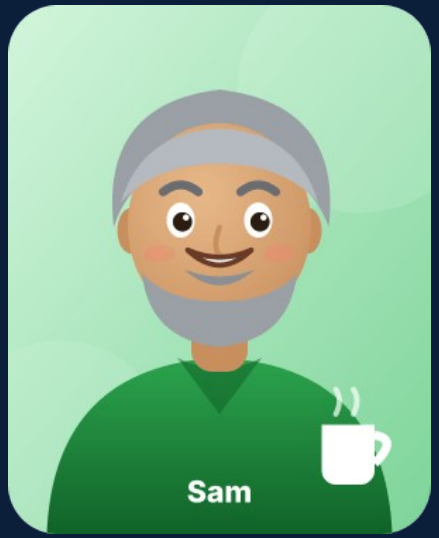
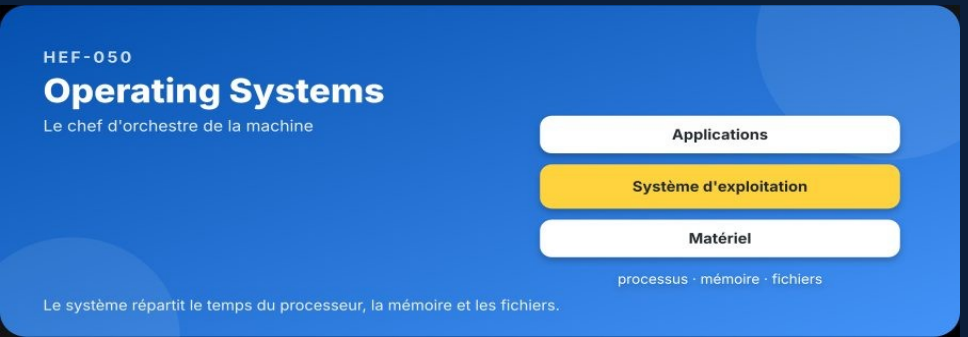
HEF-050 · MISSION 03 / 10

Threads et concurrence

Faire plusieurs choses « en même temps » dans un seul programme, c'est puissant et dangereux.





Badge : Jongleur de fils

80 min · intermédiaire

L'histoire

Le serveur de HashBox gère le paiement de deux membres en même temps. Quelques jours plus tard, le comptage des ventes est faux : deux ventes ont disparu.

> *« Deux fils d'exécution ont lu le même total en même temps, dit Sam. Chacun a ajouté sa vente à l'ancienne valeur, et le second a écrasé le premier. »*



Tu découvres les threads, les conditions de course et les verrous.

Ta mission

À la fin, je serai capable de…

1. distinguer processus et thread
2. expliquer une condition de course avec un exemple
3. protéger une section critique avec un verrou (`Lock`)
4. expliquer le GIL de Python et ses conséquences
5. choisir entre threads, processus et asynchrone selon la charge

Défi d'échauffement

Défi d'échauffement

Deux caissiers partagent le même cahier de comptes. Chacun lit le total (100 €), ajoute sa vente (10 €), et réécrit le total. S'ils font cela **exactement en même temps**, quel total obtient-on dans le cahier ? Quel total aurait dû être écrit ? Comment éviter ce problème ?

Le concept

Thread ou processus ?

Un **thread** est un fil d'exécution à l'intérieur d'un processus. Les threads d'un même processus **partagent la mémoire** (tas, variables globales) mais ont chacun leur pile et leurs registres.

	Processus	Thread
Mémoire	séparée	partagée
Création	plus coûteuse	plus légère
Communication	tubes, fichiers, sockets	variables partagées
Isolation	forte	faible
Risque	:	conditions de course

Le concept

La condition de course

Une **condition de course** (*race condition*) survient quand le résultat dépend de l'ordre d'exécution de plusieurs threads qui accèdent à une donnée partagée.

```
import threading, time

compteur = 0

def incrementer(n):
    global compteur
    for _ in range(n):
        valeur = compteur          # lire
        time.sleep(0)              # cède la main (rend la course visible)
        compteur = valeur + 1      # écrire

fils = [threading.Thread(target=incrementer, args=(1000,)) for _ in range(4)]
```

Le concept

La condition de course (suite)

```
for f in fils: f.start()
for f in fils: f.join()
print(compteur, "au lieu de 4000")
```

Le résultat est presque toujours **inférieur à 4000** : des incrémentations se sont perdues, car deux threads ont lu la même valeur.

Le verrou (lock)

Un **verrou** garantit qu'un seul thread à la fois exécute une **section critique**.

```
import threading

compteur = 0
verrou = threading.Lock()
```

Le concept

Le verrou (lock) (suite)

```
def incrementer(n):
    global compteur
    for _ in range(n):
        with verrou:                # un seul thread entre à la fois
            valeur = compteur
            compteur = valeur + 1

fils = [threading.Thread(target=incrementer, args=(1000,)) for _ in range(4)]
for f in fils: f.start()
for f in fils: f.join()
print(compteur)    # 4000
```

Le prix : les threads s'attendent. Une section critique **courte** limite ce coût.

Le GIL de Python

Le concept

Le GIL de Python (suite)

En Python (implémentation CPython), le **GIL** (*Global Interpreter Lock*) fait qu'**un seul thread exécute du bytecode Python à la fois**. Conséquences :

- les threads n'accélèrent pas un calcul **purement CPU** en Python ;
- ils sont utiles pour les **entrées/sorties** (réseau, disque), car un thread bloqué libère le GIL ;
- pour paralléliser du calcul : `multiprocessing` (plusieurs processus) ou des bibliothèques en C.

Quel outil choisir ?

Charge	Outil
Beaucoup d'attentes réseau/disque	threads ou asynchrone (`asyncio`)
Calcul intensif sur plusieurs cœurs	processus (`multiprocessing`)
Isolation forte	processus

Le concept

Quel outil choisir ? (suite)

Charge	Outil
Partage de données	threads (avec verrous) ou file thread-safe (`queue.Queue`)

🧠 ***Astuce*** *: le meilleur verrou est celui dont on n'a pas besoin. Préfère échanger des* ***messages*** *(files) plutôt que partager des variables.*

Exemple pas à pas

Problème : télécharger (simuler) 8 pages web, d'abord en séquence puis avec des threads.

```
import threading, time

def telecharger(i, resultats):
    time.sleep(0.2)               # simule l'attente réseau (libère le GIL)
    resultats[i] = f"page {i}"

def sequentiel(n):
    resultats = {}
    for i in range(n):
        telecharger(i, resultats)
    return resultats

def avec_threads(n):
    resultats = {}
    fils = [threading.Thread(target=telecharger, args=(i, resultats)) for i in ran
   ge(n)]
```

Exemple pas à pas (suite)

```
    for f in fils: f.start()
    for f in fils: f.join()
    return resultats

for f in (sequentiel, avec_threads):
    t = time.perf_counter()
    r = f(8)
    print(f.__name__, round(time.perf_counter() - t, 2), "s", len(r), "pages")
```

Lecture : environ 1,6 s en séquence (8 × 0,2 s) contre environ 0,2 s avec threads. Pour des **attentes**, les threads aident beaucoup.

Écriture dans un dictionnaire partagé : ici chaque thread écrit une clé **différente**, donc pas de conflit. Si tous écrivaient la même clé ou faisaient `total += …`, il faudrait un verrou.

À toi, avec aide

Un compte bancaire est partagé entre plusieurs threads.

1. Écris une classe `Compte` avec `deposer(montant)` et `solde()`, **sans** protection. Montre avec 4 threads de 1 000 dépôts de 1 € (avec `time.sleep(0)` dans le dépôt) que le solde final est erroné.
2. Corrige avec un verrou et vérifie que le solde est exactement 4 000.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Dans `deposer`, sépare la lecture et l'écriture avec `time.sleep(0)` entre les deux.

Indice 2

Crée un `threading.Lock()` dans `__init__`.

Indice 3

Utilise `with self._verrou:` autour de la lecture et de l'écriture.

À toi, avec aide · réponse

Réponse

```
import threading, time

class CompteFragile:
    def __init__(self):
        self._solde = 0
    def deposer(self, m):
        v = self._solde
        time.sleep(0)
        self._solde = v + m
    def solde(self):
        return self._solde

class CompteSur(CompteFragile):
    def __init__(self):
        super().__init__()
        self._verrou = threading.Lock()
    def deposer(self, m):
        with self._verrou:
            super().deposer(m)
```

Réponse (suite)

```
def tester(compte):
    def travail():
        for _ in range(1000):
            compte.deposer(1)
    fils = [threading.Thread(target=travail) for _ in range(4)]
    for f in fils: f.start()
    for f in fils: f.join()
    return compte.solde()

print("sans verrou :", tester(CompteFragile()))    # presque toujours < 4000
r = tester(CompteSur())
print("avec verrou :", r)
assert r == 4000
```

À toi, en solo

Producteur-consommateur. Un thread producteur met 20 tâches dans une `queue.Queue` ; 3 threads consommateurs les traitent (chaque tâche = un calcul + une petite pause) et rangent le résultat dans une liste partagée. Utilise un signal de fin (par exemple `None`) pour arrêter les consommateurs. Vérifie : les 20 résultats sont présents, aucun n'est perdu, et le temps total est réduit par rapport à un seul consommateur.

À toi, en solo · réponse

Pistes de correction

```
import queue, threading, time

def traiter(nb_consommateurs):
    q = queue.Queue()
    resultats = []
    verrou = threading.Lock()

    def producteur():
        for i in range(20):
            q.put(i)
        for _ in range(nb_consommateurs):
            q.put(None)                    # un signal de fin par consommateur

    def consommateur():
        while True:
            tache = q.get()
            if tache is None:
                break
            time.sleep(0.02)               # simule le travail
```

À toi, en solo · réponse

Pistes de correction (suite)

```
            with verrou:
                resultats.append(tache * tache)

    fils = [threading.Thread(target=producteur)] + [threading.Thread(target=consom
   mateur) for _ in range(nb_consommateurs)]
    t = time.perf_counter()
    for f in fils: f.start()
    for f in fils: f.join()
    return sorted(resultats), time.perf_counter() - t

for n in (1, 3):
    r, duree = traiter(n)
    assert r == [i * i for i in range(20)]
    print(n, "consommateur(s) :", round(duree, 2), "s")
```

`queue.Queue` gère elle-même la synchronisation : on échange des messages au lieu de partager des variables.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Variable partagée sans verrou	Résultats faux, aléatoires	Verrou ou file thread-safe
Section critique trop longue	Les threads s'attendent, plus de gain	Garder le verrou le plus court possible
Deux verrous dans le désordre	Interblocage (deadlock)	Toujours les prendre dans le même ordre
Threads pour du calcul pur en Python	Le GIL empêche le gain	`multiprocessing`

Mini-boss

Mini-boss

Compteur de mots concurrent. Tu as 8 « fichiers » (chaînes de texte). Écris trois versions de `compter_mots_global(fichiers)` qui renvoie le total des occurrences de chaque mot :

1. séquentielle ;
2. avec threads et un dictionnaire partagé protégé par un verrou ;
3. avec threads où chacun construit **son propre** dictionnaire, puis fusion à la fin (sans verrou pendant le travail).

Vérifie que les trois donnent le même résultat. Mesure le temps de chacune (ajoute `time.sleep(0.05)` par fichier pour simuler la lecture). Explique laquelle est la plus propre et pourquoi.

Mini-boss · réponse

Pistes de correction

```
import threading, time
from collections import Counter

fichiers = [("le chat et le chien " * 50) for _ in range(8)]

def lire(f):
    time.sleep(0.05)
    return f.split()

def sequentiel(fs):
    c = Counter()
    for f in fs:
        c.update(lire(f))
    return c

def avec_verrou(fs):
    total, verrou = Counter(), threading.Lock()
    def travail(f):
        mots = lire(f)
```

Mini-boss · réponse

Pistes de correction (suite)

```
        with verrou:
            total.update(mots)
    fils = [threading.Thread(target=travail, args=(f,)) for f in fs]
    for t in fils: t.start()
    for t in fils: t.join()
    return total

def sans_partage(fs):
    locaux = [None] * len(fs)
    def travail(i, f):
        locaux[i] = Counter(lire(f))
    fils = [threading.Thread(target=travail, args=(i, f)) for i, f in enumerate(fs
   )]
    for t in fils: t.start()
    for t in fils: t.join()
    total = Counter()
    for c in locaux:
        total.update(c)
    return total
```

Mini-boss · réponse

Pistes de correction (suite)

```
ref = sequentiel(fichiers)
for f in (sequentiel, avec_verrou, sans_partage):
    t = time.perf_counter()
    r = f(fichiers)
    assert r == ref
    print(f.__name__, round(time.perf_counter() - t, 2), "s")
```

La version sans partage évite les verrous : chaque thread travaille sur ses données, et la fusion se fait au calme. C'est le schéma « map-reduce » à petite échelle.

Quiz éclair

Quiz éclair

1. Les threads d'un même processus partagent :

A. la pile
B. la mémoire
C. les registres

2. Une condition de course est :

A. un bug de syntaxe
B. un type de verrou
C. un résultat qui dépend de l'ordre d'exécution

3. Un verrou sert à :

A. n'autoriser qu'un thread à la fois dans une section critique

Quiz éclair

Quiz éclair (suite)

B. accélérer le programme
C. partager des fichiers

4. 🔁 *Révision mission 02* : Un code de sortie 0 signifie généralement :

A. erreur
B. succès
C. attente

5. 🔁 *Révision mission 01* : Le noyau est :

A. une application
B. un type de fichier
C. le cœur du système qui contrôle le matériel

Quiz éclair · réponse

Réponses

1. **B**.
2. **C**.
3. **A**.
4. **B**.
5. **C**.

Je réfléchis

1. Dans ta vie, où deux personnes partagent-elles une « ressource » (calendrier, compte) qui nécessite une règle pour éviter les conflits ?
2. Pourquoi vaut-il mieux éviter de partager de l'état entre threads quand c'est possible ?

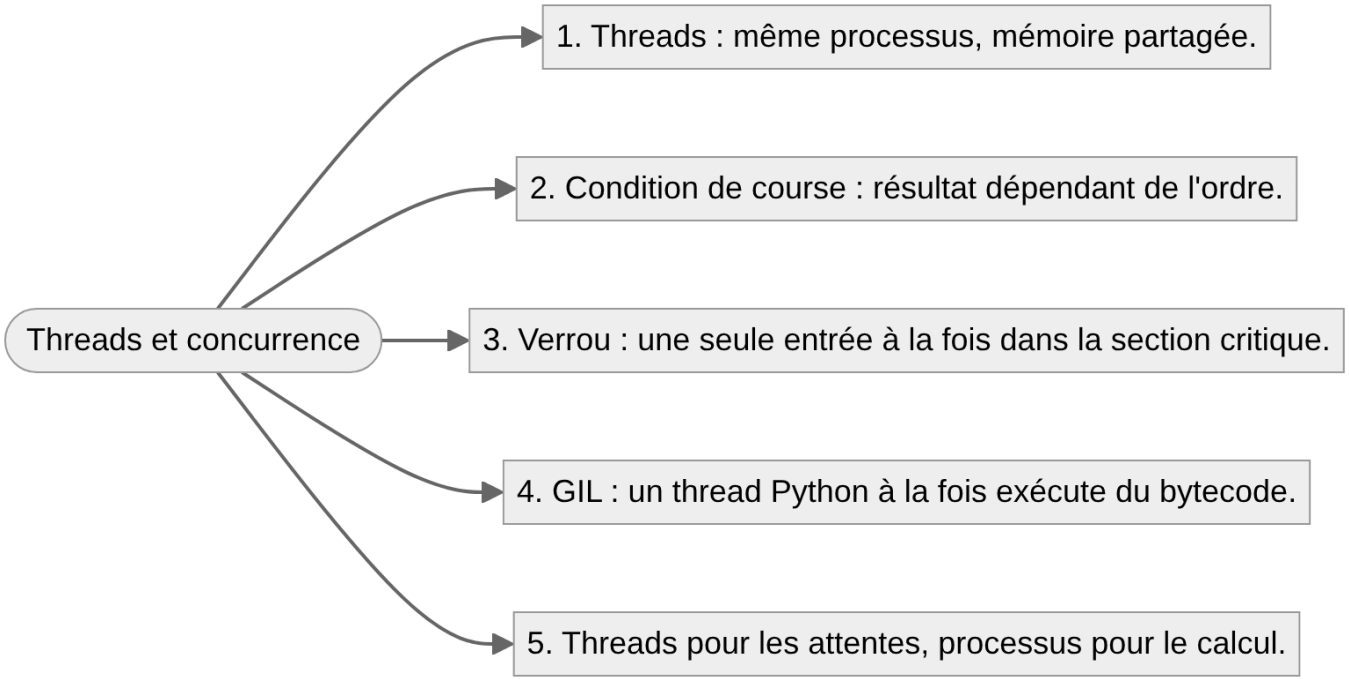
À retenir

À retenir

1. Threads : même processus, mémoire partagée.
2. Condition de course : résultat dépendant de l'ordre.
3. Verrou : une seule entrée à la fois dans la section critique.
4. GIL : un thread Python à la fois exécute du bytecode.
5. Threads pour les attentes, processus pour le calcul.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Jongleur de fils

Prochaine mission : L'ordonnancement

